ISOffIECffTC1ffSC22ffWff5ffN1357

Object Orientation and Fortran-2002: Part II

Malcolm Cohen, (revisited by AFNOR)

1. Introduction

(no changes)

2. Overview: Inheritance

(no changes)

3. Type Extension

Type extension lets a programmer extend an existing derived type by adding zero or more additional components. The *base type* must be declared with the `EXTENDS()` attribute, and the `EXTENDS(subtype)` clause is used to declare an *extended* type. For example:

```
TYPE,EXTENDS() :: world_point
  REAL latitude,longitude
END TYPE

TYPE,EXTENDS(world_point) :: radio_beacon_point
  REAL frequency
END TYPE
```

Variables of extensible types are declared in the usual fashion, e.g.

```
TYPE(world_point) wp
TYPE(radio_beacon_point) rbp
```

These two variables would have components as follows:

- the components of `wp` are `wp%latitude` and `wp%longitude`;
- the components of `rbp` are `rbp%latitude`, `rbp%longitude` and `rbp%frequency`.

However, a type-cast mechanism is required to extract a supertype from any extensible type. In the previous example, the expression `world_point@rbp` is a variable of type `world_point`, containing only the data fields of a `world_point`. The symbol `@` (at sign) is used here as a supertype-cast operator. The expressions `wp` and `world_point@wp` are equivalent and the expression `radio_beacon_point@wp` is illegal (it is not legal to perform a subtype-cast operation on a variable declared with a `TYPE` statement). Only one type-cast operation is allowed in a casting expression.

Thus assignments like the following are allowed:

```
wp%latitude = 0.0
rbp%longtitude = 1.0
rbp%frequency = 0.5
wp = world_point@rbp
world_point@rbp = wp
```

4. Polymorphic Variables

Polymorphic variables are declared with the `CLASS` keyword, e.g.

```
CLASS(world_point) wpp
```

The variable `wpp` can refer to a data entity of any type in the class of types consisting of `TYPE(world_point)` and any type that extends `TYPE(world_point)`. We say that the *declared type* of `wpp` is `TYPE(world_point)`, and that the *dynamic type* of `wpp` is the type of the data to which it refers. `wpp` can be

- ☐ a dummy argument, in which case its dynamic type is established by argument association (i.e. it gets the dynamic type of its actual argument), or
- ☐ a pointer, in which case its dynamic type is established either by pointer assignment or by allocation.

Note that polymorphic variables only give immediate access to the components of the declared type; to gain access to further extended components, a subtype-cast mechanism is required. For example, the component `frequency` of a type `(radio_beacon_point)` variable `wpp` declared as `CLASS(world_point) wpp` is reached using the expression

```
y = radio_beacon_point@wpp%frequency
```

If this expression is applied on a value of `wpp` with the incorrect dynamic type (i.e. if `wpp` is not of type `(radio_beacon_point)`), a run time error is enabled. Also note that the subtype-cast mechanism *cannot* be used to change the dynamic type of a polymorphic variable.

4.1 Polymorphic Dummy Arguments

Polymorphic dummy arguments allow us to write procedures that will operate on entities of any type extended from a particular extensible type, without needing to use generics or dynamic dispatch. For example, given:

```
REAL FUNCTION distance_between(wp1,wp2)
  CLASS(world_point),INTENT(IN) :: wp1,wp2
  distance_between = (complicated formula using latitude and
          longitude of wp1 and wp2)
END FUNCTION
```

This function can be applied to any two entities of `TYPE(world_point)`, `TYPE(radio_beacon_point)` or any other type extended from `TYPE(world_point)`.

4.2 Polymorphic Pointers

The dynamic type of a polymorphic pointer is that of its target. This may be specified in a pointer assignment statement, or created in an `ALLOCATE` statement. For example:

```
TYPE(world_point),TARGET :: wp
TYPE(radio_beacon_point),TARGET :: rbp
CLASS(world_point),POINTER :: p1,p2
CLASS(radio_beacon_point),POINTER :: rbpp
!
p1 => wp     ! The dynamic type of p1 is now TYPE(world_point)
p2 => rbp    ! The dynamic type of p2 is now TYPE(radio_beacon_point)
rbpp => rbp ! The dynamic type of rbpp is now TYPE(radio_beacon_point)
p2 => p1     ! The dynamic type of p2 is now the same as p1
```

However,

```
rbpp => p1
```

is not directly allowed, since `p1` is not guaranteed to point to an object in the class of `(radio_beacon_point)` types. A subtype cast operation on variable `p1` can be used to do this:

```
rbpp => radio_beacon_point@p1
```

Here, a run-time error is enabled if `p1` do not point to an object in the class of `(radio_beacon_point)` types.

The dynamic type of a polymorphic pointer may also be created in an `ALLOCATE` statement. For example:

```
ALLOCATE(p1) ! Allocates a new object of TYPE(world_point), and
             ! associate p1 with it

ALLOCATE(radio_beacon_point@p2)
             ! Allocates a new object of dynamic type
             ! (radio_beacon_point), and associate p2 with it

ALLOCATE(p2,CAST=p1)
             ! Allocates a new object of the same dynamic type as p1,
             ! and associate p2 with it, Here, a compile-time check
             ! is done to ensure that p1 is a subtype of (or is of
             ! the same type as) the declared type of p2.
```

4.3 Arrays

(no changes)

5 Type Enquiry

Two new intrinsic functions are provided for determining the actual type of a polymorphic variable at runtime. These are:

```
SUB_TYPE(TYP@POLY)       ! Whether the dynamic type of POLY is TYP or is a
                         ! subtype of TYP. The result is .FALSE. if the
                         ! type-cast operation is illegal.

SELF_TYPE(TYP@POLY)      ! Whether the dynamic type of POLY is TYP. The
                         ! result is .FALSE. if the type-cast operation is
                         ! illegal.
```

For instance:

```
CLASS(world_point),POINTER :: p1,p2
ALLOCATE(p1)
ALLOCATE(radio_beacon_point@p2)
!
PRINT *,SUB_TYPE(radio_beacon_point@p1) ! prints F
PRINT *,SUB_TYPE(world_point@p2)        ! prints T
PRINT *,SELF_TYPE(world_point@p1)       ! prints T
```

6. Type Selection

Type selection is provided through the existing IF THEN ELSE construct with type enquiry functions:

```
CLASS(world_point) :: p1
...
IF(SELF_TYPE(world_point@p1)) THEN
  ! Here only if the dynamic type of p1 is TYPE(world_point), not
  ! anything extended from it.
  PRINT *,'Ordinary point at latitude', p1%latitude
ELSE IF(SUB_TYPE(radio_beacon_point@p1)) THEN
  ! Here if the dynamic type of p1 is TYPE(radio_beacon_point) or
  ! any one of its subtypes.
  PRINT *,'This point is a radio beacon'
  PRINT *,'Latitude =', p1%latitude
  PRINT *,'Frequency =', radio_beacon_point@p1%frequency
ELSE
  ! Here only if no other clause was selected.
  PRINT *,'Unrecognized extended point at latitude', p1%latitude
ENDIF
```

However, a type-safe instruction is available to ensure that no runtime error is possible in attempting to access non-existent components. Here, the consistency check can be done at compile-time since the variable `point` is not defined outside the CASE context. The `SELECT TYPE` construct is written:

```
CLASS(world_point) :: p1
...
SELECT TYPE(p1)
CASE (point==world_point@p1)
  ! Here only if the dynamic type of p1 is TYPE(world_point), not
  ! anything extended from it.
  PRINT *,'Ordinary point at latitude', point%latitude
CASE (point.IN.radio_beacon_point@p1)
  ! Here if the dynamic type of p1 is TYPE(radio_beacon_point) or
  ! any one of its subtypes.
  PRINT *,'This point is a radio beacon'
  PRINT *,'Latitude =', point%latitude
  PRINT *,'Frequency =', point%frequency
CASE DEFAULT
  ! Here only if no other clause was selected.
  PRINT *,'Unrecognized extended point at latitude', p1%latitude
END SELECT
```

Note that a subtype-cast operation is required to recover the frequency as `p1` have only immediate access to the components of the declared type `world_point`.

7 Overview: Dynamic Dispatch

(no changes)

8 Procedure Pointers

(no changes)

9 Type-bound Procedures

Type-bound procedures are like procedure pointer components, except that:

- the pointer is fixed (it does not change) and
- there is only one pointer per type, not one per variable

When a type is extended, its type-bound procedures may be inherited by the new type or they may be overridden by specifying a new binding for the type-bound procedure name.

Usually, type-bound procedures will be declared with the `PASS_OBJ` attribute. This attribute causes the object through which the procedure is invoked to be passed as an extra actual argument, to the first suitable dummy argument of the procedure. Such a procedure must have a scalar non-pointer polymorphic dummy argument of the type. For example:

```
MODULE raster_points ! Provides a data type for raster graphics
  PRIVATE
  TYPE,PUBLIC,EXTENDS() :: point
    INTEGER x,y
    INTERFACE plot
      MODULE PROCEDURE,PASS_OBJ :: plot_point
    END INTERFACE
  END TYPE point
CONTAINS
  SUBROUTINE plot_point(obj,screen) ! Draws a point on the screen
    USE x11
    TYPE(screen),INTENT(IN) :: screen
    CLASS(point),INTENT(IN) :: obj
    CALL x11_set_pixel(screen,obj%x,obj%y,x11_white)
  END SUBROUTINE plot_point
END MODULE raster_points
...
PROGRAM ex1
  USE raster_points,ONLY:point
  USE x11
  TYPE(point) x
  TYPE(screen) s
  ...
  CALL x%plot(s)
END PROGRAM ex1
```

In the example above, the procedure reference `x%plot` results in the invocation of `plot_point` from module `raster_points`, with the variable `x` being passed to the `obj` dummy argument.

Note that a fringe benefit of type-bound procedures is that the procedure names occupy the same name-space as component names; they do not pollute the global name-space. This is illustrated in the example above where only the type-name `point` is imported from module `raster_points`, but this does not prevent usage of the type-bound procedure `plot`.

9.1 Inheriting Type-bound Procedures

The next example shows an extension of `TYPE(point)` where an additional component and an additional type-bound procedure are added. The existing type-bound procedure `plot` will be inherited in this new type.

```
MODULE data_points
  USE raster_points
  PRIVATE
  TYPE,PUBLIC,EXTENDS(point) :: data_point
```

```
    READ data(23)
    INTERFACE magnitude
      MODULE PROCEDURE,PASS_OBJ :: magnitude
    END INTERFACE
  END TYPE data_point
CONTAINS
  REAL FUNCTION magnitude(self)
    CLASS(data_point),INTENT(IN) :: self
    magnitude = SQRT(SUM(self%data_point**2))
  END FUNCTION magnitude
END MODULE data_points
```

Note that the module procedure being supplied for the type-bound procedure binding has the same name as the binding itself.

9.2 Overriding a Type-bound Procedure

A type-bound procedure may be overriden in a new type by supplying a different binding for the same type-bound procedure name. The new procedure must have exactly the same characteristics as the old one except for any `PASS_OBJ` argument, which must be a polymorphic scalar of the new type. (The requirement for the characteristics to be the same allows compile-time checking of the argument lists). For example:

```
MODULE pseudo_colour_points
  USE raster_points
  PRIVATE
  TYPE,PUBLIC,EXTENDS(point) :: pseudo_colour_point
    ! Inherits components x and y from point
    INTEGER colour_map_index
    INTERFACE plot
      MODULE PROCEDURE,PASS_OBJ :: plot_pcp
    END INTERFACE
  END TYPE pseudo_colour_point
CONTAINS
  SUBROUTINE plot_pcp(obj,screen)
    USE x11
    TYPE(screen),INTENT(IN) :: screen
    CLASS(pseudo_colour_point),INTENT(IN) :: obj
    CALL x11_set_pixel(screen,obj%x,obj%y,obj%colour_map_index)
  END SUBROUTINE plot_pcp
END MODULE pseudo_colour_points
```

In the above example, the reference to `x%plot` results in a call to `plot_pcp`. As before, `x` is automatically passed to the invoked procedure.

So far the examples have all used variables of fixed type (i.e. not polymorphic). In such cases, a compiler can see at compile-time which procedure is to be called, allowing a static binding. For dispatch actually to be dynamic, a polymorphic object must be used in the calling procedure. For example:

```
PROGRAM ex3
  USE raster_points
  USE data_points
  USE pseudo_colour_points
  TYPE(point) a
  TYPE(data_point) b
  TYPE(pseudo_colour_point) c
  ...
  CALL show(a)
  CALL show(b)
  CALL show(c)
CONTAINS
  SUBROUTINE show(ptt)
    CLASS(point),INTENT(IN) :: ptt
    WRITE(logfile,*) ptt%x,ptt%y ! Save the point to the log file
    CALL ptt%plot                ! Draw the point on the screen
  END SUBROUTINE show
END PROGRAM ex3
```

In the above example, the first call to `show` results in a call of `plot_point` from module `raster_points`, as does the second. The third call to `show` results in a call of `plot_pcp` from module `pseudo_colour_points`.

The dynamic binding capability is enabled when a polymorphic variable is used in the calling procedure (e.g., the variable `ptt` in subroutine `show`). The search for a type-bound procedure is dynamically performed for the dynamic type of the polymorphic variable and, recursively, for each of its next supertype, up to the root.

9.3 Non-Overridable Type-bound Procedures

Sometimes it would not make sense for a type-bound procedure to be overridden in an extension of a particular type. To communicate this information to the compiler, the `NON_OVERRIDABLE` attribute is used. For example:

```
TYPE,EXTENDS() :: mycomplex
  REAL theta,magnitude
  INTERFACE real
    MODULE PROCEDURE,PASS_OBJ,NON_OVERRIDABLE :: real_part
  END INTERFACE
  INTERFACE imag
    MODULE PROCEDURE,PASS_OBJ,NON_OVERRIDABLE :: imag_part
  END INTERFACE
CONTAINS
```

```
    REAL FUNCTION real_part(a)
    CLASS(mycomplex),INTENT(IN) :: a
    real_part = a%magnitude*cos(a%theta)
  END FUNCTION real_part
```

Sometimes, as in the example above, the type-bound procedure is non-overridable in the base type; in other cases, it might become non-overridable only after some extensions. In either case, the `NON_OVERRIDABLE` attribute both prevents the user from overriding it in an extension, and enables the compiler optimization of statically determining which procedure is to be called.