Date: 26 June 1998
To: J3
From: Van Snyder
Subject: A single mechanism to support C TYPEDEF and enumerations
References: 98-113, 98-145r2, 98-157?, 98-165r1

1 Introduction

We propose here similar mechanisms to unify at least these features, as extensions of type declaration:

- A functionality similar to the `TYPEDEF` statement in C was advocated in paper 98-165r1, *Interoperability Syntax (part 1)*. This has substantially broader applicability than supporting interoperability with C.
- Enumeration types were apparently proposed in a tutorial during meeting 145, but I was not able to attend, and have not yet seen the slides (paper 98-157, *Slides from ENUM tutorial*, has not been posted to the server). Enumeration types were presumably advocated to support interoperability with C, but have substantially broader applicability.

No new statement keywords are advocated. Two new intrinsic functions, and expanded interpretation of an existing one, are advocated. It is also advocated to re-examine whether function result types should participate in generic dis-ambiguation. This would also be necessary to support the functionality of the simplified syntax advocated in paper 98-113, *Constants for Opaque Data Types*.

2 Syntax of type redefinition

To provide functionality similar to the `TYPEDEF` statement in C, the `TYPE` statement is extended:

type-definition-stmt **is** TYPE [,*access-spec*] :: *type-name* => *type-spec*

Notice that "::" is not optional, just as it is not optional in the case of initializing a pointer object by using "=> `NULL()`" in a *type-declaration-stmt*.

If *type-spec* refers to a parameterized type (intrinsic, derived, or redefined), parameters may be specified, or *deferred* by omitting all of them or using ":" for any parameter value. If any parameter is deferred in *type-spec*, the redefined type is a parameterized type; the "dummy" parameters of the redefined type occur in the same order, and have the same names, types, kinds, and other characteristics as the deferred parameters in *type-spec*.

It is not advocated to allow a list of *type-name* => *type-spec*, as doing so would make it more difficult for future extensions to allow attaching attributes to the *type-spec*, e.g.

```
TYPE :: A_REAL => REAL, ASYNCHRONOUS
```

3 Syntax of enumeration type definition

To declare enumeration types and their literals, the `TYPE` statement is extended:

type-definition-stmt	**is** TYPE [,*access-spec*] :: *type-name* => *literals*
literals	**is** ORDERED [*kind-selector*] ■ ■ (*named-constant-list*) **or** UNORDERED [*kind-selector*] (*enum-list*)
enum	**is** *named-constant* ■ ■ [= *scalar-int-initialization-expr*]

Values of enumeration types are represented by integers. Also see section 6.

If *kind-selector* is not specified, the kind of integer used to represent ordered enumerations, or unordered enumerations for which no *scalar-int-initialization-expr* is provided, is separately selected for each enumeration type by the processor.

If *kind-selector* is not specified and a *scalar-int-initialization-expr* is specified, the kind of the representation is the kind of the *scalar-int-initialization-expr*. If more than one *scalar-int-initialization-expr* is specified, they must all have the same kind. If *kind-selector* is specified, the kind of every *scalar-int-initialization-expr* must be the kind specified by *kind-selector*.

Notice that "::" is not optional, just as it is not optional in the case of initializing a pointer object by using "=> `NULL()`" in a *type-declaration-stmt*.

4 Syntax of reference to user-defined types

User-defined types – derived types, type redefinitions, or enumeration types – can be referenced by using `TYPE(`*`type-name`*`[(`*`type-parameters`*`)])`.

If a user-defined name is not identical to an intrinsic type, or attribute name, or ORDERED or UNORDERED, it would not be ambiguous to allow a simplified syntax consisting of the type name alone, so long as :: is present, or on the right-hand-side of => in a *type-definition-stmt*.

For example, if one defines `TYPE :: REWIND => LOGICAL`, the redefined type REWIND could be accessed, without change to existing syntax rules by using `TYPE(REWIND) I`. If it were agreed to allow a simplified syntax of usage, it would not be ambiguous to allow `REWIND :: I`. The latter would be ambiguous without the :: symbol. If one defines `TYPE :: REAL => TYPE(EXTENDED(49))` then `REAL :: X` would be ambiguous.

Should the simplified syntax be allowed at this time? (I think it's too much of a mess to describe and to understand the exceptions to be worth the trouble.) *Straw Vote*

5 Semantics of type redefinition

A *type-definition-stmt* introduces a new type that is an extension of the type it redefines (in the same sense as an extensible type). It is not a textual substitution that results in a reference to an existing type.

A re-defined type may be used to dis-ambiguate generic procedures, even if it resolves to the same type as a corresponding argument. In this way, one could be confident that the following is portable (yes, I know this isn't the recommended style, but users seem to want it):

```
TYPE :: SP => REAL(SELECTED_REAL_KIND(6,10))
 TYPE :: DP => REAL(SELECTED_REAL_KIND(13,10))
 INTERFACE SUB
   SUBROUTINE SUB_SP ( ARG )
     TYPE(SP) :: ARG
```

```
  END SUBROUTINE SUB_SP
  SUBROUTINE SUB_DP ( ARG )
    TYPE(DP) :: ARG
  END SUBROUTINE SUB_DP
END INTERFACE
```

A redefined type inherits existing values and properties from the type from which it is defined. (This makes programs less mutable than I desire, but I see no alternative other than explicit casting, which would be onerous in most cases.)

For example, the following is legal:

```
TYPE :: REWIND => LOGICAL
TYPE(REWIND) :: I => .false.
IF (I) REWIND 10             ! "IF (I)" is OK, because I's underlying
                             ! type is LOGICAL
```

Defined operations are inherited by redefined types, unless over-ridden (over-riding is not possible if redefined types are not specified to be new types, and may be undesirable in any case). Inheriting the defined operations is very useful, for example, if one defines

```
TYPE :: MYREAL => <REAL or DOUBLE PRECISION or TYPE(EXTENDED(49))>
TYPE(MYREAL) :: X, Y, Z
X = Y + Z ! Should use intrinsic assignment and addition if MYREAL
          ! resolves to REAL or DOUBLE PRECISION, and defined assignment
          ! and addition if it resolves to TYPE(EXTENDED(49)).  Similar
          ! arguments apply to -, *, /, **, SIN(), ATAN2() ...
```

It should probably not be allowed to define intrinsic operations, e.g. +, to operate on redefined types that resolve to intrinsic types. Thus, one couldn't count on the following continuing to work if TYPE(MYREAL) were changed from TYPE(EXTENDED(49)) to REAL:

```
INTERFACE OPERATOR(+)
  TYPE(MYREAL) FUNCTION MYREAL_PLUS ( A, B )
    TYPE(MYREAL), INTENT(IN) :: A, B
  END FUNCTION MYREAL_PLUS
END INTERFACE
```

It should be remarked as a note that defined operator arguments (or at least arguments of redefinitions of intrinsic operator symbols) should be declared using intrinsic or derived types, rather than redefined types, whenever possible, to avoid the problem illustrated above.

Type redefinition clearly has much similarity to type extension and inheritance. This mechanism would not be needed if the type inheritance facility allowed to extend intrinsic types, with the restriction that no new components could be added, and the component name that is the same as the parent type did not exist. The same restrictions would apply to types extended from extensions of intrinsic types. On the other hand, the syntax proposed here is more terse than

```
TYPE, EXTENDS(<REAL or DOUBLE PRECISION or EXTENDED(49)>) :: MYREAL
END TYPE MYREAL
```

If this facility is needed, should it be provided as described here, or by enhancing the type inheritance mechanism described in paper 98-145r2 and its antecedents? *Straw Vote*

6 Semantics of enumeration type values

The intrinsic function INT may be used to retrieve the numeric representation of an enumeration literal. The intrinsic function KIND may be applied to the result of applying INT to a value of enumeration type to determine the kind of integer used to represent values of the type.

Should KIND be directly applicable to values of enumeration types? *Straw Vote*

In the case of ordered enumerations, or of unordered enumerations in which no explicit value is provided for the *k*'th literal, the first literal is represented by zero, and the *k*'th literal is represented by 1 + INT(*k-1*'th literal).

It is possible for two literals of an unordered enumeration type to have the same representation.

The only intrinsic operations defined on values of unordered enumeration types are assignment (=), equality (.EQ. or ==), and inequality (.NE. or /=).

Additional features of ordered enumerations

- All relational operators are defined on variables and literals of ordered enumeration types.
- Values of ordered enumeration types may be used in SELECT CASE constructs and DO constructs.
- TINY and HUGE are defined for ordered enumeration types, and return the first and last literal of the type, respectively (not an integer). Thus if one has a variable `E` of an ordered enumeration type, it is permitted to write `DO E = TINY(E), HUGE(E)`, to use TINY(E) and HUGE(E) for array dimensions, etc.
- Values of ordered enumeration types may be used in array dimensions and subscripts. If an array has a dimension bound given by a value of an ordered enumeration type, the other bound of that dimension must be of the same type, or omitted (in which case it is taken to be TINY or HUGE, as appropriate), and a subscript for that dimension must be of the same type as the bound. This insures that subscripts stay within bounds.
- A constructor having the same name as the type is defined. It takes a single integer as an argument and returns a value of the enumeration type. (It would be cool be able to raise an exception if an out-of-range argument is used.) One can guard against an out-of-range argument by writing, e.g.

```
IF ( I >= TINY(E) .AND. I <= HUGE(E) ) E = <type-of-E>(I)
```

- Two additional intrinsic functions are defined, say `SUCC` and `PRED` (spelling negotiable) that return the successor and predecessor of a variable or literal of an ordered enumeration type. The result is the same type as the argument, not an integer.

  Should SUCC(*last-literal*) be an error, or *first-literal*? If it's an error, it would be cool to be able to raise an exception. The obvious anti-symmetric question applies to PRED. One can guard against the error similarly to guarding against the error in the constructor. *Straw Vote*

  `SUCC(E)` ≡ `<type-of-E>(1+INT(E))` or `<type-of-E>(MOD(1+INT(E),1+HUGE(E)))`. *Straw Vote*

  Should SUCC and PRED be provided?

6.1 Suggestion for writing standardese

Introduce Discrete and Continuous classifications of types. Types INTEGER, LOGICAL and enumeration types are discrete; REAL and COMPLEX are continuous. Derived types are neither.

Introduce the term *defined type* to include derived types, redefined types, and enumeration types. This allows to prohibit, all at once, that none may appear in COMMON or EQUIVALENCE.

7 Allowing function result type to participate in generic dis-ambiguation

At present, the result of a function does not participate in generic dis-ambiguation.

If the result type were used in generic dis-ambiguation, two benefits would accrue:

- It becomes possibile to use a character value to invoke a value constructor for a derived type (opaque or otherwise), as advocated in paper 98-113.
- If one considers a literal of an enumeration type to be a pure function having zero arguments, it becomes possible to allow different enumeration types to have literals having the same name. E.g., the following type definitions can coexist:

  ```
  TYPE :: COLOR => ORDERED(RED, GREEN, BLUE)
  TYPE :: NAMES => ORDERED(WHITE, BROWN, BLACK, GREEN, BLUE)
  ```

  (Yes, I know the usual syntax to reference argument-less functions includes an empty actual argument list in parentheses. What is proposed here is only a sophistry to simplify explanation, not a definition or implementation.)

It is not an open problem to decide the dis-ambiguation. It has been explained in the Ada-83 standard (ANSI/MIL-STD-1815A), and in numerous books on functional programming (e.g. Anthony J. Field and Peter G. Harrison, **Functional Programming**, Addison-Wesley (1988) Chapter 7).

Including the function result type in the criteria to dis-ambiguate generic function references would not invalidate any program that conforms to the Fortran 95 standard, if doing so is used only as a last resort when the arguments are insufficient.