****

J3/02-300r1

Date: October 2002
To: J3
From: Aleksandar Donev
Subject: Improving the OOP Framework in F2x:
Renaming Parent Components &
Type Relations Between Polymorphic Entities &
Reference: J3-007R3, J3/02-295

****

Summary

I discuss two issues relevant to OOP aspects in F2x.

1. I propose that it be allowed to rename the parent components of extended types (see 4.5.3.1, 54:13). This makes codes more readable and only introduces a minor amount of extra work for parsers, but no changes in the language of substance.

2. There is a frequent need to specify relations between the dynamic type of polymorphic entities. This cannot be done in F2x. No solution to this is proposed as I don't have (an implementable) one. But I do want to point out the issue.

J3 response:

The proposed functionality was not in the list of requirements for this revision of the standard. J3 will recommend that your request be added to the list of proposed requirements for the next revision of the standard.

Renaming parent components

The parent component in F2x has the name of the parent type. Many programmers pay close attention to naming types and objects with different conventions, and this immutable rule in F2x will make some codes hard to read. It is easy to correct this by allowing extended types to rename their parent component upon extension. This new name for the parent component will only be valid if the declared type is compatible with the extended type (see J3/02-295):

```
TYPE, EXTENSIBLE :: FileHandlingTicket
   ! A long type name
   ...
END TYPE

TYPE, EXTENDS (PUBLIC :: ticket=>FileHandlingTicket) :: FileHandle
   ...
END TYPE

CLASS(FileHandlingTicket), POINTER :: ticket
CLASS(FileHandle), POINTER :: handle
...
ticket=>handle%ticket
   ! Compare to ticket=>handle%FileHandlingTicket
```

In relation to J3/02-295, ticket%FileHandlingTicket is a valid component selection but not ticket%ticket, unless some other modification is introduced, such as the ability to name the parent component when declaring the base type:

```
TYPE, EXTENSIBLE(ticket) :: FileHandlingTicket
   ...
```

```
END TYPE
```

My preference would be to allow both.

Type relations in OO systems

When extending a type, relations are very often imposed between the dynamic types of various polymorphic entities involved in the specification of the base type ``interface''. These relations come out naturally for the programmer, but the compiler cannot see them. In F2x, it is not possible to specify sych relations or get any compile-time checks in such cases.

Here is one example of this: doubly-linked lists as an extension of singly-linked Lists. It is related to the example in Note 4.50 in the standard draft. A more challenging example is given later

```
TYPE, EXTENSIBLE :: Singly_Linked
    CLASS(*), POINTER :: data
    CLASS(Singly_Linked), POINTER :: next=>NULL()
CONTAINS
    PROCEDURE, PASS(cell) :: ReplaceNext=>ReplaceNextSingle
END TYPE

SUBROUTINE ReplaceNextSingle(cell, replacement)
    CLASS(Singly_Linked), INTENT(INOUT) :: cell ! PASSed
    CLASS(Singly_Linked), INTENT(INOUT) :: replacement ! Not PASSed
    ...
    cell%next=>replacement
    .... ! Also fix replacement's next pointer
END SUBROUTINE
```

Now we extend this to allow double-linked lists. Here is how this would be done in current F2x:

```
TYPE, EXTENDS(Singly_Linked) :: Doubly_Linked
    CLASS(Doubly_Linked), POINTER :: previous=>NULL()
CONTAINS
    PROCEDURE, PASS(cell) :: ReplaceNext=>ReplaceNextDouble
END TYPE

SUBROUTINE ReplaceNextDouble(cell, replacement)
    CLASS(Doubly_Linked), INTENT(INOUT) :: cell ! PASSed
    CLASS(Singly_Linked), INTENT(INOUT) :: replacement ! Not PASSed
    ...
    SELECT TYPE (replacement)
      CLASS IS(Doubly_Linked)
          cell%next=>replacement
          replacement%previous=>cell
         ...! Also fix replacement%next
      CLASS DEFAULT
        WRITE(*,*) "Replace requires a double linkable!"
        STOP
    END SELECT
END SUBROUTINE
```

Notice the need for the SELECT TYPE to make sure that replacement is indeed of the correct type (class), and the very ugly error handling needed to be provided by the user to handle the anomalous case. This is because 55:6-7 says that when overriding a type-bound procedure, one can only "upgrade" the type of the PASSed argument, the rest must remain the same.

The problem here is that the dynamic types of the dummy arguments cell and replacement are not independent. They need to both compatible with the type Doubly_Linked, since one cannot

mix single and double linkables together. Such type relations arise very frequently in OOP.

Type overriding upon extension

One possibility is to allow extensions to specialize the types of polymorphic components and non-passed arguments of type-bound procedures, as in:

```
TYPE, EXTENDS(Singly_Linked) :: Doubly_Linked
    CLASS(Doubly_Linked), POINTER :: next=>NULL()
    ! Redefine (specialize) the CLASS of next
    CLASS(Doubly_Linked), POINTER :: previous=>NULL()
CONTAINS
    PROCEDURE, PASS(cell) :: ReplaceNext=>ReplaceNextDouble
END TYPE

SUBROUTINE ReplaceNextDouble(cell, replacement)
    CLASS(Doubly_Linked), INTENT(INOUT) :: cell ! PASSed
    CLASS(Double_Linked), INTENT(INOUT) :: replacement
        ! Specialize the type of the non PASSed argument
    ...
    cell%next=>replacement
    replacement%previous=>cell
    ....
END SUBROUTINE
```

Any code that now uses Doubly_Linked can perform compile-time checks on the types of the non-passed argument of ReplaceNext to make sure they conform to the type Doubly_Linked.

The problem here is what happens if someone makes a polymorphic pointer:

```
TYPE(Singly_Linked) :: invalid_argument
CLASS(Singly_Linked) :: unknown_type
```

and then calls

```
CALL unknown_type.ReplaceNext(invalid_argument)
```

which won’t work if unknown_type has a dynamic type Doubly_Linked. Should the above call produce an exception? How will this be implemented. These are difficult question. Somehow the fact that the arguments replacement and cell are type related needs to be expressed even in the declaration of the base type Singly_Linked. In Eiffel this is done with "anchors", i.e. declarations of some variables in terms of the type of other variables. How to adopt this to Fortran is not really clear to me.

A challenge

A textbook example of the above issue of expressing type-relations between polymorphic variables that should be a challenge for any Fortran scheme proposed is the need to specify that two type-bound operations, NotEquivalent, and Equivalent, are opposites of one another. In F2x this would be done with (see my paper "Reinstating Deferred Bindings" for the syntax used):

```
TYPE, EXTENSIBLE :: Comparable
CONTAINS
   PROCEDURE(Equivalent), DEFERRED, PASS(A) :: Equivalent
   PROCEDURE, PASS(A) :: NotEquivalent
END TYPE

ABSTRACT INTERFACE
        FUNCTION Equivalent(A, B) RETURN(comparison)
           CLASS(Equivalent), INTENT(IN) :: A, B
```

```
      LOGICAL :: comparison
        END FUNCTION Equivalent
END INTERFACE

FUNCTION NotEquivalent(A, B) RETURN(comparison)
   CLASS(Equivalent), INTENT(IN) :: A, B
   LOGICAL :: comparison
   comparison=.NOT.Equivalent(A,B)
      ! Use the user-provided equivalence test
END FUNCTION NotEquivalent
```

Any extension of Comparable will define its own equivalence testing operation (we could have also made this a type-bound generic operator). However, it will have to test, just like in the previous example, that the argument B is of he correct type, since it has to be of the same type as A for the comparison to be meaningful. At the same time, the relation between NotEquivalent and Equivalent should remain as specified. This is an extra challenge.

The C++ solution I've seen for the problem above is to use a template for the type Comparable, and declare A and B to be of the template type. This is very ugly and not really satisfactory in my opinion.

! EOF