Date: 4 November 1998
To: J3
From: Van Snyder
Subject: Controlled explicit covariance
References: 98-220, 98-222

1 Background

The problem of *binary methods* is frequently discussed in the object-oriented programming literature. A *binary method* is a type-bound procedure that takes two arguments of the type to which it is bound, and, when inherited, both arguments are expected to change to the new type.

A simple example that is frequently used is the 2-dimensional `DISTANCE` function. Suppose the `DISTANCE` function is bound to a type `POINT` that has `X` and `Y` components, and suppose `DISTANCE` has two arguments of type `POINT`. Suppose a new type `COLOR_POINT` is extended from `POINT` by adding a `COLOR` component. The `COLOR` component doesn't participate in calculations of distance between two `COLOR_POINT` objects, so it is possible and reasonable to use the `DISTANCE` function bound to the type `POINT`, rather than to require defining a new, identical one.

The Ada-95 language, and perhaps others, specify that every dummy argument of the type to which the procedure is bound is expected to change to the extended type when it is inherited. The present design for object-oriented programming in Fortran specifies that only the first such argument changes type.

In this case, the Fortran policy is clearly wrong – it doesn't make sense to require converting a `COLOR_POINT` object to a `POINT` object before its distance from another `COLOR_POINT` object can be computed.

In other cases, the Fortran policy is correct. Neither policy is universally applicable, and there appears to be no automatic way to choose which one to use, if both are allowed.

2 Proposal

Allow a specification that dummy arguments and perhaps other objects have the same type as the argument called the *passed-object* dummy argument in 98-007r3. This is the dummy argument associated with the object in which context the procedure is invoked (the *invoking object*).

In 98-220 Werner Schulz advocated a declaration `LIKE(me) :: ARG`, where `me` is the object with which the invoking object is associated (the *passed-object* dummy argument in 98-007r3, the `SELF` object advocated in 98-220, and the `SELF` dummy argument advocated in 98-222). A `LIKE(me) :: ARG` declaration could also be used with the current syntax and semantics of 98-007r3.

Suppose we define

```
  REAL FUNCTION DISTANCE ( A, B )
! REAL FUNCTION DISTANCE (B) SELF (A) ! using notation from 98-222
! REAL FUNCTION A % DISTANCE (B)      ! yet another alternative
    TYPE(POINT) :: A, B
    ...
  END FUNCTION DISTANCE
```

Then when DISTANCE is inherited into COLOR_POINT, the A argument is considered to be of type COLOR_POINT but the B argument remains of type POINT.

Suppose instead we define

```
  REAL FUNCTION DISTANCE ( A, B )
! REAL FUNCTION DISTANCE (B) SELF (A) ! using notation from 98-222
! REAL FUNCTION A % DISTANCE (B)      ! yet another alternative
    TYPE(POINT) :: A
    LIKE(A) :: B
    ...
  END FUNCTION DISTANCE
```

Then when DISTANCE is inherited into COLOR_POINT, both the A and B arguments are considered to be of type COLOR_POINT – that is, the A and B arguments are *covariant*.

The name in parentheses after LIKE is required to be what is called the *passed-object* dummy argument in 98-007r3, the SELF object in 98-220, or the SELF argument in 98-222.

In addition to declaring that dummy arguments are LIKE the SELF argument, it is useful to declare that function results, dummy function results, dummy procedure SELF arguments, and dummy procedure dummy arguments are LIKE the SELF argument.

This form of explicit controlled covariance is useful, safe and easy to explain.

If a DISTANCE function were to be inherited into a 3-dimensional type, say POINT_3D, it would be silly to use it. It would also be silly to use a DISTANCE function that takes a POINT_3D SELF argument, and a POINT dummy argument. The most useful form is to take SELF and dummy arguments of the same type. Therefore, the inherited DISTANCE function can be overridden with one that takes SELF and dummy arguments both of type POINT_3D, not one of type POINT and one of type POINT_3D.

More precisely, the overriding procedure shall have the same characteristics as the inherited procedure, after adjusting the type of the SELF argument and any others LIKE it to be of the extended type.

Without a LIKE(A) :: B declaration, one can simulate the desired effect by using a polymorphic B argument, which may have the undesirable side-effect of unnecessary run-time procedure dispatching if it's used as an invoking object. With a LIKE(A) :: B declaration, in the absence of an exception system, it is important to prohibit polymorphic invoking objects for procdedures that have a LIKE dummy argument, and polymorphic actual arguments associated with LIKE dummy arguments. The semantic of LIKE is that the specified object has the same type as the invoking object. This cannot be verified by a compiler if either the invoking object, or objects associated with LIKE dummy arguments are polymorphic. Without an exception system, there is no way to handle the run-time error that ought to result if the invoking object and an object associated with a LIKE dummy argument have different dynamic types.

Provision for a LIKE(A) :: B declaration in the language standard would not compel its use in object-oriented programs. It would allow a choice and flexibility that would otherwise be absent.