Single Inheritance Model - Type Extension
Specifications and Illustrative Syntax

1. Introduction

This provides specifications with illustrative syntax for single inheritance based on the existing derived type mechanism. The goals and requirements are those set out in paper J3/97-131.

2. Declaring a base type

A type that can be extended shall be declared with the EXTENSIBLE keyword, e.g.

```
TYPE, EXTENSIBLE :: point_2d
      REAL x, y
END TYPE
```

This may not be strictly necessary, but ensures object-code compatibility with Fortran 90/95.

3. Extending a base type

A type can be extended anywhere the base type is accessible. The memory layout of the extended type is such that the base part occurs at the beginning of the type, and components within the base part are laid out the same. It is not necessary to supply any new components (the extension type could simply be a re-packaging of the original type) but any new component names shall not conflict with accessible component names of the type being extended (the base type). The accessibility of components inherited from the base type is the same as in the base type. An extended type is automatically extensible and need not specify the EXTENSIBLE keyword.

```
TYPE point_3d, EXTENDS TYPE(point_2d)
      REAL z
END TYPE
```

It is possible to declare variables of TYPE(point_2d) and TYPE(point_3d) and those variables are statically typed - the compiler can allocate (the correct amount of) storage and resolve generic references at compile time.

The components of TYPE(point_3d) are x, y and z. It is possible to reference the base, i.e. TYPE(point_2d), part of an entity of this type by using the base part's type name as a component selector.
For example, given:

```
TYPE(point_2d) p2d
TYPE(point_3d) p3d
```

these entities have the components

```
p2d%x, p2d%y              ! TYPE(point_2d) only has its own components
p3d%x, p3d%y, p3d%z       ! TYPE(point_3d) has the additional "z" component
p3d%point_2d              ! plus the base part as a whole (the x and y
components)
p3d%point_2d%x, p3d%point_2d%y ! these are redundant but possible
```

Neither the base type nor the extended type may be a SEQUENCE derived type.

4. Polymorphic Variables

Polymorphic variables have a declared base type (that may in itself be an extended type) but can contain an object of that type or of any type extended from its declared base type. Therefore they usually cost more to use than normal variables (i.e. those of static - determined at compile-time - type); e.g. they sometimes imply an extra indirection (because of the reference semantics) and sometimes lose optimisation opportunities. They are declared with a separate keyword, e.g.

```
      OBJECT(point_2d) polly
```

Access via "polly" only provides access to those components in TYPE(point_2d). "polly" can be passed as an actual argument only to a TYPE(point_2d) dummy or an OBJECT(something) dummy where "something" is point_2d or a type extended therefrom. A polymorphic variable can be passed to a dummy argument that is of a parent type. When it is passed to an extended type, the runtime type must be compatible. E.g.

```
      CALL sub1(polly)  ! sub1 expects TYPE(point_2d), so this is
legal
      CALL sub2(polly)  ! sub2 expects OBJECT(point_2d), so legal
      CALL sub3(polly)  ! sub3 expects OBJECT(point_3d), legal
provided
                        ! polly's runtime type is "point_3d" or
extended
                        ! from point_3d.
```

The possible classes of polymorphic variables are:

A. Polymorphic dummy arguments. The actual argument can be of any compatible type (as above, this is the declared base type of the dummy argument or any type extended from that base type). Access to a polymorphic dummy argument is via reference (copy in/out is possible but only when done by the caller). An explicit interface is required for a routine with a polymorphic dummy argument.

B. Polymorphic pointers. These are polymorphic entities with the POINTER attribute, and may be:

- (i) local variables
- (ii) function results
- (iii) structure components

Note that Fortran's auto-dereference facility is ideal for convenient use of polymorphic pointers.

For example: (polymorphic dummy arguments):

```
      REAL FUNCTION argument(p)
            OBJECT(point_2d) p
            argument = ATAN2(p%y, p%x)
      END FUNCTION
      ! No need to redefine ARGUMENT for POINT_3D objects
      REAL FUNCTION azimith(p)
            OBJECT(point_3d) p
            azimuth = ATAN2(p%z,argument(p))
      END
      ! P is a polymorphic dummy argument so it will work with any later extension of
point_3d,
      ! e.g. a point_4d.
```

Another example, (polymorphic pointer variables):

```
        OBJECT(point_2d), POINTER :: p2
        OBJECT(point_3d), POINTER :: p3
        TYPE(point_2d),TARGET :: t2
        TYPE(point_3d),TARGET :: t3
        p2 => t2; p2 => t3; p2 => p3            ! All legal - p2 can point to a point_2d or any
type
                                                ! extended therefrom
        p3 => t2                                ! Illegal, t2 is not extended from point_3d
        p3 => t3                                ! Legal
        p3 => p2                                ! Legal provided p2 is NULL() or its target is
of
                                                ! TYPE(point_3d) or a type extended
therefrom
```

Note that at this point, by construction, arrays of these "objects" are homogeneous. However, non-homogeneous array-like collections are possible using the same circumlocution which allows arrays of pointers.

5. Generic Resolution.

A polymorphic dummy argument can be a "disambiguator" provided that its type is "completely incompatible" with that of its corresponding disambiguator. That is, the type of the corresponding argument must be different (or have different kind type parameters) and must not be extended from the same root type. I.e., they must be in different inheritance trees.

This retains static resolution of generic references and avoids complication.

6. Type Enquiry

There are two enquiry functions provided for determining the exact type of a polymorphic variable at runtime:

(a) SAME_TYPE_AS(POLY,MOLD)

POLY is a polymorphic variable, i.e. OBJECT(something) POLY
MOLD may be polymorphic or of fixed type
The result is .TRUE. iff POLY is referring to an object of the same actual type as
MOLD.

(b) SIMILAR_TYPE_TO(POLY,MOLD)

POLY is a polymorphic variable (as above).
MOLD may be polymorphic or of fixed type (as above).
The result is .TRUE. iff POLY is referring to an object of the same actual type as
MOLD or of a type extended from that type.

In order for these functions to work each extension type must have a unique signature and polymorphic pointers (and dummy arguments) have a "tag" identifying their type. But there is no need for each subobject to contain this tag - in particular since arrays are homogeneous there is no need for each array element to contain such a tag. It would be natural for this signature to be an internal runtime structure holding type-specific information (e.g. the type-bound procedure dispatch table) and for the tag to be the address of this structure.

SAME_TYPE_AS is computable in fixed time simply by comparing the tags on the polymorphic objects. SIMILAR_TYPE_TO can be computed in time proportional to the height of the inheritance tree with a small constant space overhead in the type signature, or in fixed time with a space overhead proportional to the height of the inheritance tree.