**********

J3/02-294r2

Date: October 2002
To: J3
From: Aleksandar Donev
Subject: Defficient handling of objects in F2x, OR
Default CREATE, COPY and CLONE for polymorphic objects
Reference: J3-007R3, J3/02-295, J3/02-330

**********

The contents of this paper has been restricted to the contents in J3/02-274r1+, which only discusses MOLD in ALLOCATE.

Resolution of the remaining issues is deferred until issues in J3/02-330 have been answered.

## Summary

An OOP language should provide default COPY, CLONE and CREATE operators/methods for polymorphic objects. This way objects, and not just references to them, can be manipulated inside (endogenous) data-structures and other objects-that-manipulate-objects. The critical point is that these need to be accessible via the object itself, even if the type of the object is not!

CLONE is implemented via SOURCE in ALLOCATE. I propose to implement CREATE via a new MOLD argument in ALLOCATE, and COPY by modifying the meaning of intrinsic assignment for polymorphic variables to work on the actual and not on the declared types.

I illustrate the urgent need for these with a very elementary example of implementing a generic stack via arrays. Comments and suggestions on this code are very welcome. The code should really be written using types parameterized with types (a.k.a. templated types), but this is beyond repair for this revision of the language.

## Motivation

In Fortran 2002 (and other OOP languages), polymorphic objects are implemented via dynamic references to them, i.e., POINTER or ALLOCATABLE polymorphic variables. Fortran 2002 provides excellent OOP facilities for manipulating these polymorphic references to objects (see my comments on lack of type specification and overuse of SELECT TYPE though). Thus implementing exogenous dynamic data-structures is supported well. But the language has profound deficiencies when dealing with the objects themselves, so that implementing endogenous structures (and these are very common in scientific programming), is all but impossible.

Eiffel is an OOP language whose design I value greatly (though I would not trade it for Fortran, of course!). It provides three default methods for any object:
CREATE, COPY and CLONE
These three are fundamental operations needed to manipulate objects (and not just references to them) directly. Note that these methods are accessible through the object itself, without access to its type. In the current version of the Fortran draft, CREATE (structure constructors) and COPY (assignment inside SELECT TYPE) are only accessible if the type is accessible.

CLONE is at present the only one provided in F2002 via the SOURCE argument to ALLOCATE. I feel strongly that the other two need to be provided also. CREATE can most easily be provided by adding a MOLD argument to ALLOCATE similar to the SOURCE argument, which would only give the type of the allocated data, but not the contents. This would allow one to allocate an array of a given type given only a scalar of that type, which is needed, for example, to design custom allocators for various dynamic data structures.

COPY is trickier. I think it should be provided by modifying the meaning of intrinsic assignment for polymorphic variables to work on the actual and not on the declared types. I propose two possible interpetations in the case when the dynamic types of the left and right hand sides defer, and propose to accept the simpler of the two choices. This also requires a modification of the way base component selection works for polymorphic objects, so that the current meaning of intrinsic assignment can be used if needed (for efficiency). This will also be proposed in a separate paper J3/02-295.

---

Example: Array-based stack

---

A homogeneous stack can be implemented via an array. The allocation and deallocation of the array should be handled inside the stack implementation. But at present this is not possible in Fortran since the type of the allocated array is dynamic and cannot be given to ALLOCATE. Furthermore, when the stack fills up, the contents of the array should be copied to a larger array (reallocation). Again, this copy is very hard to do in Fortran 2002, since intrinsic assignment works on the declared types. One must therefore ask clients of the Endogenous_Stack class to themselves implement defined assignment and creation. This is unneccessary and very cumbersome for such a common and elementary application.

Here is how the stack would be implemented in my modified Fortran 2002. Later I will write this in what I think should be in Fortran 200+ (i.e., the next revision), using generic objects (templates in C++, unconstrained generic type parameters in Eiffel).

```
MODULE Endogenous_Stacks
   ! For homogeneous stacks only!
   ! NOTEs:
   ! CREATE, COPY and CLONE are all needed and used in this example.
   ! Data is not carefully encapsulated--assume the user is trustable!
   ! Memory allocation failure is also *not* checked!

   TYPE, EXTENSIBLE, PUBLIC :: Endogenous_Stack
      CLASS(*), POINTER :: stack_mold=>NULL()
         ! What is this a stack of?
      INTEGER :: actual_size=0
      INTEGER :: expected_size=1000
   CONTAINS
      PROCEDURE, DEFERRED, PASS(stack) :: PushOnStack, PopOffStack
   END TYPE Endogenous_Stack

   ABSTRACT INTERFACE
      FUNCTION PushOnStack(stack,element) RESULT(success)
         CLASS(Endogenous_Stack), INTENT(INOUT) :: stack
         CLASS(*), INTENT(IN) :: element
         LOGICAL, INTENT(OUT) :: success
      END FUNCTION PushOnStack
      FUNCTION PopOffStack(stack,element) RESULT(success)
         CLASS(Endogenous_Stack), INTENT(INOUT) :: stack
         CLASS(*), INTENT(OUT) :: element
```

```
         LOGICAL, INTENT(OUT) :: success
      END FUNCTION PopOffStack
   END INTERFACE

END MODULE Endogenous_Stacks

MODULE Endogenous_Array_Stacks
   USE Endogenous_Stacks

   TYPE, EXTENDS(Endogenous_Stack), PUBLIC :: Endogenous_Array_Stack
      CLASS(*), DIMENSION(:), ALLOCATABLE :: storage
         ! Array-based implementation
   CONTAINS
      PROCEDURE, PASS(stack) :: PushOnStack=>PushOnArrayStack, &
                                PopOffStack=>PopOffArrayStack
   END TYPE Endogenous_Array_Stack

CONTAINS

   FUNCTION PushOnArrayStack(stack,element) RESULT(success)
      CLASS(Endogenous_Stack), INTENT(INOUT) :: stack
      CLASS(*), INTENT(IN) :: element
      LOGICAL, INTENT(OUT) :: success

      CLASS(*), DIMENSION(:), ALLOCATABLE :: temp_storage
         ! For reallocation

      IF(.NOT.ASSOCIATED(stack_mold)) THEN
         WRITE(*,*) "No mold for stack!"
         success=.FALSE.
         RETURN
      ELSE
         IF(.NOT.SAME_TYPE_AS(stack_mold,element)) THEN
            WRITE(*,*) "Wrong argument type for stack push"
            success=.FALSE.
            RETURN
         END IF
      END IF

      ! No allocation error handling here for now:
      IF(.NOT.ALLOCATED(stack%storage)) &
         ALLOCATE(stack%storage(stack%expected_size), MOLD=stack_mold)
         ! Use the MOLD argument to CREATE the array

      stack%actual_size=stack%actual_size+1
      IF(SIZE(stack%storage)<stack%actual_size) THEN

         ! Reallocate the stack storage
         ALLOCATE(temp_storage(SIZE(stack%storage)), MOLD=stack_mold)
         temp_storage=stack%storage
            ! Use extended assignment to COPY the stack contents

         DEALLOCATE(stack%storage)
         ALLOCATE(stack%storage(2*SIZE(temp_storage)+1), &
            MOLD=stack_mold)
            ! Double the stack size
         stack%storage(1:SIZE(temp_storage))=temp_storage
            ! Copy the contents back

         DEALLOCATE(temp_storage)
      END IF

      stack%storage(stack%actual_size)=element
         ! The actual push, also a COPY
```

```
      success=.TRUE.
      RETURN
   END FUNCTION PushOnArrayStack

   FUNCTION PopOffArrayStack(stack,element) RESULT(success)
      CLASS(Endogenous_Stack), INTENT(INOUT) :: stack
      CLASS(*), INTENT(OUT) :: element
      LOGICAL, INTENT(OUT) :: success

      IF(stack%actual_size<1) THEN
         ! Empty stack
         success=.FALSE.
         RETURN
      END IF

      ALLOCATE(element, SOURCE=stack%storage(stack%actual_size))
         ! Now use CLONE via SOURCE
      stack%actual_size=stack%actual_size-1

      success=.TRUE.
      RETURN
   END FUNCTION PopOffArrayStack

END MODULE Endogenous_Array_Stacks
```

---

Solution

---

Assume:

```
CLASS(base_type), POINTER :: variable, expression
```

Part 1.

To fix the above defficiency, I propose to add a MOLD argument to ALLOCATE, with the same semantics as SOURCE, but without the contents in the lines 110:8-10. The mold variable will thus simply serve as a type template for the creation (allocation) of the new object, and can be a scalar even if the allocate-object is an array. Initialization will occur for the newly allocated variable.

In summary,

```
ALLOCATE(variable, MOLD=expression)
```

should be equivalent to:

```
ALLOCATE(dynamic_type_of_expression::variable)
```

when the type of the mold is accessible.

Part 2.

Intrinsic assignment should also be modified to work with the dynamic, and not the declared type of polymorphic variables. The basic idea is to remove the "variable shall not be polymorphic" in 132:5. The question to decide is what requirements we have on the dynamic types of the left and right-hand side. I propose a specific solution here, but other possibilites are possible if J3 believes them to be more appropriate.

In summary, the intrinsic assignment

```
variable=expression
```

should act as if the following defined assignment was bound to the dynamic type of variable, dynamic_type_of_variable (which must be an extension of base_type):

```
SUBROUTINE DefaultAssignment(rhs,lhs)
   CLASS(base_type), INTENT(IN) :: rhs
   TYPE(dynamic_type_of_variable), INTENT(OUT) :: lhs

   SELECT TYPE(rhs)
      CLASS IS (dynamic_type_of_variable)
         lhs=rhs ! In the sense of 007R3
      CLASS DEFAULT
         lhs%base_type=rhs ! In the sense of 007R3
   END SELECT

END SUBROUTINE
```

## Problems and Alternatives

### CREATE

Fortran does not really have creation operators a-la C++. However, we do have default constructors, which are the ones most commonly used and needed. If something more is needed, the user should make a type-bound generic constructor. The default constructor should however be accessible via the object itself, without the need to access its actual type.

To implement allocation with SOURCE, the compiler only needs to know the size of the allocation. There is no need to initialize the allocated storage to anything since the contents of the source is immediately copied into the new storage. However, for MOLD, it is not trivial to initialize the storage. In particular, if we want default initialization to happen, then compilers really need to create an actual creation operator (a default one) for all extensible types (that is, a pointer to a creation procedure must become a part of the class slot table). I do not see this as any problem, since there is only one slot table per type and there is no overhead other then a minimimal extra work on the part of the compiler.

However, it is also possible to specify that only default initialization of the base type will happen when allocating from a MOLD. I find this undersirable, but acceptable.

### COPY

#### Efficiency

A major issue with extending intrinsic assignment to be a COPY method is efficiency and implementation. Will the need for type checking slow down programs that do not need this COPY? To prevent this, and also to allow a greater expressivity in the language, one needs to allow the selection
polymorphic_object%base_type
for objects declared with
CLASS(base_type), ... :: polymorphic_object

so that the present form of intrinsic assignment, i.e., working with the declared (base) type, can still be done:

```
CLASS(base_type), POINTER :: variable, expression
...
variable%base_type=expression%base_type ! Copies only the base
```

## Exact meaning of polymorphic assignment

After meeting 162, the situation with polymorphic assignment is as follows:
The declared types of the rhs and lhs must match, and it is allowed for the rhs to be polymorphic and for its dynamic type to be an extension of the lhs, in which case only the declared (base) part is copied.

The change I propose is to modify this so as to also allow the lhs to be polymorphic. What the assignment does depends on the dynamic types. One thing is clear: If the dynamic type are the same, then the rhs object should be copied completely to the lhs object. The less-clear issue, which is to be decided by a majority, is what to do when the dynamic types of variable and expression do not match. From the start, I eliminate the option of raising some kind of exception when this happens, as we have no means for trapping those. Thefore legal intrinsic assignment should always complete normally.

I see two possibilities:

1. Copy the part in common, i.e. choose the type in the type hierarchy where the two types diverge and copy it:

```
variable=expression
```

is really

```
variable%last_common_type=expression%last_common_type
```

Since variable and expression must at least share the base type, there will always be some useful action performed by this copy.

2. (Preferred)
If the dynamic rhs type is an extension of the dynamic lhs type, then copy the portion of the rhs corresponding to the lhs completely. Otherwise just copy the base part:

```
variable=expression
```

is really (this is not really legal Fortran, but an illustration):

```
SELECT TYPE(expression)
   CLASS IS(dynamic_type_of_variable)
      variable=expression%dynamic_type_of_variable
   CLASS DEFAULT
      variable%base_type=expression%base_type
END IF
```

3. Almost the same as choice (2), but also allow for the possibility that the lhs type extends the rhs type:

```
SELECT TYPE(expression)
   CLASS IS(dynamic_type_of_variable)
      variable=expression%dynamic_type_of_variable
      assigned=.TRUE.
END IF
```

```
SELECT TYPE(variable)
   CLASS IS(dynamic_type_of_expression)
      variable%dynamic_type_of_expression=expression
      assigned=.TRUE.
END IF

IF(.NOT.assigned) THEN
   variable%base_type=expression%base_type
END IF
```

The third option is a minor variation of the second one.

I expect the second option will be more efficient and simpler to implement. In all applications I can think of, choice 2 will do. It also has the advantage of being clear and void of ambiguity, and it can also be implemented as a type-bound procedure for any polymorphic type, just like creation. But as already stressed in this paper, the need is to provide it as a default operator for all polymorphic objects.

## Edits

Will be written after comments are received.

! EOF