Date: 5 August 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, 98-178

1 Introduction

We propose here three mechanisms to provide functionality similar to `TYPEDEF` declarations in C, which 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. A mechanism to declare enumeration types is proposed in this paper, rather than in a separate paper, because it is similar to the second mechanism proposed for type redefinition.

All mechanisms proposed herein are proposed as extensions of type declaration.

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

1.1 Summary of methods of type redefinition

The first method of type redefinition expands extensible types by allowing to extend intrinsic types. The advantages are that no new mechanism is necessary – just an expansion of type extensibility – and redefined types define new types, which can therefore be used for generic resolution. The disadvantage is that special cases, clumsy circumlocutions, or additional new mechanisms are necessary to make obviously desirable things work. Some of the new mechanisms have already been proposed for other purposes.

The second method introduces a new mechanism by which type synonyms, not new types, are defined. The advantage is that obviously desirable things work. The disadvantages are that a new mechanism is necessary, and type synonyms cannot be used for generic resolution if they are synonyms for the same "real" type.

The third method – the simplest one – was recently proposed to me informally by Malcolm Cohen, and perhaps more formally at an earlier time. It allows to put `TYPE()` around intrinsic types, and then rename them during USE association. There is the slight irregularity that the "constructors" `REAL` and `LOGICAL` exist, while the "constructors" `INTEGER`, `COMPLEX` and `CHARACTER` are unfortunately spelled `INT`, `CMPLX` and `CHAR`. This could be repaired simply by defining new intrinsic "constructors" `INTEGER`, `COMPLEX` and `CHARACTER` with obvious definitions. This method has the advantage of simplicity (so much so that nothing more is said here), and the disadvantage of requiring construction of a module for the purpose of renaming a type.

2 First method of type redefinition

The first method of type redefinition expands the applicability of type extension, by allowing to extend intrinsic types. This method requires no changes to existing syntax.

Types defined in this way are new types, as are all other extended types.

No new restrictions appear to be necessary, from the viewpoint of linguistic consistency, but some may be desirable from the viewpoint of implementability. Prohibiting polymorphic objects of a class that begins at an intrinsic type or an extension of one complicates inheritance of intrinsic procedures defined on intrinsic types. There seems to be no need to prohibit additional components in extensions of intrinsic types, whereas prohibiting them would introduce an irregularity. Intrinsic operations on extensions of intrinsic types could be defined to be non-overridable, but this, too, may be unnecessary (perhaps even undesirable).

We propose here that a type that is an extension of a parameterized type is a parameterized type. If the base type is an intrinsic type, for which a default parameterization exists, use of the new type in an object declaration without parameterization results in its use with the default parameterization; use to define another type defines another parameterized type. Thus, the following are legal

```
TYPE, EXTENDS(REAL) :: MYREAL; END TYPE MYREAL ! A parameterized type
TYPE(MYREAL(KIND(0.0D0))) :: MYVAR1  ! Double precision real
TYPE(MYREAL) :: MYVAR2               ! Single precision real
```

It would be convenient if an extended type inherits existing properties from the type from which it is defined, and values without explicit construction, so that the following would be legal:

```
TYPE, EXTENDS(LOGICAL) :: REWIND; END TYPE REWIND
TYPE(REWIND) :: I = .false.
IF (I) REWIND 10
```

The second statement is, at present, opposite to the rules of extensible type assignment. One might remedy the problem by allowing assignment of an object of parent type to an object of descendant type if the descendant type adds no new components. The alternative is to require an explicit constructor, so that the second statement above becomes

```
TYPE(REWIND) :: I = REWIND(.false.)
```

The third statement is at present completely undefined because the possibility of extending intrinsic types has not yet been pondered. By the methods explained in section 6 it may be possible automatically to apply an anonymous "conversion constructor" to convert `I` from `TYPE(REWIND)` to `LOGICAL`, and, hopefully, expect the "conversion constructor" to come into existence automatically if the extension has no additional components and no constructor has been defined.

Defined operations are inherited by redefined types, unless over-ridden. Inheriting the defined operations is very useful, for example, if one defines

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

Intrinsic procedures defined on intrinsic types should be "inherited" by extensions of intrinsic types. The syntax isn't right to consider the intrinsic procedures to be type-bound procedures, e.g. we don't write X%SQRT. An alternative is to define the intrinsic procedures to have polymorphic dummy arguments.

If it were prohibited to define intrinsic operations, e.g. +, to operate on types that extend intrinsic types, one couldn't count on the following continuing to work if `TYPE(MYREAL)` were changed from `TYPE(EXTENDED(49))` to `REAL`:

```
INTERFACE OPERATOR(+)
  MODULE PROCEDURE MYREAL_PLUS
END INTERFACE
CONTAINS
  TYPE(MYREAL) FUNCTION MYREAL_PLUS ( A, B )
    TYPE(MYREAL), INTENT(IN) :: A, B; ...
  END FUNCTION MYREAL_PLUS
```

To avoid this problem, procedures should be defined in terms of base types, while objects are defined in terms of extended types. The correct procedure will be chosen by the usual generic resolution rules.

3 Second method 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 *type-name* occur in the same order, and have the same names, kinds, and other characteristics as the deferred parameters in *type-spec*. If *type-spec* refers to a parameterized type for which a default parameterization exists, e.g. `REAL`, the redefined name denotes the default parameterization when used to declare an object, and the unspecialized parameterized type when used to define another type.

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
```

This mechanism of type redefinition is a "macro substitution" that does not introduce a new type. It simply introduces a synonym for an existing type, perhaps indirectly by way of a previously defined synonym. The "real" type in terms of which a synonym is defined, not the new synonym or a previously existing one, is used for generic resolution.

A redefined type cannot be used to resolve generic procedures if it is a synonum for the type of a corresponding argument. Thus one could not be confident that the following is portable:

```
MODULE...
  TYPE :: SP => REAL(SELECTED_REAL_KIND(6,10))
```

```
  TYPE :: DP => REAL(SELECTED_REAL_KIND(13,10))
  INTERFACE SUB
    MODULE PROCEDURE SUB_SP, SUB_DP
  END INTERFACE
CONTAINS
  SUBROUTINE SUB_SP ( ARG )
    TYPE(SP) :: ARG
  END SUBROUTINE SUB_SP
  SUBROUTINE SUB_DP ( ARG )
    TYPE(DP) :: ARG
  END SUBROUTINE SUB_DP
END MODULE ...
```

4 Enumeration types

To declare enumeration types and their literals, the `TYPE` statement is extended by a syntax similar to the second one proposed for type redefinition:

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

Values of enumeration types are represented by integers.

If BIND(C) is specified, C representational rules apply, *kind-selector* is not allowed, and *scalar-int-initialization-expr*, if any, shall have default integer kind. Is it better to ignore the kind? *J3 note*

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 shall all have the same kind. If *kind-selector* is specified, the kind of every *scalar-int-initialization-expr* shall 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*.

The intrinsic function `INT` may be used to retrieve the numeric representation of an enumeration literal. 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 `SIZE`(*k-1*'th literal) + `INT`(*k-1*'th literal).

For unordered enumerations, or for ordered enumerations for which *explicit-shape-spec* is not specified, the size is one.

If *explicit-shape-spec* is specified for an ordered enumeration, the size must be positive. If `E` is an enumeration literal with bounds e_1:e_2, `E`(e_1) denotes the first value, etc., `E` and `E`($k : l$) are sequences of values of the type of `E`, and `INT(E)` and `INT(E(`$k : l$`))` are sequences of integers.

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

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 (exept maybe not for `BIND(C)` enumerations).

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

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 numeric relational operators are defined on values of ordered enumeration types.
- Values of ordered enumeration types may be used in `SELECT CASE` constructs and scalar ones may be used in `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.
- Scalar values of ordered enumeration types may be used in array dimensions and scalar or array ones may be used in subscripts. If an array has a dimension bound given by a value of an ordered enumeration type, the other bound of that dimension shall 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 shall be of the same type as the bound. An omitted lower or upper bound of a subscript triplet is taken to be `TINY` or `HUGE`, respectively. An increment of a subscript triplet is an integer. Should increments of subscript triplets of enumeration types be prohibited? *Straw vote*
- An elemental constructor having the same name as the type is defined. It takes a single integer argument and returns a value of the enumeration type. (It would be cool to 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 >= INT(TINY(E)) .AND. I <= INT(HUGE(E)) ) E = <type-of-E>(I)
```

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

  Should `SUCC` and `PRED` be provided? *Straw Vote*

  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.

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

5 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*

5.1 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. Or, INTEGER is an ordered enumeration, and LOGICAL is an unordered enumeration (except it's allowed in SELECT CASE).

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.

6 Allowing function result type to participate in generic resolution

At present, the result of a function does not participate in generic resolution. Including the function result type in the criteria to resolve 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.

If the result type were used in generic resolution, at least two presently impossible things become possible (one can debate whether these are benefits):

- It would be possible in many cases to apply automatically what in C++ are called "conversion constructors."

  As a special case, It becomes possible to use a character value to invoke a value constructor for a derived type (opaque or otherwise), as advocated in paper 98-113.

- Generic function references that have identical actual argument characteristics, and that therefore cannot now be resolved, could sometimes be resolved.

  As a special case, 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 => UNORDERED(RED, GREEN, BLUE)
```

```
TYPE :: NAMES => UNORDERED(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 would be useful to extend to user-defined types what is already done automatically for intrinsic types: When different types or kinds of numbers are combined, there are rules that specify which "conversion constructor" to invoke, and thence which "operator function" to invoke. For example, in `0.5d0 + 3` the REAL "conversion constructor" with KIND = KIND(0.5d0) is applied to `3` and then the double precision real `add` operator function is invoked. A "conversion constructor" is a function that has the same generic name as a type, and has one `INTENT(IN)` argument (except that `INTEGER` and `COMPLEX` are spelled `INT` and `CMPLX`, respectively). An intrinsic conversion function with a `KIND` argument is considered to have only one argument; the `KIND` argument is considered to be an indication of the result type's kind. To be consistent with parameterized derived type constructors, intrinsic constructors (`REAL`, `INT`, etc.) should be changed to allow the parameterization in the same syntactic position. Thus `REAL(3,KIND(0.5d0))` could alternatively be written `REAL(KIND(0.5d0))(3)`. If explicitly typed expressions were allowed (see paper 98-178), and if the result type could participate in resolving generic function references, and if it were possible to apply conversion constructors automatically, the above expression could be equivalently written `0.5d0 + REAL(KIND(0.5d0)) :: REAL(3)` and `0.5d0 + REAL(KIND(0.5d0)) :: 3`.

It is not an open problem to decide the resolution. 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). Here's an attempt at an initial superficial explanation. Maybe it works.

In the sequel, *operand* and *actual argument* are interchangable, and *operator*, *function*, *operation* and *subroutine* are interchangable. For example, in $a = b + c$, the operation "=" may be implemented by a subroutine, the operator "+" may be implemented by a function, and the operands a, b and c may be actual arguments.

To resolve a generic reference, one analyzes a statement from the "outside inward." If one had an attributed parse tree or equivalent representation, one would analyze it from the the root toward the leaves, starting with the set of all possible operators or operations that the root of the tree might represent.

1. For each dummy argument position, construct the set of all possible characteristics – the union of dummy argument types for all possible operations.

2. For each actual argument, construct the set of all possible characteristics.

3. For each argument position, intersect the set of possible dummy argument characteristics with the set of possible actual argument characteristics.

4. Remove each operation for which the set of possible actual argument types does not include the dummy argument type, for any argument position. This will remove operator-operand possibilities that do not have the correct number of arguments.

5. Remove each operand type that is not in the set of possible dummy argument types, at each position. Repeat the previous step and this one if either one removes anything.

6. Repeat this process for every actual argument that is a function result.

If anything changes during an iteration, it is repeated starting at the root. The iteration may converge faster if it is repeated alternately starting from the root and leaves, or if started initially from the leaves. When a fixed point is reached, if there is any operand for which there are zero possible interpretations of its type, apply all visible "conversion constructors" to the set of original types of the operand, and start over again. Be careful of introducing a loop in this step. E.g. don't apply `INT(REAL(INT(REAL...)))`.

Ultimately, there are three possibilities:

- At least one operation, or the type of at least one operand has two possible interpretations, and no operations or operand types have zero possible interpretations. The program is ambiguous. If one of the interpretations is correct, the program can be repaired by providing an explicit conversion or type mark (see paper 98-178).
- Every operation, and the type of every operand, has exactly one possible interpretation. The program is correct as it stands.
- At least one operation, or the type of at least one operand, has zero possible interpretations. The program is incorrect.