It is proposed that for a polymorphic dummy argument to be a “disambiguator”, its type must be “completely incompatible” with its corresponding disambiguator. That is, the two types must not be extension types that are extended from the same original base type (so they are in different inheritance trees).

This retains static resolution of generic references and avoids complication.

Tagging, or Typed Pointers

Tagging is only necessary or useful for entities whose type is not static, i.e. polymorphic entities. In a traditional OO system, tagging performs two main functions

1. the provision to the user of type information; that is, a language capability for checking the actual type of a polymorphic object (e.g. via an intrinsic function).
2. the provision to the runtime system of type information; this can then be used to allow type checking (of polymorphic assignments).
3. a hook for attaching implementation details that vary according to the type, e.g. the method table, layout information (used by a garbage collector), etc.

Tagging need not be expensive; a tag need only be stored for each set of objects of the same type. Essentially, this means including the type information in a polymorphic pointer.

Cost/Benefit Summary:

1. polymorphic pointers are larger than non-polymorphic pointers (typically 4 bytes larger).
2. Intrinsic function SAME_TYPE_AS(A,B) becomes possible (and efficient).

Type-wide method tables could be added with no additional cost (per-object method tables are already possible via the procedure pointer requirement).

This should be read as declaring that “p” can contain an object of any type from the “class” of types extended from “point_2d”. Access to components via “p” only provides access to those in “point_2d”.

The possible types of polymorphic variables are as follows:

1. Polymorphic dummy arguments. The actual argument can be of any type derived from the dummy argument’s class. Access to a polymorphic dummy argument is not more expensive that access to normal dummy arguments.
2. Polymorphic (local) variables.
3. Polymorphic function results.
4. Polymorphic components.

Access to polymorphic variables implies reference semantics (e.g. consider polymorphic components - the space required varies according to the actual type of the component, which varies at runtime - therefore the space for this component must be allocated separately from the rest of the derived type). Therefore, we require that polymorphic variables have the POINTER attribute except for dummy arguments. Note that Fortran’s auto-dereference property of pointers is ideal for this situation.

e.g. (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 AZIMUTH(P)
     OBJECT(POINT_3D) P
     AZIMUTH = ATAN2(P%Z,ARGUMENT(P))
END FUNCTION
! P is polymorphic so it will work with any later
! extension of point_3d, e.g. a point_4d.
```

e.g. (polymorphic 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
P3 => T2       ! Illegal, T2 not extended from point_3d
P3 => T3       ! Legal
P3 => P2       ! Legal provided P2 is NULL() or is
               ! pointing at a TYPE(point_3d) object or
               ! something extended therefrom.
```

Note that at this point, by construction, arrays of “objects” are homogeneous. However, non-homogeneous array-like collections are possible using the same (slightly clumsy) circumlocution which allows arrays of pointers.

Generic Resolution and Polymorphic Dummy Arguments

Single Inheritance Model for Fortran 2000 X3J3/97-131

WG5/N1272

This paper discusses the single-inheritance model of 96-149, 96-172 and 97-106.

The goals of the single inheritance model proposed are as follows:
1. as storage-efficient as normal derived types
2. as execution-time efficient as normal derived types
3. as convenient as possible for the user
4. type extension does not require modifying code using the original type

These goals are satisfied by the model in the previous papers. For convenience we will use the illustrative syntax of the following example:

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

At this point we can declare variables of either type, and those variables are statically typed - so the compiler can allocate (the correct amount of) storage and resolve generic references at compile time.

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

With component selection possibilities of p3d being
     `p3d%point_2d, p3d%x, p3d%y and p3d%z`
(`p3d%point_2d%x` and `p3d%point_2d%y` are possible but redundant).

Since the declaration and use of non-polymorphic entities of extended derived types is exactly the same as that of normal derived types, and is equally efficient, we should use the keyword TYPE for declaring such entities.

Polymorphic Variables

Polymorphic variables are essentially different to those of static (determined at compile-time) type. They usually also cost more to use (sometimes they imply an extra indirection, and optimisation opportunities are often lost); therefore it is proposed that they are differentiated by being declared with a different keyword, e.g.

```
OBJECT(point_2d) p
```