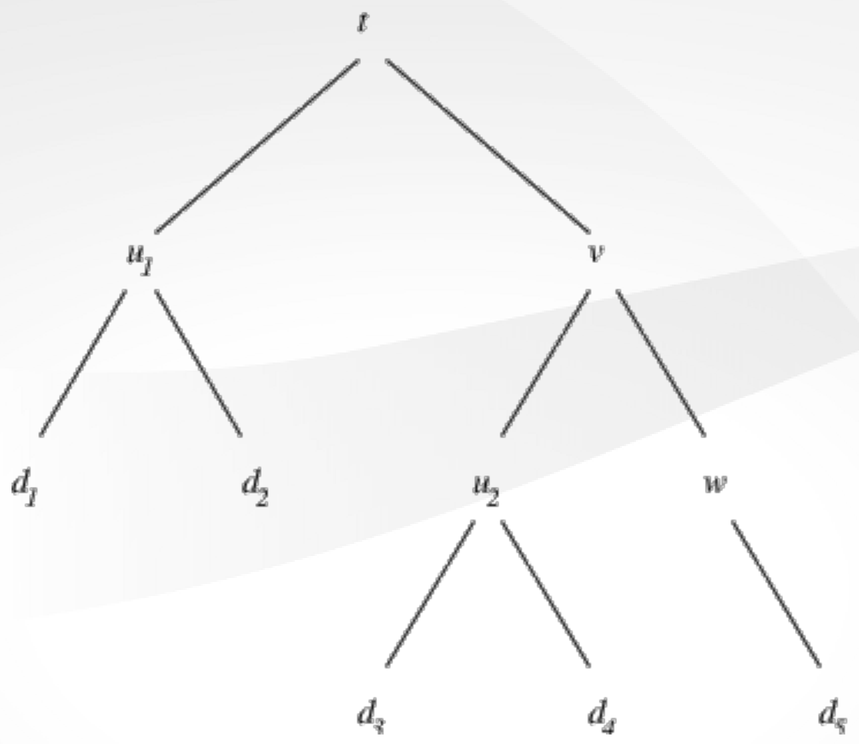
Easy and effficient programming with IRPF90

Anthony Scemama <scemama@irsamc.ups-tlse.fr>
Michel Caffarel <michel.caffarel@irsamc.ups-tlse.fr>

Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

Introduction

- A program is a function of its input data: output = program(input)
- A program can be represented as a tree where:
- the vertices are the variables
- the edges represent the relation '*depends on*'
- The root of the tree is the output of the program
- The leaves are the input data



u (x, y) = x + y + 1
v (x, y) = x + y + 2
w (x) = x + 3
t (x, y) = x + y + 4
This production tree computes
t(u(d_1, d_2), v(u(d_3, d_4), w(d_5)))

Usual programming

```
program exemple_1
   implicit none
   integer :: d1,d2,d3,d4,d5  ! Input data
   integer :: u1, u2, w, v    ! Temporary variables
   integer :: t               ! Output data

   call read_data(d1,d2,d3,d4,d5)
   call compute_u(d1,d2,u1)
   call compute_u(d3,d4,u2)
   call compute_w(d5,w)
   call compute_v(u2,w,v)
   call compute_t(u1,v,t)

   print * , 't=', t
end program
```

Alternative way with functions

```fortran
program exemple_2
   implicit none
   integer :: d1,d2,d3,d4,d5  ! Input data
   integer :: u1, u2, w, v, t ! Variables
   integer :: compute_u,compute_t,compute_w,compute_w

   call read_data(d1,d2,d3,d4,d5)
   u1 = compute_u(d1,d2)
   u2 = compute_u(d3,d4)
   w  = compute_w(d5)
   v  = compute_v(u2,w)
   t  = compute_t(u1,v)

   print * , 't=', t
end program
```

Single-line with functions

```fortran
program exemple_3
   implicit none
   integer :: d1,d2,d3,d4,d5  ! Input data
   integer :: u, v, w, t

   call read_data(d1,d2,d3,d4,d5)

   print * , 't=', &
     t( u(d1,d2), v( u(d3,d4), w(d5) ) )
end program
```

Now, the sequence of execution is handled by the compiler.

Same example with IRPF90

```
program exemple_4
   implicit none
   print * , 't=', t
end program
```

That's it!

- Using *t* triggers the exploration of the production tree
- Completely equivalent to the previous example, but the parameters of the function *t* are not expressed
- IRP : Implicit Reference to Parameters

Definition of the nodes of the tree

For each node, we write a **provider**. This is a subroutine whose role is to build the variable *and* guarantee that it is built properly.

file: *uvwt.irp.f*

```
BEGIN_PROVIDER [ integer, t ]
  t = u1+v+4
END_PROVIDER

BEGIN_PROVIDER [integer,w]
  w = d5+3
END_PROVIDER

BEGIN_PROVIDER [ integer, v ]
  v = u2+w+2
END_PROVIDER
```

```
BEGIN_PROVIDER [ integer, u1 ]
  u1 = d1+d2+1
END_PROVIDER

BEGIN_PROVIDER [ integer, u2 ]
  u2 = d3+d4+1
END_PROVIDER
```

file : *input.irp.f*

```
 BEGIN_PROVIDER [ integer, d1 ]
&BEGIN_PROVIDER [ integer, d2 ]
&BEGIN_PROVIDER [ integer, d3 ]
&BEGIN_PROVIDER [ integer, d4 ]
&BEGIN_PROVIDER [ integer, d5 ]
 read(*,*) d1
 read(*,*) d2
 read(*,*) d3
 read(*,*) d4
 read(*,*) d5
END_PROVIDER
```

When you write a provider for *x*, you **only** have to focus on

- How do I build *x*?
- What are the variables that I need to build *x*?
- Am I sure that *x* is built correctly when I exit the provider?

Using this method:

- You don't have to know the execution sequence
- If you need a variable (node), you are *sure* that it has been built properly when you use it
- You will never break other parts of the program
- Many people can work simultaneously on the same program with minimal effort
- If a node has already been built, it will not be built again. The correct value will be returned by the provider.

Fortran code generation

- Run *irpf90* in the current directory
- *irpf90* reads all the **.irp.f* files
- All the providers are identified
- All the corresponding variables (IRP entities) are searched for in the code
- The dependence tree is built
- Providers are transformed to subroutines (*subroutine provide_**)
- Calls to *provide_** are inserted in the code
- Each file **.irp.f* generates a module containing the IRP entities, and a Fortran file containing the subroutines/functions
- As the dependence tree is built, the dependences between the files are known and the *makefile* is built automatically

Generated code example

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

program irp_program                                ! irp_example1:   0
 call irp_example1                                 ! irp_example1.irp.f:   0
 call irp_finalize_742559343()                     ! irp_example1.irp.f:   0
end program                                        ! irp_example1.irp.f:   0
subroutine irp_example1                            ! irp_example1.irp.f:   1
  use uvwt_mod
  implicit none                                    ! irp_example1.irp.f:   2
  character*(12) :: irp_here = 'irp_example1'      ! irp_example1.irp.f:   1
  if (.not.t_is_built) then
    call provide_t
  endif
  print *, 't = ', t                                   ! irp_example1.irp.f:   3
end                                                    ! irp_example1.irp.f:   4
```

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

module uvwt_mod
  integer :: u1
  logical :: u1_is_built = .False.
  integer :: u2
  logical :: u2_is_built = .False.
  integer :: t
  logical :: t_is_built = .False.
  integer :: w
  logical :: w_is_built = .False.
  integer :: v
  logical :: v_is_built = .False.
end module uvwt_mod
```

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

subroutine provide_u1
  use uvwt_mod
```

```
   use input_mod
   implicit none
   character*(10) :: irp_here = 'provide_u1'
   integer                         :: irp_err
   logical                         :: irp_dimensions_OK
   if (.not.d1_is_built) then
     call provide_d1
   endif
  if (.not.u1_is_built) then
   call bld_u1
   u1_is_built = .True.

  endif
end subroutine provide_u1

subroutine bld_u1
   use uvwt_mod
   use input_mod
  character*(2) :: irp_here = 'u1'                    ! uvwt.irp.f:  13
  u1 = d1+d2+1                                        ! uvwt.irp.f:  14
end subroutine bld_u1
subroutine provide_u2
  use uvwt_mod
  use input_mod
  implicit none
  character*(10) :: irp_here = 'provide_u2'
  integer                    :: irp_err
  logical                    :: irp_dimensions_OK
  if (.not.d1_is_built) then
    call provide_d1
  endif
 if (.not.u2_is_built) then
  call bld_u2
  u2_is_built = .True.

 endif
```

```fortran
end subroutine provide_u2

subroutine bld_u2
  use uvwt_mod
  use input_mod
  character*(2) :: irp_here = 'u2'                    ! uvwt.irp.f:  17
  u2 = d3+d4+1                                        ! uvwt.irp.f:  18
end subroutine bld_u2
subroutine provide_t
  use uvwt_mod
  implicit none
  character*(9) :: irp_here = 'provide_t'
  integer                   :: irp_err
  logical                   :: irp_dimensions_OK
  if (.not.u1_is_built) then
    call provide_u1
  endif
  if (.not.v_is_built) then
```

```
    call provide_v
  endif
 if (.not.t_is_built) then
  call bld_t
  t_is_built = .True.

 endif
end subroutine provide_t

subroutine bld_t
  use uvwt_mod
  character*(1) :: irp_here = 't'                  ! uvwt.irp.f:   1
  t = u1+v+4                                       ! uvwt.irp.f:   2
end subroutine bld_t
subroutine provide_w
  use uvwt_mod
  use input_mod
  implicit none
  character*(9) :: irp_here = 'provide_w'
  integer                     :: irp_err
  logical                     :: irp_dimensions_OK
  if (.not.d1_is_built) then
    call provide_d1
  endif
 if (.not.w_is_built) then
  call bld_w
  w_is_built = .True.

 endif
end subroutine provide_w

subroutine bld_w
  use uvwt_mod
  use input_mod
  character*(1) :: irp_here = 'w'              ! uvwt.irp.f:   5
  w = d5+3                                     ! uvwt.irp.f:   6
```

```
end subroutine bld_w
subroutine provide_v
  use uvwt_mod
  implicit none
  character*(9) :: irp_here = 'provide_v'
  integer                      :: irp_err
  logical                      :: irp_dimensions_OK
  if (.not.w_is_built) then
    call provide_w
  endif
  if (.not.u2_is_built) then
    call provide_u2
  endif
 if (.not.v_is_built) then
  call bld_v
  v_is_built = .True.

 endif
```

```
end subroutine provide_v

subroutine bld_v
  use uvwt_mod
  character*(1) :: irp_here = 'v'                 ! uvwt.irp.f:   9
  v = u2+w+2                                      ! uvwt.irp.f:  10
end subroutine bld_v
```

Code execution with debug mode on:

```
$ ./irp_example1
           0 : -> provide_t
           0 :  -> provide_u1
           0 :   -> provide_d1
           0 :    -> d1
1
2
3
4
```

```
5
           0 :     <- d1   0.000000000000000
           0 :    <- provide_d1   0.000000000000000
           0 :    -> u1
           0 :    <- u1   0.000000000000000
           0 :   <- provide_u1   0.000000000000000
           0 :   -> provide_v
           0 :    -> provide_w
           0 :     -> w
           0 :     <- w   0.000000000000000
           0 :    <- provide_w   0.000000000000000
           0 :    -> provide_u2
           0 :     -> u2
           0 :     <- u2   0.000000000000000
           0 :    <- provide_u2   0.000000000000000
           0 :    -> v
           0 :    <- v   0.000000000000000
           0 :   <- provide_v   0.000000000000000
           0 :  -> t
           0 :  <- t   0.000000000000000000
           0 : <- provide_t   0.000000000000000000
           0 : -> irp_example1
t =            26
           0 : <- irp_example1   0.000000000000000000
```

Using subroutines/functions

```
BEGIN_PROVIDER [ integer, u1 ]
  integer :: fu
  u1 = fu(d1,d2)
END_PROVIDER

BEGIN_PROVIDER [ integer, u2 ]
  integer :: fu
  u2 = fu(d3,d4)
END_PROVIDER

integer function fu(x,y)
  integer :: x,y
  fu = x+y+1
end function
```

Providing arrays

An array is considered built when all its elements are built. Its dimensions can be provided variables, constants and intervals (a:b).

```
BEGIN_PROVIDER [ integer, fact_max ]
 fact_max = 10
END_PROVIDER

BEGIN_PROVIDER [ double precision, fact, (0:fact_max) ]
  integer :: i

  fact(0) = 1.d0
  do i=1,fact_max
    fact(i) = fact(i-1)*dble(i)
  enddo
END_PROVIDER
```

```fortran
program test
  print *,  fact(5)
end
```

```
$ ./test
           0 : -> provide_fact
           0 :  -> provide_fact_max
           0 :   -> fact_max
           0 :   <- fact_max   0.000000000000000000
           0 :  <- provide_fact_max   0.000000000000000000
           0 :  -> fact
           0 :  <- fact   0.000000000000000000
           0 : <- provide_fact   0.000000000000000000
           0 : -> test
   120.000000000000000
           0 : <- test   0.000000000000000000
```

The allocation behaves as follows:

- If the array is not already allocated, it is allocated

- If the array already allocated, check if the dimensions have changed
- If the dimensions have not changed, then OK.
- Else deallocate the array and re-allocate it with the correct dimensions
- All allocations/deallocations are checked with *stat=err*

```fortran
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

subroutine provide_fact_max
  use fact_mod
  implicit none
  character*(16) :: irp_here = 'provide_fact_max'
  integer                    :: irp_err
  logical                          :: irp_dimensions_OK
 if (.not.fact_max_is_built) then
  call bld_fact_max
  fact_max_is_built = .True.

 endif
end subroutine provide_fact_max

subroutine bld_fact_max
  use fact_mod
  character*(8) :: irp_here = 'fact_max'     ! fact.irp.f:   1
 fact_max = 10                               ! fact.irp.f:   2
end subroutine bld_fact_max
subroutine provide_fact
  use fact_mod
  implicit none
  character*(12) :: irp_here = 'provide_fact'
  integer                          :: irp_err
  logical                         :: irp_dimensions_OK
  if (.not.fact_max_is_built) then
    call provide_fact_max
  endif
 if (allocated (fact) ) then
  irp_dimensions_OK = .True.
  irp_dimensions_OK = irp_dimensions_OK.AND. &
    (SIZE(fact,1)==(fact_max - (-1)))
  if (.not.irp_dimensions_OK) then
   deallocate(fact,stat=irp_err)
   if (irp_err /= 0) then
     print *, irp_here//': Deallocation failed: fact'
     print *, ' size: (0:fact_max)'
   endif
   if ((fact_max - (-1)>0)) then
    allocate(fact(0:fact_max),stat=irp_err)
    if (irp_err /= 0) then
     print *, irp_here//': Allocation failed: fact'
     print *, ' size: (0:fact_max)'
    endif
   endif
  endif
 else
   if ((fact_max - (-1)>0)) then
    allocate(fact(0:fact_max),stat=irp_err)
    if (irp_err /= 0) then
     print *, irp_here//': Allocation failed: fact'
     print *, ' size: (0:fact_max)'
    endif
   endif
 endif
 if (.not.fact_is_built) then
  call bld_fact
  fact_is_built = .True.

 endif
```

```
end subroutine provide_fact

subroutine bld_fact
  use fact_mod
  character*(4) :: irp_here = 'fact'                ! fact.irp.f:    5
  integer :: i                                      ! fact.irp.f:    6
  fact(0) = 1.d0                                    ! fact.irp.f:    8
  do i=1,fact_max                                   ! fact.irp.f:    9
    fact(i) = fact(i-1)*dble(i)                     ! fact.irp.f:   10
  enddo                                             ! fact.irp.f:   11
end subroutine bld_fact
```

Modifying a variable outside of its provider

In iterative processes, a variable needs to be modified outside of its provider. If it is the case, IRPF90 has to be informed of the change by the **TOUCH** keyword.

Example: computing numerical derivatives

```
BEGIN_PROVIDER [ real, dPsi ]
  x += 0.5*delta_x
  TOUCH x
  dPsi = Psi
  x -= delta_x
  TOUCH x
  dPsi = (dPsi - Psi)/delta_x
  x += 0.5*delta_x
  SOFT_TOUCH x
END_PROVIDER
```

Generated code:

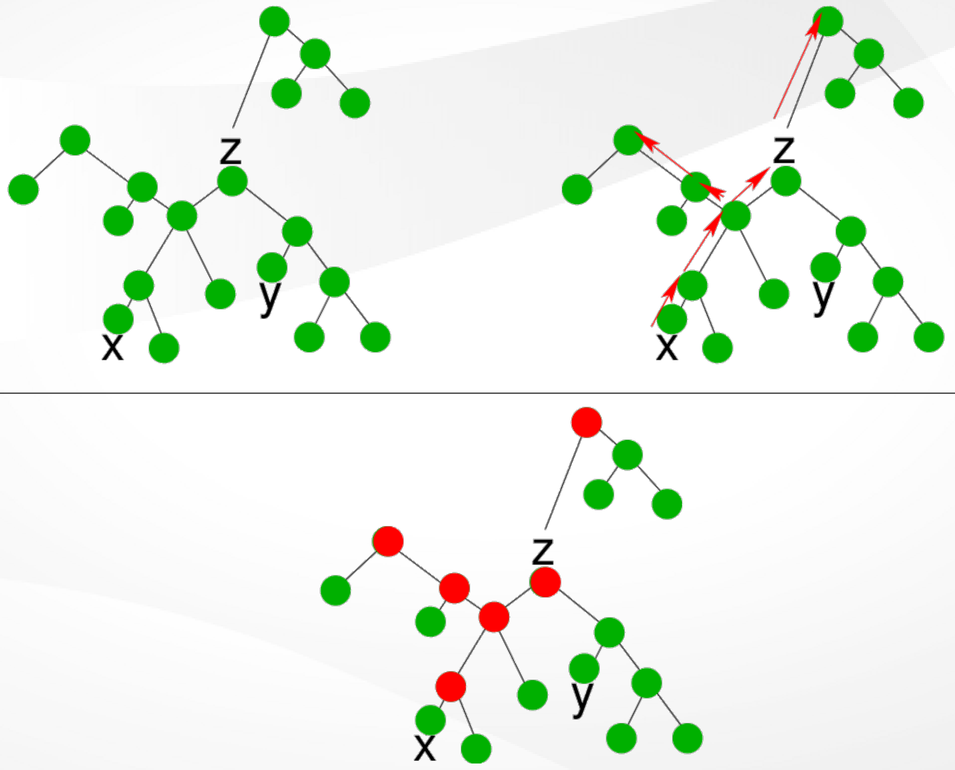
```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

subroutine provide_dpsi
  use y_mod
  use x_mod
  implicit none
  character*(12) :: irp_here = 'provide_dpsi'
  integer                    :: irp_err
  logical                    :: irp_dimensions_OK
  if (.not.x_is_built) then
    call provide_x
```

```
   endif
   if (.not.psi_is_built) then
     call provide_psi
   endif
   if (.not.delta_x_is_built) then
     call provide_delta_x
   endif
  if (.not.dpsi_is_built) then
   call bld_dpsi
   dpsi_is_built = .True.

  endif
end subroutine provide_dpsi

subroutine bld_dpsi
   use y_mod
   use x_mod
  use y_mod                                              ! x.irp.f:    3
```

```
 use y_mod                                          ! x.irp.f:   6
 use y_mod                                          ! x.irp.f:   9
  character*(4) :: irp_here = 'dpsi'                ! x.irp.f:   1
  x =x +( 0.5*delta_x)                              ! x.irp.f:   2
!                                                   ! x.irp.f:   3
! >>> TOUCH x                                       ! x.irp.f:   3
 call touch_x                                       ! x.irp.f:   3
! <<< END TOUCH                                     ! x.irp.f:   3
  if (.not.x_is_built) then
    call provide_x
  endif
  if (.not.psi_is_built) then
    call provide_psi
  endif
  if (.not.delta_x_is_built) then
    call provide_delta_x
  endif
  dPsi = Psi                                        ! x.irp.f:   4
  x =x -( delta_x)                                 ! x.irp.f:   5
!                                                  ! x.irp.f:   6
! >>> TOUCH x                                      ! x.irp.f:   6
 call touch_x                                      ! x.irp.f:   6
! <<< END TOUCH                                    ! x.irp.f:   6
  if (.not.x_is_built) then
    call provide_x
  endif
  if (.not.psi_is_built) then
    call provide_psi
  endif
  if (.not.delta_x_is_built) then
    call provide_delta_x
  endif
  dPsi = (dPsi - Psi)/delta_x                      ! x.irp.f:   7
  x =x +( 0.5*delta_x)                             ! x.irp.f:   8
!                                                  ! x.irp.f:   9
! >>> TOUCH x                                      ! x.irp.f:   9
```

```
 call touch_x                                          ! x.irp.f:    9
! <<< END TOUCH (Soft)                                 ! x.irp.f:    9
end subroutine bld_dpsi
```

How this works:



Templates

When pieces of code are very similar, it is possible to use a template:

```
BEGIN_TEMPLATE

 subroutine insertion_$Xsort (x,iorder,isize)
  implicit none
  $type,intent(inout)    :: x(isize)
  integer,intent(inout)  :: iorder(isize)
  integer,intent(in)     :: isize
  $type                  :: xtmp
  integer                :: i, i0, j, jmax

  do i=1,isize
   xtmp = x(i)
   i0 = iorder(i)
   j = i-1
   do j=i-1,1,-1
    if ( x(j) > xtmp ) then
     x(j+1) = x(j)
     iorder(j+1) = iorder(j)
    else
     exit
    endif
   enddo
   x(j+1) = xtmp
   iorder(j+1) = i0
  enddo

 end

SUBST [ X, type ]

   ; real ;;
 d ; double precision ;;
 i ; integer ;;

END_TEMPLATE
```

Generated code:

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                            !
!           DO NOT MODIFY IT BY HAND  !
!-----------------------------------------------!

 subroutine insertion_sort (x,iorder,isize)   !x.irp.f_tpl_35:  3
  implicit none                               !x.irp.f_tpl_35:  4
  character*(14) :: irp_here='insertion_sort' !x.irp.f_tpl_35:  3
  real,intent(inout)    :: x(isize)           !x.irp.f_tpl_35:  5
  integer,intent(inout)  :: iorder(isize)     !x.irp.f_tpl_35:  6
  integer,intent(in)     :: isize                 !x.irp.f_tpl_35:  7
  real                  :: xtmp                  !x.irp.f_tpl_35:  8
  integer                :: i, i0, j, jmax       !x.irp.f_tpl_35:  9
  do i=1,isize                                   !x.irp.f_tpl_35: 11
   xtmp = x(i)                                   !x.irp.f_tpl_35: 12
   i0 = iorder(i)                                !x.irp.f_tpl_35: 13
   j = i-1                                       !x.irp.f_tpl_35: 14
   do j=i-1,1,-1                                 !x.irp.f_tpl_35: 15
    if ( x(j) > xtmp ) then                      !x.irp.f_tpl_35: 16
     x(j+1) = x(j)                               !x.irp.f_tpl_35: 17
     iorder(j+1) = iorder(j)                     !x.irp.f_tpl_35: 18
    else                                         !x.irp.f_tpl_35: 19
     exit                                        !x.irp.f_tpl_35: 20
    endif                                        !x.irp.f_tpl_35: 21
   enddo                                         !x.irp.f_tpl_35: 22
   x(j+1) = xtmp                                 !x.irp.f_tpl_35: 23
   iorder(j+1) = i0                              !x.irp.f_tpl_35: 24
  enddo                                          !x.irp.f_tpl_35: 25
```

```
end                                          !x.irp.f_tpl_35: 27
subroutine insertion_dsort (x,iorder,isize)  !x.irp.f_tpl_35: 32
 implicit none                               !x.irp.f_tpl_35: 33
 character*(15) :: irp_here='insertion_dsort'!x.irp.f_tpl_35: 32
 double precision,intent(inout)   :: x(isize)!x.irp.f_tpl_35: 34
 integer,intent(inout)  :: iorder(isize)     !x.irp.f_tpl_35: 35
 integer,intent(in)     :: isize             !x.irp.f_tpl_35: 36
 double precision                   :: xtmp  !x.irp.f_tpl_35: 37
 integer                :: i, i0, j, jmax    !x.irp.f_tpl_35: 38
 do i=1,isize                                !x.irp.f_tpl_35: 40
  xtmp = x(i)                                !x.irp.f_tpl_35: 41
  i0 = iorder(i)                             !x.irp.f_tpl_35: 42
  j = i-1                                    !x.irp.f_tpl_35: 43
  do j=i-1,1,-1                              !x.irp.f_tpl_35: 44
   if ( x(j) > xtmp ) then                   !x.irp.f_tpl_35: 45
    x(j+1) = x(j)                            !x.irp.f_tpl_35: 46
    iorder(j+1) = iorder(j)                  !x.irp.f_tpl_35: 47
   else                                      !x.irp.f_tpl_35: 48
```

```
     exit                                         !x.irp.f_tpl_35: 49
    endif                                         !x.irp.f_tpl_35: 50
   enddo                                          !x.irp.f_tpl_35: 51
   x(j+1) = xtmp                                  !x.irp.f_tpl_35: 52
   iorder(j+1) = i0                               !x.irp.f_tpl_35: 53
  enddo                                           !x.irp.f_tpl_35: 54
 end                                              !x.irp.f_tpl_35: 56
 subroutine insertion_isort (x,iorder,isize)      !x.irp.f_tpl_35: 61
  implicit none                                   !x.irp.f_tpl_35: 62
  character*(15) :: irp_here='insertion_isort'!x.irp.f_tpl_35: 61
  integer,intent(inout)    :: x(isize)            !x.irp.f_tpl_35: 63
  integer,intent(inout)  :: iorder(isize)         !x.irp.f_tpl_35: 64
  integer,intent(in)     :: isize                 !x.irp.f_tpl_35: 65
  integer                  :: xtmp                !x.irp.f_tpl_35: 66
  integer                :: i, i0, j, jmax        !x.irp.f_tpl_35: 67
  do i=1,isize                                    !x.irp.f_tpl_35: 69
   xtmp = x(i)                                    !x.irp.f_tpl_35: 70
   i0 = iorder(i)                                 !x.irp.f_tpl_35: 71
   j = i-1                                         !x.irp.f_tpl_35: 72
   do j=i-1,1,-1                                   !x.irp.f_tpl_35: 73
    if ( x(j) > xtmp ) then                        !x.irp.f_tpl_35: 74
     x(j+1) = x(j)                                 !x.irp.f_tpl_35: 75
     iorder(j+1) = iorder(j)                       !x.irp.f_tpl_35: 76
    else                                           !x.irp.f_tpl_35: 77
     exit                                          !x.irp.f_tpl_35: 78
    endif                                          !x.irp.f_tpl_35: 79
   enddo                                           !x.irp.f_tpl_35: 80
   x(j+1) = xtmp                                   !x.irp.f_tpl_35: 81
   iorder(j+1) = i0                                !x.irp.f_tpl_35: 82
  enddo                                            !x.irp.f_tpl_35: 83
end                                                !x.irp.f_tpl_35: 85
```

Metaprogramming

Shell scripts can be inserted in the IRPF90 code, and the output of the script will be inserted in the generated Fortran. For example:

```
program test
 BEGIN_SHELL [ /bin/bash ]
  echo print *, \'Compiled by $(whoami) on $(date)\'
 END_SHELL
end
```

Generated code:

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
```

```
!-----------------------------------------------!

program irp_program                                                           ! test:    0
 call test                                                                    ! test.irp.f:    0
 call irp_finalize_491024427()                                                ! test.irp.f:    0
end program                                                                   ! test.irp.f:    0
subroutine test                                                               ! test.irp.f:    1
  character*(4) :: irp_here = 'test'                                          ! test.irp.f:    1
print *, 'Compiled by scemama on Mon Jul 8 11:28:16 CEST 2013'                ! test.irp.f_shell_4:    1
end                                                                           ! test.irp.f:    5
```

Example: Computing powers of x

```
BEGIN_SHELL [ /usr/bin/python ]

POWER_MAX = 20

def compute_x_prod(n,d):
  if n == 0:
    d[0] = None
    return d
  if n == 1:
    d[1] = None
    return d
  if n in d:
    return d
  m = n/2
  d = compute_x_prod(m,d)
  d[n] = None
  d[2*m] = None
  return d

def print_subroutine(n):
  keys = compute_x_prod(n,{}).keys()
  keys.sort()
  output = []
  print "real function power_%d(x1)"%n
  print " real, intent(in) :: x1"
  for i in range(1,len(keys)):
     output.append( "x%d"%keys[i] )
  if output != []:
    print " real :: "+', '.join(output)
  for i in range(1,len(keys)):
   ki = keys[i]
   ki1 = keys[i-1]
   if ki == 2*ki1:
     print " x%d"%ki + " = x%d * x%d"%(ki1,ki1)
   else:
     print " x%d"%ki + " = x%d * x1"%(ki1)
  print " power_%d = x%d"%(n,n)
  print "end"

for i in range(POWER_MAX):
  print_subroutine (i+1)
  print ''

END_SHELL
```

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

real function power_1(x1)                  ! power.irp.f_shell_44:   1
  character*(7) :: irp_here = 'power_1'    ! power.irp.f_shell_44:   1
 real, intent(in) :: x1                    ! power.irp.f_shell_44:   2
 power_1 = x1                              ! power.irp.f_shell_44:   3
end                                        ! power.irp.f_shell_44:   4
real function power_2(x1)                  ! power.irp.f_shell_44:   6
  character*(7) :: irp_here = 'power_2'    ! power.irp.f_shell_44:   6
 real, intent(in) :: x1                    ! power.irp.f_shell_44:   7
 real :: x2                                ! power.irp.f_shell_44:   8
 x2 = x1 * x1                              ! power.irp.f_shell_44:   9
 power_2 = x2                                ! power.irp.f_shell_44:  10
end                                          ! power.irp.f_shell_44:  11
real function power_3(x1)                    ! power.irp.f_shell_44:  13
  character*(7) :: irp_here = 'power_3'      ! power.irp.f_shell_44:  13
 real, intent(in) :: x1                      ! power.irp.f_shell_44:  14
 real :: x2, x3                              ! power.irp.f_shell_44:  15
 x2 = x1 * x1                                ! power.irp.f_shell_44:  16
 x3 = x2 * x1                                ! power.irp.f_shell_44:  17
 power_3 = x3                                ! power.irp.f_shell_44:  18
end                                          ! power.irp.f_shell_44:  19
real function power_4(x1)                    ! power.irp.f_shell_44:  21
  character*(7) :: irp_here = 'power_4'      ! power.irp.f_shell_44:  21
 real, intent(in) :: x1                      ! power.irp.f_shell_44:  22
 real :: x2, x4                              ! power.irp.f_shell_44:  23
 x2 = x1 * x1                                ! power.irp.f_shell_44:  24
 x4 = x2 * x2                                ! power.irp.f_shell_44:  25
 power_4 = x4                                ! power.irp.f_shell_44:  26
end                                          ! power.irp.f_shell_44:  27
```

```
real function power_5(x1)                        ! power.irp.f_shell_44:  29
  character*(7) :: irp_here = 'power_5'          ! power.irp.f_shell_44:  29
 real, intent(in) :: x1                          ! power.irp.f_shell_44:  30
 real :: x2, x4, x5                              ! power.irp.f_shell_44:  31
 x2 = x1 * x1                                    ! power.irp.f_shell_44:  32
 x4 = x2 * x2                                    ! power.irp.f_shell_44:  33
 x5 = x4 * x1                                    ! power.irp.f_shell_44:  34
 power_5 = x5                                    ! power.irp.f_shell_44:  35
end                                              ! power.irp.f_shell_44:  36
real function power_6(x1)                        ! power.irp.f_shell_44:  38
  character*(7) :: irp_here = 'power_6'          ! power.irp.f_shell_44:  38
 real, intent(in) :: x1                          ! power.irp.f_shell_44:  39
 real :: x2, x3, x6                              ! power.irp.f_shell_44:  40
 x2 = x1 * x1                                    ! power.irp.f_shell_44:  41
 x3 = x2 * x1                                    ! power.irp.f_shell_44:  42
 x6 = x3 * x3                                    ! power.irp.f_shell_44:  43
 power_6 = x6                                    ! power.irp.f_shell_44:  44
end                                              ! power.irp.f_shell_44:  45
```

```fortran
real function power_7(x1)                      ! power.irp.f_shell_44:  47
  character*(7) :: irp_here = 'power_7'         ! power.irp.f_shell_44:  47
 real, intent(in) :: x1                         ! power.irp.f_shell_44:  48
 real :: x2, x3, x6, x7                         ! power.irp.f_shell_44:  49
 x2 = x1 * x1                                   ! power.irp.f_shell_44:  50
 x3 = x2 * x1                                   ! power.irp.f_shell_44:  51
 x6 = x3 * x3                                   ! power.irp.f_shell_44:  52
 x7 = x6 * x1                                   ! power.irp.f_shell_44:  53
 power_7 = x7                                   ! power.irp.f_shell_44:  54
end                                             ! power.irp.f_shell_44:  55
real function power_8(x1)                      ! power.irp.f_shell_44:  57
  character*(7) :: irp_here = 'power_8'         ! power.irp.f_shell_44:  57
 real, intent(in) :: x1                         ! power.irp.f_shell_44:  58
 real :: x2, x4, x8                             ! power.irp.f_shell_44:  59
 x2 = x1 * x1                                   ! power.irp.f_shell_44:  60
 x4 = x2 * x2                                   ! power.irp.f_shell_44:  61
 x8 = x4 * x4                                   ! power.irp.f_shell_44:  62
 power_8 = x8                                   ! power.irp.f_shell_44:  63
```

```
end                                        ! power.irp.f_shell_44:  64
real function power_9(x1)                  ! power.irp.f_shell_44:  66
  character*(7) :: irp_here = 'power_9'    ! power.irp.f_shell_44:  66
 real, intent(in) :: x1                    ! power.irp.f_shell_44:  67
 real :: x2, x4, x8, x9                    ! power.irp.f_shell_44:  68
 x2 = x1 * x1                              ! power.irp.f_shell_44:  69
 x4 = x2 * x2                              ! power.irp.f_shell_44:  70
 x8 = x4 * x4                              ! power.irp.f_shell_44:  71
 x9 = x8 * x1                              ! power.irp.f_shell_44:  72
 power_9 = x9                              ! power.irp.f_shell_44:  73
end                                        ! power.irp.f_shell_44:  74
real function power_10(x1)                 ! power.irp.f_shell_44:  76
  character*(8) :: irp_here = 'power_10'   ! power.irp.f_shell_44:  76
 real, intent(in) :: x1                    ! power.irp.f_shell_44:  77
 real :: x2, x4, x5, x10                   ! power.irp.f_shell_44:  78
 x2 = x1 * x1                              ! power.irp.f_shell_44:  79
 x4 = x2 * x2                              ! power.irp.f_shell_44:  80
 x5 = x4 * x1                              ! power.irp.f_shell_44:  81
 x10 = x5 * x5                                ! power.irp.f_shell_44:  82
 power_10 = x10                               ! power.irp.f_shell_44:  83
end                                           ! power.irp.f_shell_44:  84
real function power_11(x1)                    ! power.irp.f_shell_44:  86
  character*(8) :: irp_here = 'power_11'      ! power.irp.f_shell_44:  86
 real, intent(in) :: x1                       ! power.irp.f_shell_44:  87
 real :: x2, x4, x5, x10, x11                 ! power.irp.f_shell_44:  88
 x2 = x1 * x1                                 ! power.irp.f_shell_44:  89
 x4 = x2 * x2                                 ! power.irp.f_shell_44:  90
 x5 = x4 * x1                                 ! power.irp.f_shell_44:  91
 x10 = x5 * x5                                ! power.irp.f_shell_44:  92
 x11 = x10 * x1                               ! power.irp.f_shell_44:  93
 power_11 = x11                               ! power.irp.f_shell_44:  94
end                                           ! power.irp.f_shell_44:  95
real function power_12(x1)                    ! power.irp.f_shell_44:  97
  character*(8) :: irp_here = 'power_12'      ! power.irp.f_shell_44:  97
 real, intent(in) :: x1                       ! power.irp.f_shell_44:  98
 real :: x2, x3, x6, x12                      ! power.irp.f_shell_44:  99
 x2 = x1 * x1                                ! power.irp.f_shell_44: 100
 x3 = x2 * x1                                ! power.irp.f_shell_44: 101
 x6 = x3 * x3                                ! power.irp.f_shell_44: 102
 x12 = x6 * x6                               ! power.irp.f_shell_44: 103
 power_12 = x12                              ! power.irp.f_shell_44: 104
end                                          ! power.irp.f_shell_44: 105
real function power_13(x1)                   ! power.irp.f_shell_44: 107
  character*(8) :: irp_here = 'power_13'     ! power.irp.f_shell_44: 107
 real, intent(in) :: x1                      ! power.irp.f_shell_44: 108
 real :: x2, x3, x6, x12, x13                ! power.irp.f_shell_44: 109
 x2 = x1 * x1                                ! power.irp.f_shell_44: 110
 x3 = x2 * x1                                ! power.irp.f_shell_44: 111
 x6 = x3 * x3                                ! power.irp.f_shell_44: 112
 x12 = x6 * x6                               ! power.irp.f_shell_44: 113
 x13 = x12 * x1                              ! power.irp.f_shell_44: 114
 power_13 = x13                              ! power.irp.f_shell_44: 115
end                                          ! power.irp.f_shell_44: 116
real function power_14(x1)                   ! power.irp.f_shell_44: 118
```

```
  character*(8) :: irp_here = 'power_14' ! power.irp.f_shell_44: 118
 real, intent(in) :: x1                   ! power.irp.f_shell_44: 119
 real :: x2, x3, x6, x7, x14              ! power.irp.f_shell_44: 120
 x2 = x1 * x1                             ! power.irp.f_shell_44: 121
 x3 = x2 * x1                             ! power.irp.f_shell_44: 122
 x6 = x3 * x3                             ! power.irp.f_shell_44: 123
 x7 = x6 * x1                             ! power.irp.f_shell_44: 124
 x14 = x7 * x7                            ! power.irp.f_shell_44: 125
 power_14 = x14                           ! power.irp.f_shell_44: 126
end                                       ! power.irp.f_shell_44: 127
real function power_15(x1)                ! power.irp.f_shell_44: 129
  character*(8) :: irp_here = 'power_15' ! power.irp.f_shell_44: 129
 real, intent(in) :: x1                   ! power.irp.f_shell_44: 130
 real :: x2, x3, x6, x7, x14, x15         ! power.irp.f_shell_44: 131
 x2 = x1 * x1                             ! power.irp.f_shell_44: 132
 x3 = x2 * x1                             ! power.irp.f_shell_44: 133
 x6 = x3 * x3                             ! power.irp.f_shell_44: 134
 x7 = x6 * x1                             ! power.irp.f_shell_44: 135
 x14 = x7 * x7                              ! power.irp.f_shell_44: 136
 x15 = x14 * x1                             ! power.irp.f_shell_44: 137
 power_15 = x15                             ! power.irp.f_shell_44: 138
end                                         ! power.irp.f_shell_44: 139
real function power_16(x1)                  ! power.irp.f_shell_44: 141
  character*(8) :: irp_here = 'power_16'    ! power.irp.f_shell_44: 141
 real, intent(in) :: x1                     ! power.irp.f_shell_44: 142
 real :: x2, x4, x8, x16                    ! power.irp.f_shell_44: 143
 x2 = x1 * x1                               ! power.irp.f_shell_44: 144
 x4 = x2 * x2                               ! power.irp.f_shell_44: 145
 x8 = x4 * x4                               ! power.irp.f_shell_44: 146
 x16 = x8 * x8                              ! power.irp.f_shell_44: 147
 power_16 = x16                             ! power.irp.f_shell_44: 148
end                                         ! power.irp.f_shell_44: 149
real function power_17(x1)                  ! power.irp.f_shell_44: 151
  character*(8) :: irp_here = 'power_17'    ! power.irp.f_shell_44: 151
 real, intent(in) :: x1                     ! power.irp.f_shell_44: 152
 real :: x2, x4, x8, x16, x17               ! power.irp.f_shell_44: 153
```

```
  x2 = x1 * x1                                 ! power.irp.f_shell_44: 154
  x4 = x2 * x2                                 ! power.irp.f_shell_44: 155
  x8 = x4 * x4                                 ! power.irp.f_shell_44: 156
  x16 = x8 * x8                                ! power.irp.f_shell_44: 157
  x17 = x16 * x1                               ! power.irp.f_shell_44: 158
  power_17 = x17                               ! power.irp.f_shell_44: 159
 end                                           ! power.irp.f_shell_44: 160
 real function power_18(x1)                    ! power.irp.f_shell_44: 162
   character*(8) :: irp_here = 'power_18'      ! power.irp.f_shell_44: 162
  real, intent(in) :: x1                       ! power.irp.f_shell_44: 163
  real :: x2, x4, x8, x9, x18                  ! power.irp.f_shell_44: 164
  x2 = x1 * x1                                 ! power.irp.f_shell_44: 165
  x4 = x2 * x2                                 ! power.irp.f_shell_44: 166
  x8 = x4 * x4                                 ! power.irp.f_shell_44: 167
  x9 = x8 * x1                                 ! power.irp.f_shell_44: 168
  x18 = x9 * x9                                ! power.irp.f_shell_44: 169
  power_18 = x18                               ! power.irp.f_shell_44: 170
 end                                           ! power.irp.f_shell_44: 171
```

```
real function power_19(x1)                       ! power.irp.f_shell_44: 173
  character*(8) :: irp_here = 'power_19'         ! power.irp.f_shell_44: 173
 real, intent(in) :: x1                          ! power.irp.f_shell_44: 174
 real :: x2, x4, x8, x9, x18, x19                ! power.irp.f_shell_44: 175
 x2 = x1 * x1                                    ! power.irp.f_shell_44: 176
 x4 = x2 * x2                                    ! power.irp.f_shell_44: 177
 x8 = x4 * x4                                    ! power.irp.f_shell_44: 178
 x9 = x8 * x1                                    ! power.irp.f_shell_44: 179
 x18 = x9 * x9                                   ! power.irp.f_shell_44: 180
 x19 = x18 * x1                                  ! power.irp.f_shell_44: 181
 power_19 = x19                                  ! power.irp.f_shell_44: 182
end                                              ! power.irp.f_shell_44: 183
real function power_20(x1)                       ! power.irp.f_shell_44: 185
  character*(8) :: irp_here = 'power_20'         ! power.irp.f_shell_44: 185
 real, intent(in) :: x1                          ! power.irp.f_shell_44: 186
 real :: x2, x4, x5, x10, x20                    ! power.irp.f_shell_44: 187
 x2 = x1 * x1                                    ! power.irp.f_shell_44: 188
 x4 = x2 * x2                                    ! power.irp.f_shell_44: 189
```

```
  x5 = x4 * x1                                         ! power.irp.f_shell_44: 190
  x10 = x5 * x5                                        ! power.irp.f_shell_44: 191
  x20 = x10 * x10                                      ! power.irp.f_shell_44: 192
  power_20 = x20                                       ! power.irp.f_shell_44: 193
end                                                    ! power.irp.f_shell_44: 194
```

IRPF90 for HPC

Using the *--align* option, IRPF90 can introduce compiler directives for the Intel Fortran compiler, such that *all* the arrays are aligned. The *$IRP_ALIGN* variable corresponds to this alignment.

For example,

```
integer function align_double(i)
 integer, intent(in) :: i
 integer :: j
 j = mod(i,max($IRP_ALIGN,4)/4)
 if (j==0) then
   align_double = i
 else
   align_double = i+4-j
 endif
end
```

```
 BEGIN_PROVIDER [ integer, n ]
&BEGIN_PROVIDER [ integer, n_aligned ]
  integer :: align_double
  n = 19
  n_aligned = align_double(19)
END_PROVIDER

BEGIN_PROVIDER [ double precision, Matrix, (n_aligned,n) ]
 Matrix = 0.d0
END_PROVIDER
```

```
program test
  print *,  size(Matrix,1), size(Matrix,2)
end
```

Generated code without alignment:

```
!  -*- F90 -*-
!
```

```
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

module matrix_mod
  double precision, allocatable :: matrix (:,:)
  logical :: matrix_is_built = .False.
  integer :: n_aligned
  integer :: n
  logical :: n_is_built = .False.
end module matrix_mod
```

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                                 !
!           DO NOT MODIFY IT BY HAND              !
!-------------------------------------------------!

subroutine provide_matrix
  use matrix_mod
  implicit none
  character*(14) :: irp_here = 'provide_matrix'
  integer                      :: irp_err
  logical                      :: irp_dimensions_OK
  if (.not.n_is_built) then
    call provide_n
  endif
 if (allocated (matrix) ) then
  irp_dimensions_OK = .True.
  irp_dimensions_OK = irp_dimensions_OK.AND.(SIZE(matrix,1)==(n_aligned))
  irp_dimensions_OK = irp_dimensions_OK.AND.(SIZE(matrix,2)==(n))
  if (.not.irp_dimensions_OK) then
```

```
    deallocate(matrix,stat=irp_err)
    if (irp_err /= 0) then
```

```
     print *, irp_here//': Deallocation failed: matrix'
     print *, ' size: (n_aligned,n)'
   endif
   if ((n_aligned>0).and.(n>0)) then
    allocate(matrix(n_aligned,n),stat=irp_err)
    if (irp_err /= 0) then
     print *, irp_here//': Allocation failed: matrix'
     print *, ' size: (n_aligned,n)'
    endif
   endif
  endif
 else
   if ((n_aligned>0).and.(n>0)) then
    allocate(matrix(n_aligned,n),stat=irp_err)
    if (irp_err /= 0) then
     print *, irp_here//': Allocation failed: matrix'
     print *, ' size: (n_aligned,n)'
    endif
    endif
  endif
  if (.not.matrix_is_built) then
   call bld_matrix
   matrix_is_built = .True.

  endif
end subroutine provide_matrix

subroutine bld_matrix
   use matrix_mod
   character*(6) :: irp_here = 'matrix'                                          ! matrix.irp.f:  19
  Matrix = 0.d0                                                                  ! matrix.irp.f:  20
end subroutine bld_matrix
subroutine provide_n
   use matrix_mod
   implicit none
   character*(9) :: irp_here = 'provide_n'
```

```
   integer                          :: irp_err
   logical                          :: irp_dimensions_OK
  if (.not.n_is_built) then
   call bld_n
   n_is_built = .True.
```

```
 endif
end subroutine provide_n

subroutine bld_n
  use matrix_mod
  character*(1) :: irp_here = 'n'            ! matrix.irp.f:  12
  integer :: align_double                     ! matrix.irp.f:  14
  n = 19                                      ! matrix.irp.f:  15
  n_aligned = align_double(19)                ! matrix.irp.f:  16
end subroutine bld_n
integer function align_double(i)              ! matrix.irp.f:   1
  character*(12) :: irp_here = 'align_double'! matrix.irp.f:   1
 integer, intent(in) :: i                     ! matrix.irp.f:   2
 integer :: j                                 ! matrix.irp.f:   3
 j = mod(i,max(1,4)/4)                        ! matrix.irp.f:   4
 if (j==0) then                               ! matrix.irp.f:   5
   align_double = i                           ! matrix.irp.f:   6
 else                                                        ! matrix.irp.f:   7
   align_double = i+4-j                                      ! matrix.irp.f:   8
 endif                                                       ! matrix.irp.f:   9
end                                                           ! matrix.irp.f:  10
```

Output:

```
$ ./test
          19             19
```

Generated code with an alignment of 32 bytes:

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!
```

```fortran
module matrix_mod
  double precision, allocatable :: matrix (:,:)
  !DIR$ ATTRIBUTES ALIGN: 32 :: matrix
  logical :: matrix_is_built = .False.
  integer :: n_aligned
  integer :: n
  logical :: n_is_built = .False.
end module matrix_mod
```

```fortran
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

subroutine provide_matrix
```

```
  use matrix_mod
  implicit none
  character*(14) :: irp_here = 'provide_matrix'
  integer                    :: irp_err
  logical                    :: irp_dimensions_OK
  if (.not.n_is_built) then
    call provide_n
  endif
 if (allocated (matrix) ) then
  irp_dimensions_OK = .True.
  irp_dimensions_OK = irp_dimensions_OK.AND.(SIZE(matrix,1)==(n_aligned))
  irp_dimensions_OK = irp_dimensions_OK.AND.(SIZE(matrix,2)==(n))
  if (.not.irp_dimensions_OK) then
   deallocate(matrix,stat=irp_err)
   if (irp_err /= 0) then
     print *, irp_here//': Deallocation failed: matrix'
     print *, ' size: (n_aligned,n)'
   endif
```

```
   if ((n_aligned>0).and.(n>0)) then
    allocate(matrix(n_aligned,n),stat=irp_err)
    if (irp_err /= 0) then
     print *, irp_here//': Allocation failed: matrix'
     print *, ' size: (n_aligned,n)'
    endif
   endif
  endif
 else
   if ((n_aligned>0).and.(n>0)) then
    allocate(matrix(n_aligned,n),stat=irp_err)
    if (irp_err /= 0) then
     print *, irp_here//': Allocation failed: matrix'
     print *, ' size: (n_aligned,n)'
    endif
   endif
 endif
 if (.not.matrix_is_built) then
  call bld_matrix
  matrix_is_built = .True.
 endif
end subroutine provide_matrix

subroutine bld_matrix
  use matrix_mod
  character*(6) :: irp_here = 'matrix'                                    ! matrix.irp.f:  19
 Matrix = 0.d0                                                            ! matrix.irp.f:  20
end subroutine bld_matrix
subroutine provide_n
  use matrix_mod
  implicit none
  character*(9) :: irp_here = 'provide_n'
  integer                    :: irp_err
  logical                    :: irp_dimensions_OK
 if (.not.n_is_built) then
  call bld_n
  n_is_built = .True.
```

```
 endif
end subroutine provide_n

subroutine bld_n
```

```
  use matrix_mod
  character*(1) :: irp_here = 'n'            ! matrix.irp.f:  12
  integer :: align_double                    ! matrix.irp.f:  14
  n = 19                                     ! matrix.irp.f:  15
  n_aligned = align_double(19)               ! matrix.irp.f:  16
end subroutine bld_n
integer function align_double(i)             ! matrix.irp.f:   1
  character*(12) :: irp_here = 'align_double'! matrix.irp.f:   1
 integer, intent(in) :: i                    ! matrix.irp.f:   2
 integer :: j                                ! matrix.irp.f:   3
 j = mod(i,max(32,4)/4)                      ! matrix.irp.f:   4
 if (j==0) then                              ! matrix.irp.f:   5
   align_double = i                          ! matrix.irp.f:   6
 else                                        ! matrix.irp.f:   7
   align_double = i+4-j                      ! matrix.irp.f:   8
 endif                                       ! matrix.irp.f:   9
end                                          ! matrix.irp.f:  10
```

Output:

```
$ ./test
          20            19
```

To remove all compiler directives introduced by the programmer, it is possible to use *irpf90 --no-directives*.

More about IRPF90

- ArXiv: http://arxiv.org/abs/0909.5012
- Web site: http://irpf90.ups-tlse.fr