Class inheritance and dynamic binding polymorphism in Fortran 2000

ISO/IEC JTC1/SC22/WG5 N1188

A. Fortran 2000 requirements submission

Number: 88
Submitted By: AFNOR
Status: For Consideration

References:


Basic Functionality: We propose to supplement Fortran with two object-oriented mechanisms: class inheritance and dynamic binding polymorphism. The proposed extensions are borrowed from Ada 95 with a syntax more compatible with the Fortran culture.

Rationale: Object orientation is required to transfer information in a consistent way between components of large software projects. Technological objects can be mapped into Fortran-2000 objects and transferred from parts of a large software to others. An object in Fortran-2000 is simply an extensible derived type with its information hidden using a “PRIVATE” attribute. This information is reached through methods, i.e. public procedures belonging to the same module.

Estimated Impact: The number of extensions is kept as small as possible in order to speed-up the Fortran-2000 compiler development and to avoid complexifying the language. Dynamic binding polymorphism will be implemented using the dispatching mechanism of Ada-95 and will solve many problems associated with the strong typing character of Fortran-90. Inheritance and dispatching will be implemented using only two new statements (INHERIT and CLASS) and with the introduction of class transformation functions.

Detailed Specifications:

B. Type extension and inheritance

Type extension is the mechanism used to add components to a TYPE statement. Type extension and inheritance are accomplished using only one statement: INHERIT. Only derived types with the statement INHERIT can be extended. An extensible type always extends the information contained in the existing type indicated after the INHERIT word. If the statement INHERIT is not followed by a type, a pre-defined root type is assumed. The root type is a pre-defined type with no information included. Any extensible type
Inherit from the root type or from an existing user-defined extensible type. Multiple inheritance is not allowed. The example found in section 4.2 of Ref.(1) is now written:

```fortran
TYPE HUMAN
  INHERIT
  CHARACTER(Len=4) :: FIRST_NAME
END TYPE HUMAN

TYPE MAN
  INHERIT HUMAN
  LOGICAL :: BEARDED=.FALSE.
END TYPE MAN

TYPE WOMAN
  INHERIT HUMAN
END TYPE WOMAN
```

In this example, both types `MAN` and `WOMAN` are derived from `HUMAN`. `MAN` extends `HUMAN` by adding a new data field, `BEARDED`.

There is no need to include `SUPER` and `SELF` indirections with this model because the receiver is explicitly written in the method parameters. Depending on its type, the corresponding method will be activated.

However, a type transformation function is requested: If `OBJ2` is of type `MAN` (a super type of `HUMAN`), then the function `HUMAN(OBJ2)` have the `POINTER` attribute and returns a corresponding object of type `HUMAN`. The type transformation function own an important behaviour: If the type transformation function is applied on a variable which is not a subtype or which is not a type corresponding to the name of the function, then the function returns an empty type. If `OBJ2` is of type `MAN`, then `WOMAN(OBJ2)` returns an empty type (since `MAN` and `WOMAN` are distincts). Similarly, if `OBJ3` is of type `HUMAN`, then `MAN(OBJ3)` or `WOMAN(OBJ3)` returns an empty type (since `OBJ3` is a super-type of `MAN` and `WOMAN`). This feature will be used to test the the type or super-type membership of a variable. Note that type transformation functions and also exists in Ada-95.

Each type belonging to an extensible hierarchy own a distinct kind number. The empty type have its `KIND` equal to `-1`. A new intrinsic (and elemental) function named `CLASS_KIND` is available to obtain the `KIND` parameter (an integer value) corresponding to a scalar variable. If `OBJ2` is of type `MAN`, then `CLASS_KIND(MAN(OBJ2))` and `CLASS_KIND(HUMAN(OBJ2))` are both functions returning positive kind numbers. Similarly, `CLASS_KIND(WOMAN(OBJ2))` is a function returning `-1` (since `WOMAN(OBJ2)` is empty).

In the following example, an extensible type is defined as a public defined type with a private internal structure. Information within this structure is reached through standard Fortran-90 generic procedures:

```fortran
MODULE OBJ_PACK
  PRIVATE
  PUBLIC :: TABLE_OBJ,OBJOP,OBJACT,OBJDES,OBJCL,ASSIGNMENT(=)
  TYPE TABLE_OBJ

  PRIVATE
  PUBLIC :: TABLE_OBJ,OBJOP,OBJACT,OBJDES,OBJCL,ASSIGNMENT(=)
  TYPE TABLE_OBJ
```
Let us now assume that this object model is not sufficient for a specific project and that a SIGNATURE field should be added in the object attributes. A new object can be defined with an extended type TABLE_OBJ together with new OBJOP and ASSIGNMENT(=) procedures to manage the SIGNATURE field:

MODULE OBJ_PACK_2
USE OBJ_PACK
PRIVATE

! use the module containing TABLE_OBJ
USE OBJECT
PRIVATE

INTERFACE OBJOP
MODULE PROCEDURE OBJOP
END INTERFACE

INTERFACE OBJACT
MODULE PROCEDURE OBJACT
END INTERFACE

INTERFACE OBJDES
MODULE PROCEDURE OBJDES
END INTERFACE

INTERFACE OBJCL
MODULE PROCEDURE OBJCL
END INTERFACE

INTERFACE ASSIGNMENT(=)
MODULE PROCEDURE OBJEQ
END INTERFACE

CONTAINS

SUBROUTINE OBJOP(PFIRST,IMODE)
TYPE(TABLE_OBJ) :: PFIRST
INTEGER, OPTIONAL :: IMODE
PFIRST%MODE=IMODE
END SUBROUTINE OBJOP

PRIVATE
PUBLIC :: TABLE_OBJ_2,OBJOP,OBJACT,OBJDES,OBJCL,ASSIGNMENT(=)

TYPE TABLE_OBJ_2
  INHERIT TABLE_OBJ
  PRIVATE
  CHARACTER(LEN=12) :: SIGNATURE
END TYPE TABLE_OBJ_2

INTERFACE OBJOP
  MODULE PROCEDURE OBJOP_2
END INTERFACE

INTERFACE ASSIGNMENT(=)
  MODULE PROCEDURE OBJEQ1,OBJEQ2
END INTERFACE

CONTAINS

SUBROUTINE OBJOP_2(PFIRST,MODE,Signature)
  TYPE(TABLE_OBJ_2) :: PFIRST
  INTEGER, OPTIONAL :: MODE
  CHARACTER(LEN=12), OPTIONAL :: Signature
  IF(PRESENT(Signature)) THEN
    PFIRST%SIGNATURE=SIGNATURE
  ELSE
    PFIRST%SIGNATURE=''
  ENDIF
  CALL OBJOP(TABLE_OBJ(PFIRST),MODE) ! Use type transformation function
END SUBROUTINE OBJOP_2

SUBROUTINE OBJEQ1(PFIRST,PFIRST2)
  TYPE(TABLE_OBJ_2), INTENT(INOUT) :: PFIRST
  TYPE(TABLE_OBJ), INTENT(IN) :: PFIRST2
  CALL OBJEQ(TABLE_OBJ(PFIRST),PFIRST2)
  PFIRST%SIGNATURE=''
END SUBROUTINE OBJEQ1

SUBROUTINE OBJEQ2(PFIRST,PFIRST2)
  TYPE(TABLE_OBJ_2), INTENT(INOUT) :: PFIRST
  TYPE(TABLE_OBJ_2), INTENT(IN) :: PFIRST2
  CALL OBJEQ(TABLE_OBJ(PFIRST),TABLE_OBJ(PFIRST2))
  PFIRST%SIGNATURE=PFIRST2%SIGNATURE
END SUBROUTINE OBJEQ2
END MODULE OBJ_PACK_2

This extended type is next used from within our project as
USE OBJ_PACK_2
TYPE(TABLE_OBJ_2) :: PTYP2,PTYP3
INTEGER,POINTER :: IP
CALL OBJOP(PTYP2,1,'SIGN_DATA') ! 'SIGN_DATA' is the SIGNATURE
CALL OBJACT(TABLE_OBJ(PTYP2),'ITEM-1',IP)
PTYP3=PTYP2
...

Note that a call of the form CALL OBJACT(PTYP2,'ITEM-1',IP) would be illegal because the procedure OBJACT is defined for TABLE_OBJ derived types only (the strong typing of Fortran-90 is preserved). This difficulty is solved by the polymorphism mechanism described in the next section.

C. Dispatching

The difficulty presented at the end of the previous section is solved by replacing the statement

TYPE(TABLE_OBJ_2) :: PTYP2

with

CLASS(TABLE_OBJ) :: PTYP2

and by setting PTYP2 to type TABLE_OBJ_2 using an auto-specialization mechanism. Here, we note the addition of the second statement: CLASS.

A class in Fortran-2000 is similar to a class in Ada-95. It is an open-ended hierarchy of types, collecting in a unique declaration an extensible type and all the sub-types derived from this type. All types belonging or extended from a particular extensible type ETYPE belong to a derivation class CLASS(ETYPE) of ETYPE. For example, a variable T1 can be declared to be an element of the class HUMAN using the following declaration:

CLASS(HUMAN) :: T1

In this case, T1 can be whether a HUMAN, a MAN or a WOMAN. The correct type is selected dynamically, at execution time, as a function of the software requirements.

Note that a declaration of the form

TYPE(HUMAN) :: T2

does not have the same meaning as the CLASS declaration, since T2 can only hold the instance variables and the methods of an HUMAN. On the other hand, T1 can hold the instance variables and the methods of a MAN or of a WOMAN. Finally, note that any extensible type can be contained in a variable declared as root class:

CLASS() :: T3
Polymorphism and dynamic binding capabilities are supported on Fortran-2000 through two powerful mechanisms:

a) **Auto-specialization:** This is the capability of a extensible type declared with the `CLASS` statement to become more and more specialized as data fields belonging to its sub-types are used or as methods belonging to its sub-types are called (however, it cannot become less specialized).

b) **Method dispatching:** This is the capability of a extensible type declared with the `CLASS` statement to be the receiver of methods belonging to this type or belonging to its super-types. In the latter case, the type transformation function is called automatically.

Consider the following example in which extended types T01 to T06 are defined with corresponding methods:

```
(root)                      super-type
|                           |
T01 with methods: get_info  ||
| put_info                  /
| method1                   \

-------------------------------------

|                           |
T02 with methods: get_info T03 with methods: put_info
| put_info                  |
| method2                   |

-------------------------------------

|                           |
T04                           |
| T05                           |
| T06 with methods: method4     |
```

If a variable `OBJ` is declared as

```
CLASS(T01) :: OBJ
```

then `OBJ` is initially of type T01. Its type can be subsequently changed to T02 or T03 depending on which extended data fields are used or which methods are called. If its type is set to T03, it can subsequently be changed to T04, T05 or T06 (but it cannot be changed back to type T02 or T03). This is the auto-specialization mechanism. If `OBJ` is declared with the `DIMENSION` attribute, each element of the `OBJ` array can have a different specialization.

At any time, the membership of `OBJ` can be checked using operations of the form:

```
IF(CLASS_KIND(T02(OBJ))/=-1) THEN
```

or
IF (CLASS_KIND(T03(OBJ)) /= -1) THEN.

If the variable OBJ is set to type T03, then statements CLASS_KIND(T01(OBJ)) and CLASS_KIND(T03(OBJ)) are elemental function returning positive kind values. Statements CLASS_KIND(T02(OBJ)) and CLASS_KIND(T06(OBJ)) are elemental functions returning -1.

If the variable OBJ is finally set to type T06 and a call of the form

CALL put_info(OBJ,PAR1,PAR2)

is performed, then a check is done to see if the method put_info can operate on instances of type T06. Since this is not the case, a check is done in the next super-type T03. Here, the put_info method is found and the previous call is automatically transformed into

CALL put_info(T03(OBJ),PAR1,PAR2)

Similarly, a call of the form

CALL get_info(OBJ,PAR1,PAR2)

is automatically transformed into

CALL get_info(T01(OBJ),PAR1,PAR2)

Finally, if OBJ is set to type T03 and if a call of the form

CALL method4(OBJ,PAR3)

is performed, then the type of OBJ is automatically specialized to type T06 before the call is performed.

These are examples of the method dispatching mechanism.

**Note 1:** A procedure can have many parameters defined with the CLASS statement. Multi-methods are therefore allowed in Fortran-2000.

**Note 2:** A CLASS declaration can be combined with the POINTER attribute.

**Note 3:** A “dispatching failure” message is issued in cases where the run-time system is unable to resolve the request (e.g., if a method belonging simultaneously to two sub-types is called; in this case, the run-time system is unable to chose the sub-type into which to specialize).
D. Alternative syntax of the above proposal

Instead of using class transformation functions “à la Ada”, one could introduce a single intrinsic predefined POINTER function

\[ \text{SUPER} \left( \text{OBJECT[, KIND]} \right) \]

whose value is pointing on the direct super-type component of OBJECT if the parameter KIND is absent, or on the component of the super-type component specified by KIND if present. If OBJECT is not in a valid sub-type of the specified class, the “void” object is returned (that is, \( \text{CLASS.KIND(SUPER(OBJECT[, KIND])}) \) is -1).

The proposed name \( \text{SUPER} \) is ending with “_” in order to keep the simple name “SUPER” familiar to Object-Oriented people, though avoiding conflicts with existing identifiers, as “_” was not a legal character in the FORTRAN-77 standard.