Date: 10 July 1997
To: J3
From: Van Snyder
Subject: Single Inheritance Model – Type Extension

1 Introduction

Specifications with illustrative syntax for single inheritance based on the existing derived type mechanism are provided here. This paper addresses several objections to specifications and illustrative syntax described in papers J3/97-183r2 and J3/97-182r1:

- Objects of extension type should not have an implied component of the same name and type as the base type.
- Primitive operations of a type should not be invoked as though they are components of an object of the type.
- The positional correspondence of actual and dummy arguments should not be "shifted" for primitive operations of a type.
- Primitive operations of a type should be allowed more than one argument of the type.
- It should be possible to indicate a polymorphic type for an expression value.

Extensive background is provided by paper J3/97-194.

2 Declaring a base type

A type that may be extended shall be declared with a distinguishing keyword attribute such as `EXTENSIBLE`. An example declaration might be

```
TYPE, EXTENSIBLE :: vector_2d
  REAL x, y
END TYPE
```

A keyword may not be strictly necessary, but it preserves object-code compatibility with Fortran 95.

3 Extending a base type

A type can be extended anywhere the base type is accessible. Additional components may be supplied, but it is not necessary to do so – the extension could be nothing more than a repackaging of the base type. The components of the base type are components of the extended type – they are *inherited*. Therefore, new component names shall be different from component names of the base type. An extension type is declared by mentioning the base type in its declaration, e.g.

```
TYPE, EXTENDS(vector_2d) :: vector_3d
  REAL z
END TYPE
```

The components of `TYPE(vector_3d)` are `x, y` and `z`.

The accessibility of components of the extended type that are inherited from the base type is the same as their accessibility in the base type. If operations (see 5) of an extension type are declared in a different module from the base type, private components of the base type are not visible to operations of the extension type (see alternatives in 97-114 and 97-194).

The memory layout of the extended type is such that the part of the extended type that corresponds to the base type has the same memory layout as the base type, and occurs at the beginning of the extended type.

An extended type is automatically extensible; the `EXTENSIBLE` keyword may not be re-specified.

Variables of base or extended type are statically typed. The compiler can allocate the correct amount of storage, and resolve generic references that depend on them.

It is possible to reference the base-type part of an extended type by using the base type name as a *view conversion*. For example, given

```
TYPE(vector_2d) v2d
TYPE(vector_3d) v3d
```

Then `v2d` has the components `v2d%x` and `v2d%y`, and `v3d` has the components `v3d%x, v3d%y` and `v3d%z`. The expression `vector_2d(v3d)` is an object of `TYPE(vector_2d)`.

An object of an extended type may be created by using the type name as a *structure_constructor*. The value for the base type part is given by an argument that is an object of the base type, or by arguments that would be allowed as arguments for the base type constructor. Successive arguments (if any) provide values for components that extend the base type. For example

```
v3d = vector_3d(v2d, 0.3)       ! v3d%x = v2d%x, v3d%y = v2d%y
v3d = vector_3d(0.1, 0.2, 0.3)  ! v3d%x = 0.1,   v3d%y = 0.2
```

are both allowed, and are equivalent if `v2d%x == 0.1` and `v2d%y == 0.2`.

A base type, together with all types derived from it by extension, is called a *derivation class*. The base type of an extended type is known as its *parent type*. An extended type, or its parent type, or its parent's parent type, etc., are known as *ancestor types* of the extended type. The base type, and all types in its derivation class, are known as *descendant types* of the base type.

Neither the base type nor the extended type may be a `SEQUENCE` derived type, because there would be no unique type definition with which to associate primitive operations (see 5) or dispatch tables (see 6.4).

4 Polymorphic types

A polymorphic type name denotes the entire derivation class for the specified type. Polymorphic type names are declared by a separate statement, e.g.

```
CLASS(vector_2d) :: vectors
```

declares a polymorphic type `vectors` for the derivation class of `vector_2d`. (Contrast this with Ada's un-Fortran-like `vector_2d'class`.)

Polymorphic variables are declared by using a polymorphic type name, e.g.

```
TYPE(vectors) :: v2 = v2d ! may get only TYPE(vector_2d) values
TYPE(vectors) :: v3 = v3d ! may get TYPE(vector_2d) or
                          ! TYPE(vector_3d) values (see below)
```

A variable of polymorphic type may in principle contain a value of any type in the derivation class. Since it is impossible to know future extensions in a derivation class, a variable of polymorphic type must be a dummy argument, or must have the pointer attribute, or must have an initial value from which it takes its initial monomorphic type (this is an extension of the concept of *automatic arrays*). To provide useful functionality, the initial value expression must be executable.

Use `TYPE_OF` (see 9)?

Once a polymorphic object has been created, it may be assigned only values having types in its derivation class that are ancestral to the monomorphic type of the object at the instant it was created – a run-time check may be required. E.g.

Postponable. Same type only?

```
v2 = v3     ! error if v3 contains a TYPE(vector_3d) value
v3 = v2     ! run-time check will determine this is OK
v2 = v2d    ! OK
v2 = v3d    ! v2 won't be big enough at run time
v3 = v2d    ! OK
v3 = v3d    ! run-time check will determine this is OK
```

Variables of polymorphic type that are structure components must have the pointer attribute.

Only the components of the base type of a polymorphic variable may be referenced directly. An object of polymorphic type may be view-converted to a descendant type e.g. `vector_3d(v3)`, but a run-time check is required to make sure the view-converted type is ancestral to the actual type.

Polymorphic type names can be used as view-converters. For example, assume `OPERATOR(-)` has been defined for `TYPE(vector_2d)`. Then `v2 - vector_2d(v3d)` has `TYPE(vector_2d)`, but `vectors(v2 - vector_2d(v3d))` has `TYPE(vectors)`.

5 Primitive operations of an extensible type

A *primitive operation of an extensible type* is a procedure that is defined in the same procedure or module as the type, and that has at least one dummy argument of the type. E.g.

```
REAL FUNCTION length_2d (v)
  TYPE(vector_2d) :: v
  length_2d = sqrt(v%x**2 + v%y**2)
END FUNCTION length_2d
```

and

```
INTERFACE OPERATOR (+)
  TYPE(vector_2d) FUNCTION plus_2d (a, b)
    TYPE(vector_2d) :: a, b
  END FUNCTION plus_2d
END INTERFACE OPERATOR (+)
```

are primitive operations of `TYPE(vector_2d)`.

If a procedure has more than one dummy argument of a monomorphic extensible type, all such arguments must have the same type. That is, a procedure may only be a primitive operation of one extensible type. Otherwise, operation inheritance (see 5.3) and dispatching (see 6.4) may be ambiguous.

In addition to monomorphic dummy arguments of extensible type, a primitive operation may have dummy arguments of non-extensible types, or polymorphic types.

5.1 Over-riding definition

If primitive operations of both base and extended types exist, have identical signature except for the difference between base and extended type dummy arguments, dummy arguments have the same names, and the procedures appear in a generic interface block or implement the same operator, the primitive operation for the extension type *over-rides* the primitive operation for the base type. E.g. given

```
REAL FUNCTION length_3d (v)
  TYPE(vector_3d) :: v
  length_3d = sqrt(v%x**2 + v%y**2 + v%z**2)
END FUNCTION length_3d
INTERFACE length
  MODULE PROCEDURE length_2d
  MODULE PROCEDURE length_3d
END INTERFACE length
```

and

```
INTERFACE OPERATOR (+)
  TYPE(vector_3d) FUNCTION plus_3d (a, b)
    TYPE(vector_3d) :: a, b
  END FUNCTION plus_3d
END INTERFACE OPERATOR (+)
```

then `length_3d` over-rides `length_2d`, and `plus_3d` over-rides `plus_2d`.

If any of the above conditions are violated, the primitive operation of the descendant type *does not* over-ride the primitive operation of the ancestor type. If they appear in a generic interface block they are "joined" in the ordinary generic sense. This affects dispatching (see 6.4).

5.2 Destructors

A *destructor* is a procedure that is automatically invoked when an object ceases to exist. They are useful in connection with derived types even in the absence of object oriented programming (see 97-194). Is a destructor callable?

Inheritance affects the definition of destructors: Suppose a destructor is declared for an extensible type `T`. It would have exactly one `INTENT(INOUT)` argument of type `T`, and would therefore be a primitive operation of type `T`. It should *not* be allowed to appear in a generic interface, but *should* nonetheless be considered to over-ride a destructor for a base type.

5.3 Inherited primitive operations

If an over-riding definition is not provided, the primitive operation for the base type is *inherited* for use as the primitive operation for the extension type. E.g., if `length_3d` and `plus_3d` were not defined, or not "joined" to `length_2d` and `plus_2d` by generic and operator interfaces, `length_2d` would be a primitive operation both of `TYPE(vector_2d)` and `TYPE(vector_3d)`, and the `OPERATOR(+)` implemented by `plus_2d` would be a primitive operation of both types.

6 Arguments

6.1 Monomorphic actual arguments

An actual argument of monomorphic type must be associated to a dummy argument of ancestral type. The compiler can determine the identity of the primitive operation.

6.2 Polymorphic dummy arguments

A polymorphic dummy argument may correspond to a polymorphic or monomorphic actual argument of descendant type, without run-time checking. A polymorphic dummy argument may correspond to a polymorphic actual argument of ancestor type, subject to a run-time check that the actual type of the actual argument is descendant from the type of the dummy argument.

Procedures may have dummy arguments of several polymorphic types, and of other types (including at most one monomorphic type). Actual arguments that correspond to polymorphic dummy arguments do not participate in over-ride resolution, or in dispatching (see 6.4).

6.3 Polymorphic actual arguments

A polymorphic actual argument may correspond to a dummy argument of an ancestor (mono- or polymorphic) type, without checking. It may be an actual argument corresponding to a dummy argument of descendant (mono- or polymorphic) type, but a run-time check is necessary to verify that the run-time actual argument type is descendant from the declared dummy argument type.

6.4 Dispatching

If a polymorphic actual argument corresponds to a monomorphic dummy argument, over-ride resolution takes place at run time. This is called *dispatching*. For example, suppose `SUB1` is the generic name for two subroutines that take an argument of `TYPE(vector_2d)` and `TYPE(vector_3d)`, respectively. Then `call SUB1(v2)` results in run time selection of the specific procedure. Except in the case of the equality operator (see 7), when several polymorphic actual arguments correspond to monomorphic dummy arguments, the actual arguments must all have the same declared polymorphic type, the same monomorphic type at run time, and the dummy arguments must all have the same declared type. Otherwise, dispatching can be ambiguous.

7 Equality and inequality

Two polymorphic objects may be compared by intrinsic or defined equality or inequality operations, and they need not have the same actual type. If the types are different, then equality returns the value `.FALSE.`, and inequality returns the value `.TRUE.`, without dispatching.

8 Abstract types and procedures

8.1 Abstract types

It is frequently useful to declare a base type, but prevent existence of objects of the type, and therefore operations on the type. Such a type is called *abstract*. An abstract type is declared by adjoining a keyword into the type definition, e.g.

```
TYPE, ABSTRACT, EXTENSIBLE :: image
  REAL x_coord, y_coord
END TYPE image
```

declares a type for which no objects can be declared. (This example might be a base type for objects to be drawn – they all need a position, and the abstract type so specifies.)

8.2 Abstract procedures

An abstract procedure is a procedure that cannot be invoked. It is declared by adjoining a distinguishing keyword to the procedure "header," e.g.

primitive operation?

```
ABSTRACT SUBROUTINE sub1 (a, b)
  ...
ABSTRACT REAL FUNCTION length_0d (v)
  ...
```

A concrete type may not have an abstract primitive operation.

An abstract type may have concrete primitive operations. These can never be reached by dispatching, but they can serve as primitive operations of extension types (if they are not over-ridden).

If an extension type is concrete, then all abstract primitive operations of the base type must be over-ridden by concrete primitive operations.

A concrete base type may have an abstract extension type.

9 Run-time type information

An intrinsic function `TYPE_OF` takes a mono- or polymorphic argument of extensible type, and returns a value of an intrinsic private type that denotes the actual type of its argument. Needed?

The intrinsic relational operators, and only the intrinsic relational operators, e.g. `.GE.`, are defined for this type. Assignment and argument association are not defined.

Elements of derivation classes are partially ordered. The base type of a derivation class is "less" than the base type of an extension thereof. E.g., the expression `TYPE_OF(vector_3d) .GT. TYPE_OF(vector_2d)` has the value `.TRUE.`. If two objects are of types neither of which is an extension of the other, then all relational operations on the results of `TYPE_OF`, other than inequality, return `.FALSE.`, and inequality returns `.TRUE.`