Date: 8 February 1997
To: X3J3
From: Van Snyder
Subject: Extended TYPE and SUBTYPE Notation for Fortran

Introduction

Some of the irregularity in Fortran is caused by the absence of a consistent and complete type and subtype system.

Some of the present proposals for additions in Fortran 2000 attack this problem obliquely, or suffer themselves from the absence of a complete and consistent type and subtype system.

We propose here a syntax for type and subtype declaration, based on present Fortran syntax conventions and interpretations.

We propose for discussion several applications of a more complete and consistent type and subtype system. We *do not* assert that this proposal provides a complete discussion of the problem.

Much of the problem of types and subtypes has been previously considered in depth by workers in other but related fields. We commend to you especially the Ada-83 and Ada-95 standards and rationales, and related textbooks, monographs and articles. The doctoral dissertation and other work by Paul Hilfinger at MIT, and work by Luca Cardelli, are also germane.

1 Attributing the TYPE statement

Fortran has developed a tradition of using *attributes* to refine the meaning of declarations. At present, the only attribute allowed for a *type-declaration-stmt* is *access-spec*. We propose to allow additional existing attributes, *viz.* ALLOCATABLE or POINTER (but not both), DIMENSION and EXTERNAL.

We propose a new attribute for TYPE, *viz.* INTERFACE, explained below.

Every object declared to be of a type having attributes enjoys simultaneously all the attributes specified for the type.

1.1 Interaction of POINTER and DIMENSION attributes

If a TYPE has both the POINTER and DIMENSION attributes, an object of that type is a "pointer to an array," just as would be an object that had both attributes specified in its declaration.

If a TYPE has the DIMENSION attribute and an object of the type has the POINTER attribute, the object is a "pointer to an array".

If a TYPE has the POINTER attribute, and an object of the type has the DIMENSION attribute, the object is an "array of pointers."

If a TYPE has the POINTER and DIMENSION attributes, and an object of the type has the DIMENSION attribute, the object is a "an array of pointers to arrays."

1.2 Arrays of arrays

If a TYPE has the DIMENSION attribute, and an object of the type has the DIMENSION attribute, the object is an "array of arrays." In this case, the array is stored in *row major* order. We propose that the syntax for reference to an element of an object of the type, say A, should be A(i)(j). This is a consistent extension to the notation used to access objects of CHARACTER type.

1.3 The EXTERNAL attribute

The external attribute is used to declare that the storage organization of a type is to be the same as for another processor, perhaps for a different language. In the document N1237.alt, for example, we propose using EXTERNAL(C) to indicate that a Fortran derived type is to have the same storage organization as a C language `struct`.

1.4 The INTERFACE attribute

If a *type-declaration-stmt* includes the INTERFACE attribute then the body of the type declaration is identical in form and meaning to an interface body. An object of the type is a "procedure valued variable." It is unlikely one would wish to store the body of a procedure, so an object of a type bearing the INTERFACE attribute almost certainly would also be a "pointer to procedure," even if it did not enjoy the POINTER attribute.

At present, it appears that the only meaningful declaration is of a single explicit interface to a procedure. That is, declaration that a type is an interface to ASSIGNMENT(=) or OPERATOR(...) appears not to be valuable.

Declaration that a type is an interface to a generic collection of procedures might have value for *dynamic polymorphism* or *dynamic dispatching*, a topic important to object oriented programming.

A procedure may be "assigned" to a variable having the INTERFACE attribute only if it has identical characteristics to the INTERFACE.

Here is an example.

```
TYPE, INTERFACE :: TS ! omit TS and use S for the type name?
  SUBROUTINE S ( A, I ) ! The name S is irrelevant
    REAL :: A(:)
    INTEGER I
  END SUBROUTINE S
END TYPE TS
TYPE(TS) :: PS
...
PS => SUB1 ! SUB1 characteristics must match TS
...
CALL PS ( X(1:5), 3)
```

1.5 Types as attributes

A type, including an intrinsic type, can be an attribute of a type. This introduces a new type, even if no additional attributes are specified. Thus `TYPE, POINTER, TYPE(A) :: B` introduces a new type B, objects of which are pointers to objects of type A. The type used as an attribute is called the *base* type.

This allows pointers to pointers ... to pointers .... Since Fortran uses automatic dereferencing, this is a problem. If we have a type TB that resolves to "pointer to pointer to pointer to integer," X is of type TB and Y is integer, what do `X = Y` and `X => Y` mean? If this problem cannot be resolved, it would be best to prohibit types to have the POINTER attribute.

If parameterized derived types are introduced into Fortran, then a new type may take generic parameters that specify some or all of the parameters of the base type. If some of the parameters of the base type are specified as constants, and some by parameters of the new type, the effect is similar to a technique in functional programming known as "partial application."

2 Subtypes

A *subtype declaration* introduces a new name that denotes an existing type, perhaps with restrictions. If there are no restrictions, then a subtype declaration effectively introduces a synonym for an existing type.

A subtype declaration may declare a subtype of an intrinsic type.

As in WG5/N1237 we propose to use the `=>` notation to indicate a subtype name, with usage consistent to usage in the USE statement. That is, the name on the left side of `=>` is the new (subtype) name, and the declaration on the right side of `=>` declares the type or subtype from which the subtype is derived (the *base* type).

Although the base type may have attributes, neither the base type reference nor the new subtype specification may have adjoining attribute specifications in the subtype declaration. For example, one cannot declare a subtype to be an array of or pointer to the base type.

As for types, a subtype of a generic type or subtype may specialize zero, some or all of the generic parameters of the base type or subtype.

Subtypes of INTEGER can presently be only imprecisely described, by using SELECTED_INT_KIND. Declaration of subtypes of integer should be made more precise. One way is to allow declaration of a KIND parameter by using an extension of SELECTED_INT_KIND that takes an argument of type SEQUENCE (see X3J3/97-114) to denote the lower and upper bound for values of the subtype. Another way is to introduce a new type parameter (not an attribute), say RANGE, to denote the bounds. For example

```
  TYPE :: I10 => INTEGER(RANGE=1:10)
```

denotes a subtype of integer, for which values of objects of the subtype must be in the range 1:10.

Dimension declarations should be extended to allow subtypes of integer, in addition to objects of subtypes of integer, to be dimensions. Continuing the previous example `REAL, DIMENSION(I10) :: A` declares an array having dimension 1:10, and that subscripts must be of subtype I10 or a subtype thereof. If a compiler performs range checking of values assigned to objects of a subtype, then when a subscript consists of a variable of the same subtype as the dimension of the array, the value of the subscript is known to be within range. One can continue this argument backward by induction through assignments from objects of the same subtype, or loop inductors. The effect is that in many cases one gets array bounds checking at no run-time cost – it's a compile-time "theorem."