Subject: Wish List for Fortran after 2003
From: Van Snyder
References: 97-114r2, 98-171r2

Herein, I list my wishes for future revisions of the Fortran standard. Other than being divided into *important* and *merely useful* groups, the discussion is not in any particular order. There's also a section on things that would be useful to standardize, but that one shouldn't expect to be universally available. As such, these ought to be separate optional parts of the standard.

Contents

1 Important

The changes advocated in this section would have significant positive impact on the ability to produce readable, modifiable, correct and efficient software.

1.1 One-sided Coroutines

Introduction

Several categories of algorithms of mathematical software require access to code provided by the user of the algorithm. Examples include quadrature, differential equations, minimization, nonlinear least-squares, zero-finding, ... The most common way to access this code is for the user to provide a procedure as an argument. Another common way is for the algorithm to invoke a procedure having a specific name to access user-provided code. These methods work in the simple cases.

In more complex cases, the user-provided code needs access to additional data about which the general-purpose mathematical software package is not aware, or it uses a user-defined type to represent an abstract mathematical object, such as a (sparse) matrix. If the only way for the package to access user code is to call a procedure, one possibility is to use common or module variables. If the mathematical software package has a dummy argument of extensible type that is passed to the user code, the type could be extended. If the additional data come from numerous sources, it may be impossible to put them all into a single derived type, in which case the extension will consist mostly of pointers, which will have a negative impact on performance.

To provide for passing extra information to the user's code, or operations using a user-defined type, some packages provide a mechanism known as *reverse communication*, in which the package returns to the user with an indication that a certain calculation is to be done. The user then calls the package again, and it continues from where it left off.

From the user's point of view, this is not unduly complex: One provides an initial value for a "flag" that is usually an argument of the package. Then there is a loop, in which one calls the procedure and then tests the value of the flag to determine what calculation the package requires. One or more of the values indicate that the process is finished (perhaps abnormally).

From the package's point of view, this is a terrible mess. One needs to keep track of what process was in progress when access to user-provided code was necessary, and somehow resume that process. This usually requires violating integrity of conventional control structures, such as DO, IF and CASE constructs. This in turn leads to using GOTO instead, which in turn leads to code that is expensive to develop, augment, maintain, and gain confidence in its correctness.

The control strategy that the computational mathematics community calls reverse communication has long been known in the language design community as a *coroutine*. In its most general form, it is more general than what has been described here, in that the relation is symmetric.

Coroutines are also useful to define *iterators*. An iterator is usually used to enumerate the elements of a data structure, to control the iteration of a loop. For example:

```
call GraphIter ( Vertex, G )
do while ( associated(Vertex) )
  ...
  resume GraphIter ( Vertex, G )
end do
```

There is already in Fortran an example of a coroutine control structure: The relation between an input/output list and a FORMAT statement. So coroutines aren't really a radically new concept for Fortran.

Proposal

Provide a simple form of one-sided coroutine, which is adequate to allow developing iterators and packages that provide for reverse communication, without causing the excruciating mess within the package that is presently required.

Coroutines could be supported by adding two statements, e.g., SUSPEND and RESUME. If a procedure has never been entered, or it was last exited by a RETURN statement, or it is entered by a CALL statement, control transfers to its first executable statement. Otherwise, the procedure was previously entered, was last exited by a SUSPEND statement, and is being re-entered by a RESUME statment, in which case control resumes at the next executable statement after the SUSPEND statement. This is summarized in the following table:

	Enter by CALL	Enter by RESUME
No previous entry	First executable statement	First executable statement
Previous exit by RETURN	First executable statement	First executable statement
Previous exit by SUSPEND	First executable statement	First executable statement after the last executed SUSPEND statement

Unsaved variables don't become undefined when a SUSPEND statement is executed, so the standard's present words about "when a RETURN or END statement is executed" are adequate.

Recursive coroutines may be useful, e.g. for multidimensional quadrature, but the marginal utility of having them may not be worth the mess of describing them, and the burden of implementing them.

In their most general form, coroutines can be mutually recursive. I propose to prohibit recursive (and therefore also mutually recursive) coroutines. That is, a RESUME statement shall not refer to a recursive procedure, and a SUSPEND statement shall not appear within one.

1.2 "Physical" or "Engineering" Units

Introduction

Incorrect use of physical units is the second-most-common error in scientific or engineering software, coming immediately after mismatching the types of actual and dummy arguments. Explicit interfaces largely solve the second problem, but do nothing directly for the first. (One can use derived types to provide a physical units system, at the expense of redefining all of the intrinsic functions, operations, and assignment.) A particularly expensive and embarrassing example was the loss of the Mars Climate Orbiter. The loss resulted because the NASA contract required small forces, e.g. from attitude-control maneuvers, to be reported in Newton-Seconds, but Lockheed nonetheless reported them in Pound-Seconds. (This was quite inscrutable, as Lockheed had had contracts with JPL for over thirty years, and they've *always* specified use of the metric system.)

Proposal

Define a new UNIT attribute or type parameter (call it what you will) that can be specified for any numeric variable. Constants are unitless. Define multiplication and division operations on units. Exponentiation by an integer constant could be defined to be equivalent to multiplication or division. In the context of a unit definition, the integer constant 1 is considered to be the unitless unit.

Each unit declaration creates a *units conversion function* having the same name as the unit, that takes an argument with any units, and "converts" it to have units specified by the name of the function. There is an intrinsic UNITLESS conversion function.

Quantities can be added, subtracted, assigned, compared by relational operators, or argument associated only if they have equivalent units. Atomic units, i.e. units that are not defined in terms of other units, are equivalent by name. Other units are equivalent by structure.

When quantities are added or subtracted, the units of the result are the same as the units of the operands. When quantities are multiplied or divided, the units of the result are the units that result from applying the operation to the operands' units. Multiplication or division by a unitless operand produces a result having the same units as the other operand.

Units participate in generic resolution.

Procedure arguments and function results can have *abstract* units. This allows enforcing a particular relation between the units, without requiring particular units. For example, the SQRT intrinsic function result has abstract units A, and its argument has abstract units A*A. Abstract units do not participate in generic resolution.

Define an intrinsic RADIAN unit, and a parallel set of generic intrinsic trigonometric functions that take RADIAN arguments and produce unitless results. All of the remaining intrinsic procedures have arguments with abstract units and results that are unitless (e.g. SELECTED_INT_KIND) or have the same units as their argument (e.g. TINY). Because function results do not participate in generic resolution, it is not possible to have a parallel set of generic intrinsic inverse trigonometric functions that return RADIAN results. It may be useful to provide an intrinsic module that has some public units and procedures, e.g. units TICK and SECOND and a SYSTEM_CLOCK module procedure that has arguments with units TICK, TICK/SECOND and SECOND.

Variables are declared to have units by specifying UNIT(*unit-name*) in their declarations, or **unit-name* after their names in declaration statements.

Examples:

```
UNIT :: INCH, SECOND
UNIT :: CM, INCH_TO_CM = CM / INCH
REAL, PARAMETER, UNIT(INCH_TO_CM) :: CONVERT = INCH_TO_CM(2.54)
UNIT :: SQINCH = INCH * INCH ! or INCH ** 2
UNIT :: IPS = INCH / SECOND, FREQUENCY = 1 / SECOND ! or SECOND ** (-1)
REAL, UNIT(SQINCH) :: A
REAL, UNIT(FREQUENCY) :: F
REAL, UNIT(INCH) :: L, L2, C*CM
REAL, UNIT(SECOND) :: T
REAL, UNIT(IPS) :: V

V = A + L                  ! INVALID -- SQINCH cannot be added to INCH,
                           ! and neither one can be assigned to IPS
V = IPS(A + SQINCH(L))     ! VALID -- I'm screwing this up intentionally
V = (A / L + L2) / T       ! VALID -- IPS is compatible with INCH / SECOND
A = L * L2                 ! VALID -- SQINCH is compatible with INCH * INCH
F = V / L                  ! VALID -- units of RHS are 1/SECOND
C = CONVERT * L            ! VALID -- CM / INCH * INCH = CM
```

```
L = SQRT(A) * 5.0e-3         ! VALID -- exercise for reader
```

Add an optional U[w] suffix to numeric format descriptors, that causes units to be output by write statements, or input and checked by read statements. The problem with the Mars Climate Orbiter was that quantities with the wrong units were written by one program, and assumed to be the correct units by another program. For list-directed and namelist input, units must be supplied; for output, they are produced.

A useful extension is to allow units to be defined in terms of others using expressions of the form $U = aU' + b$ where U and U' are unit names, and a and b are initialization expressions. For example

```
UNIT :: MHz, GHz = 1000 * MHz, KHz = 0.001 * MHz
UNIT :: C, F = 1.8 * C + 32.0
REAL, UNIT(MHz) :: Frq
REAL, UNIT(C) :: Temp
...
READ(*,*) Frq, Temp
```

Assuming the input is

```
3.31 GHz 212 F
```

the variables `Frq` and `Temp` get the values 3310.0 MHz and 100.0 C respectively. This extension would have let Lockheed use whatever units they wanted to use, so long as the JPL software had the unit name, and the appropriate conversion, available.

1.3 Improvements in the type system

1.3.1 A more complete type system should be provided

Enumerations and enumerators

Enumerations that are new types should be provided. One proposition is outlined in J3/98-171r2. If generic resolution is simultaneously changed so that function result types participate (as Ada has done since 1983), and enumerators are considered to be references to zero-argument functions, we can allow enumerators of different types to have the same names.

Subranges of integers

Unsigned integers, and integers that have nondecimal range, are frequent requests. These requests could be satisfied, and numerous other benefits provided as well, by providing for named subranges of integers.

One of the additional benefits is nearly-free checking of subscript bounds: If the bounds of an array are specified by a subrange name, and if in that case the only subscripts allowed are those of that subrange, one only need check when the subscript gets a value, not when it's used as a subscript. If DO control is defined by reference to the subrange, say by using TINY and HUGE, even that check can frequently be avoided.

Another benefit is that one can get most (or perhaps all) of the desired functionality of a BIT type by specifying a subrange of 0:1.

1.3.2 New types – not type aliases – from existing types

It is useful to be able to create new types – not type aliases – from existing types. In particular, it should be possible to create a nonparameterized derived type by specializing kind type parameters of a parameterized derived type.

1.3.3 Regularize CHARACTER type

There are some irregularities with CHARACTER type that make the standard larger than necessary, and make use of character variables more difficult than necessary. Restrictions on use of character variables should be removed, to make their use more like array operations:

1. Allow a single expression to select a substring of length 1, i.e., *expr* instead of *expr*:*expr*,
2. Allow substring indices to begin other than at 1, and
3. Allow a stride specification in a substring selector.

The discussion of character substring selection could then largely refer to the description on array section selection, thereby shortening the standard.

1.4 Updaters

When a data structure is implemented by a procedure, the usual way to get values from it is by way of a function reference. The usual way to put values into it is by way of a subroutine call. For particularly simple data structures, i.e., scalars, arrays and structures, subroutine calls aren't needed. Otherwise, the situation isn't symmetric: A function reference can be used in any value-producing context, but a subroutine call can't be used in any variable-definition context.

A few obscure languages provided a form of subprogram useful in this respect: An *updater*. An updater is a function in reverse: When it is invoked, it *receives* a value.

Updaters could be provided in Fortran in at least three ways.

One is with a new form of interface block, with a subroutine that has restrictions on its interface in the same spirit as for a subroutine used for defined assignment. Something like

```
UPDATER [ generic-name ]
  procedure MY_UPDATER
END UPDATER
```

Another possibility is a new kind of procedure, beginning something like this:

```
prefix UPDATER updater-name ( [ dummy-arg-name-list ] ) [ &
  &  RECEIVE ( receive-name ) ]
```

The rest of it is like any other procedure. The *receive-name* behaves like an INTENT(IN) dummy argument (it's the same as the *updater-name* if it's not specified). It gets associated with the object being "sent" to the updater – like the RHS of an assignment statement in which the updater reference is the LHS. Neither this way nor the previous way of providing updaters guarantees that there is a function that has the same characteristics as the updater.

A third possibility is a different new kind of procedure, that provides a function and its companion updater in a single program unit, with a guarantee that the function and updater have the same characteristics. It could be something like this:

```
prefix ACCESSOR accessor-name ( [ dummy-arg-name-list ] ) [ &
  & TRANSFER (  transfer-name ) ]
  specification-part
  WHEN PROVIDE
```

```
      ! function part
    WHEN RECEIVE
      ! updater part
  END ACCESSOR accessor-name
```

The *transfer-name* behaves like an INTENT(OUT) argument in the PROVIDE part (like the *result-name* does in a function) and like an INTENT(IN) argument in the RECEIVE part. If it's not provided, the *accessor-name* is used.

In any case, generic updaters (or accessors) should be allowed, and if updaters and functions are separate the generic should be allowed to include both updaters and functions.

Where a reference appears in a value-producing context, where functions appear now, a generic is resolved to a function, or control arrives in an accessor in the PROVIDE branch.

Where a reference appears in a variable-definition context, a generic is resolved to an updater, or control arrives in an accessor in the RECEIVE branch.

Where a reference to an accessor or to a generic that resolves to both a function and an updater appears as an actual argument that is associated with a dummy argument that has INTENT(INOUT) or unspecified intent, copy in / copy out semantics are used.

1.5 Structured parallelism would assist to exploit modern architectures

Define a new execution construct:

parallel-construct	**is**	[*parallel-construct-name* :] PARALLEL
		    fork
		    [*fork*] ...
		END PARALLEL [*parallel-construct-name*]

fork	**is**	FORK [*parallel-construct-name*]
		    execution-construct
		    [*execution-construct*] ...

Upon executing a PARALLEL statement the processor may divide the sequence of execution into a number of sequences not exceeding the number of FORK blocks. Each of these sequences begins execution at the start of a different FORK block; after finishing execution of one FORK block a sequence may continue into a different one, or may continue by executing the END PARALLEL statement. In any case, each FORK block is executed exactly once. After all of the FORK blocks are executed, all sequences of execution that were created by execution of the PARALLEL statement are condensed into a single sequence, and execution proceeds at the first statement after the END PARALLEL statement. Notice that this definition permits the processor to rearrange the forks into an arbitrary order, and then ignore the PARALLEL, FORK, and END PARALLEL statements.

A GO TO statement or an arithmetic IF statement within a parallel construct shall not have a branch target outwith that parallel construct or in a different fork block, and vice-versa. A RETURN statement shall not appear within a parallel construct. This requirement avoids having to answer the question “So, how long does the activation record exist?”.

A variable shall have the ASYNCHRONOUS attribute if it or a subobject of it or a variable associated with it or a subobject of a variable associated with it may be accessed within more than one FORK block. The ASYNCHRONOUS attribute is not automatically deduced so as not to impose it when it is not necessary. Consider the following example of double-buffered input overlapped with processing. Even though references to the BUF variable appear in both FORK blocks, it should not have the ASYNCHRONOUS attribute because no element of BUF can be accessed simultaneously in two FORK blocks.

```
  type(myInput) :: BUF(blockSize,0:1)
  integer :: STAT, WHICH
...
  which = 0
  read ( inunit, iostat=stat ) buf(:,which)
  do while ( stat == 0 )
    parallel
    fork
      ! Process buf(:,which)
    fork
      ! Read into the side of BUF not being processed
      read ( inunit, iostat = stat ) buf(:,1-which)
    end parallel
    which = 1 - which
  end do
```

Notice that asynchronous input/output doesn't provide all of the functionality that structured parallelism does: It cannot be used with user-defined derived-type input/output procedures. Asynchronous input/output is different in that it's not block structured. This could be viewed as an advantage or a disadvantage.

The number of forks is statically specified. If a dynamically determined number of forks is necessary, use FORALL. The iterations in FORALL are independent but qualitatively identical; in a PARALLEL construct the forks can be qualititively different.

OpenMP provides this and additional functionality. Some of the additional functionality of OpenMP could be provided by allowing constructs to be scoping units (see 2.1.1).

1.6 Include should be user-defined

The quoted text in an include line is presently specified to be interpreted in a processor-defined way. It should be user defined. See 97-114r2.

1.7 Internal procedures as actual arguments

The problem with allowing internal procedures as actual arguments occurs if they are invoked at a time when a different instance of their host environment exists from the one in which they were used as an actual argument. In that case, there are two possibilities for the meaning of host environment – the one at the time the internal procedure was used as an actual argument, and the one at the time the internal procedure is invoked by way of a dummy procedure argument. The former is usually considered to be the more useful one.

There is no problem if there can't be more than one choice for the host environment.

When a procedure is invoked by way of a dummy procedure argument, there can only be more than one choice of the host environment if the internal procedure could be passed to or through a recursive invocation of its host, by way of dummy prodedure arguments, before being invoked. Without procedure pointers, this can be presumed to be impossible if the host procedure does not have a dummy procedure argument, or if all of its dummy procedure arguments have explicit interfaces, and none of them are compatible with the internal procedure. Procedure pointers complicate this argument. This means that the only way that this can be presumed to be impossible is if the host is not recursive.

There are thus three possibilities:

1. Prohibit them;
2. Allow them if they have nonrecursive hosts;
3. Allow them unconditionally;

If they are allowed for recursive hosts, it is important to specify that the host environment is the one at the moment the internal procedure was used as an actual argument.

If we allow internal procedures to be actual arguments, it will be difficult to prohibit them from being the targets of procedure pointers. Internal procedures should be allowed to be targets of procedure pointers under the same conditions that they can be actual arguments. If internal procedures with recursive hosts are allowed to be targets of procedure pointers, the instance of the up-level environment with the procedure is invoked by way of a pointer is the one when the procedure was associated with the pointer.

Procedure pointers could benefit from an attribute that prohibits associating an internal procedure with a pointer that outlives the procedure's up-level environment. It is now too late to do this without introducing an incompatibility with Fortran 2003. On the other hand, we don't do anything like this for variables – we simply say that the pointer association status becomes undefined.

1.8 Provide something similar to C++ templates or Ada generic packages

Introduction

It is useful to develop algorithms that can be applicable to a broad class of data types, not just to different kinds of one type. An example is a sort routine, which can apply to any type for which enough relational operators are defined.

In Fortran 2003, a developer of a type that has a kind type parameter and a type-bound procedure that has a passed-object dummy argument needs to be careful to provide the type-bound procedure as a generic with enough specifics to cover all of the reasonable combinations of the values of the kind type parameters.

Proposal

Ada has a facility called a generic package. From a generic package, one can create a specific one by *instantiation*: specific ones do not spring into existence as necessary, but are rather created by applying specific values to parameters to produce one from the generic "template." In C++, the corresponding facility is called a template.

In Fortran as in Ada, one would need to create specific entities by instantiation. As with USE, it is desirable to be able to instantiate some subset of the entities. If entities from a particular template are instantiated more than once within a scoping unit, it will be necessary to have some mechanism to avoid name clashes. The USE statement has all of the necessary functionality, except for syntax to specify the values of parameters. Therefore, the simplest syntax for instantiation is probably an extension of the existing USE statement. Thereupon, the logical syntax for a template is a parameterized module.

It would be useful to allow templates to appear within other scoping units or templates. If one accesses a template from a module or template, without specifying values for its parameters, it remains a template; it still needs to be instantiated.

If templates are implemented as modules, and are allowed in other scoping units, it would be reasonable to allow modules that aren't templates to be defined within other scoping units, including other modules.

It is useful to allow for partial instantiation, to create a new template with fewer parameters rather than a completely-specified instance. If templates are allowed within templates, and an inner one has parameters that are also parameters of the outer one, when the inner one is gotten by instantiation of the outer one, there may still remain some unspecified parameters of the inner one. It would be therefore be a template, not an instance. So partial instantiation is a natural consequence of nesting.

Having a parameterized template to package a derived type with its associated type-bound procedures would allow parameterizing them consistently and in a way that is more useful than in Fortran 2003, while not requiring one to put differently-parameterized pieces of a logically cohesive entity into separate modules.

1.9 Exception generation and handling

Introduction

A provision for generating, detecting and handling exceptions is badly needed.

Proposal

Revive John Reid's technical report draft.

It would make sense to allow a HANDLE section within a procedure.

As in John Reid's proposal, it should be possible to declare and raise user-defined exceptions, and to detect system-generated exceptions. The set of system-defined exceptions should be standardized. To simplify the proposal, if an exception occurs during evaluation of an array expression, a WHERE construct, or a looping construct, no provision should be made to discover the array element or loop index causing the exception.

This is related to the proposals concerning constructs (see 2.1).

2 Useful

2.1 Improvements to constructs

In respect to many of the following, a construct that doesn't actually do anything would be useful. It's actually necessary for one of them.

2.1.1 All constructs should be scoping units

Introduction

When developing a program, one sometimes needs to have several automatic arrays that are the same size, or that have sizes that depend on factors that cannot be evaluated as specification expressions. One can't compute these sizes and put them into variables, because this would happen too late to affect elaboration of the program unit's scope. One also sometimes needs an automatic array conditionally. If constructs were scoping units, one could compute the sizes, and then put the declarations of these arrays within a construct. The declaration could be within an IF construct, which would allow the automatic array to exist conditionally. One currently needs to use pointer or allocatable arrays to achieve this functionality.

When modifying an existing program unit with which one is unfamiliar, either because one is not the author, or one wrote it long ago, it would be useful to be able to declare entities to be local to a construct, so as to avoid clashes with existing uses of existing entities without needing to study a large program unit. This would decrease maintenance costs somewhat, and development costs a little bit less. Defining constructs to be scoping units would allow this.

Proposal

Define constructs to be scoping units. Allow to declare variables, parameters and types within them. In the case of DO constructs, rather than requiring to declare the induction variable within the construct in order to make it a construct variable, declare it within the DO statement, *viz.* `DO INTEGER::I = 1, 10`. For symmetry, allow the same declaration in `FORALL`, even though induction variables of `FORALL` loops are already defined to be construct variables. This would also allow to specify the kinds of `FORALL` induction variables.

For those constructs that have multiple branches, *viz.* IF and SELECT, each branch should be a separate scoping unit.

It would be useful to have a construct that serves no purpose other than to enclose a scoping unit – similar to the **begin ... end** constructs of Algol and Ada.

2.1.2 EXIT should be allowed to apply to any construct

A few algorithms could be more clearly expressed, without resort to extra tests or GO TO statements, if EXIT could be applied to any construct, not just DO constructs. One example is "If X is not in the set S do P" where S is represented by elements of an array, say A, from 1 to N, where N is the size of S. For compatibility, an exit statement without a construct label must continue to refer to the nearest containing DO construct, but if the EXIT statement has a label it should be allowed that the label is one on any containing construct, not just a DO construct.

2.1.3 Case blocks

Introduction

Sometimes, within the purview of a select case construct, one finds a need to do the same thing before or after some subset of them. This can be done by using extra select case constructs before, within or after the main one.

Proposition

Allow to put a construct around some subset of the case blocks in a select case construct. Thereby, one wouldn't need to write the extra ones. In the case one wants to do the same thing before several case blocks, the compiler would need to write the extra select case construct, but in the case one wants to do the same thing after several case blocks, one gets away without an extra select case construct. Example:

```
select case ( expr )
case ( c1 ); ...
begin
  stuff before c2 and c3, but only if expr is in one of them.
  The processor replaces the ``begin'' with ``case ( c2, c3)'',
  then the extra stuff, then another ``select case ( expr )''
  case ( c2 ); ...
  case ( c3 ); ...
```

```
end
stuff after c2 or c3.  The processor just generates a jump to
here after c2 and c3, and then one to the ``end select'' after
the ``stuff after c2 or c3''.
case ( c4 ); ...
end select
```

2.1.4 More general ranges in case selectors

Introduction

In an array section selector, on can specify a step. One cannot do this in a *case-selector*.

Proposition

Allow the same generality for a *case-selector* as for a *section-triplet*.

2.1.5 CASE .AND.

Introduction

Sometimes one needs to execute the block after a CASE statement only when the *case-expr* has one of the *case-value*s AND a logical expression is true, otherwise one needs to execute the block after the CASE DEFAULT statement. This can't be done except by duplicating the block after the CASE DEFAULT statement or by putting it into a procedure. One can't even get there with a GO TO statement.

Proposition

Allow .AND. *scalar-logical-expr* after the (*case-value-range-list*) on a *case-stmt*, with the meaning that the *block* after that *case-stmt* is not executed if the *scalar-logical-expr* is false; the one after the CASE DEFAULT statement is executed instead.

2.1.6 Construct labels ought to be local to the construct

The only place a construct label can be referenced is from within the construct, or on its end statement. It would be helpful if they were defined to be local to the construct they label, not to the scoping unit containing that construct.

2.1.7 DO constructs that test at the end would be useful

One occasionally needs to test a loop at the end instead of (sometimes as well as) at the beginning. This can be done by putting a conditional EXIT statement as the last thing in the loop. A little bit cleaner would be to allow any DO construct to end with WHILE (*scalar-logical-expr*) or UNTIL (*scalar-logical-expr*) instead of END DO.

2.1.8 A subconstruct done on explicit exit would be useful

Introduction

Sometimes a construct has several explicit exits, and needs to do the same thing at each of them – but not upon normal termination of the construct. This can be accomplished in at least four clumsy ways: Duplicate the code, move the code into a procedure, set a flag and test it after the loop, or use some GO TO statements.

Proposition

A cleaner solution is for the language to provide a subconstruct that is executed when an EXIT statement that refers to the construct is executed. Something like the following:

```
do ...
  ...
  if ( <condition> ) exit
  ...
on exit
  ! Stuff done at every explicit exit, but not normal loop termination
end do
```

It would be similarly useful to have an ON RETURN section of a procedure that is executed when a RETURN statement is executed, but not when an END statement is executed.

The proposal that EXIT should be allowed to refer to any construct (see 2.1.2) would be affected by this proposal.

It isn't necessary to tie this functionality to the EXIT and RETURN statements. For example, it could instead be provided by a block-wise exception handling system. EXIT, or another statement that doesn't carry all the baggage of an exception handling system, would be simpler.

2.2 Improvements to the type system

2.2.1 Regularize type reference

Replace syntax rule R401 by

R401	*type-spec*	**is**	*intrinsic-type-spec*
		or	TYPE (*derived-type-spec*)
		or	TYPE (*intrinsic-type-spec*)

2.2.2 Allow parameter declarations inside of TYPE definitions

One sometimes needs to declare a parameter for the purpose of declaring a component of a type. It would reduce maintenance costs somewhat if parameters needed for this purpose and no other could be declared within the type. They could be accessed from without the type, in contexts other than variable definition contexts (16.5.7), by using component selection notation (provided they are not private). Dynamic parameters (2.10), that depend on length parameters of the type, would also be useful.

2.2.3 Provide an attribute that turns off intrinsic assignment

It is sometimes desirable to prohibit intrinsic assignment for objects of a derived type, except within the scoping unit where the type is defined. This can at present be done only by providing a type-bound defined assignment that prints an error message and stops, but this detection of a violation of the developer's intent occurs at run time, not compile time, and can't be done with a PURE subroutine. It is difficult (impossible?) to make defined assignment work one way within the scoping unit where the type is defined, and differently elsewhere, so if you prevent assignment in this way, it's gone altogether. Ada spells an attribute with this property "limited."

2.2.4 Extend the type parameter system to types

The type system would be more useful if a new variety of type parameters, that are types, were provided. This wouldn't be necessary if a generic package system (1.8) were provided.

2.2.5 Make COMPLEX a SEQUENCE derived type

In some applications it is necessary to update only the real or imaginary part of a complex variable. To change the real part of a complex variable `C`, one currently needs to write

```
  C = cmplx(new_real,aimag(C))
```

This doesn't look too bad, but consider the case of a complicated reference:

```
MyThing(I,23*(J-11*K),Func(M,I,J,K))%MyField(L+3)%Radiance(1:2,1:2) = &
  & cmplx(new_real,&
  & aimag(MyThing(I,23*(J-11*K),Func(M,I,J,K))%MyField(L+3)%Radiance(1:2,1:2)))
```

A processor might deduce that the LHS and the argument of `aimag` are the same (but it might be confused by the apparent need to invoke `Func` twice if `Func` isn't pure), but you must admit that the cost of maintenance is higher and its reliability lower than with the simpler alternative proposed below. One could simplify this a little bit by writing

```
ASSOCIATE ( &
  & C => MyThing(I,23*(J-11*K),Func(M,I,J,K))%MyField(L+3)%Radiance(1:2,1:2) )
  C = cmplx(new_real,aimag(C))
END ASSOCIATE
```

but this is still ickier than necessary.

It would be nicer if COMPLEX was a parameterized sequence derived type having two components, the real part and the imaginary part, in that order called, say, `REAL` and `IMAG`. Then the above assignment would be simplified to

```
  C%real = new_real
```

or

```
  MyThing(I,23*(J-11*K),Func(M,I,J,K))%MyField(L+3)%Radiance(1:2,1:2)%real = &
    & new_real
```

A side effect of this addition is that there would be a new constructor for objects of complex type, named COMPLEX. Its syntax of usage would be the same as for other derived-type constructors. This is a desirable side effect. It is an open question whether we want to specify that either component has a default value of zero; this would make the COMPLEX constructor work more like the CMPLX intrinsic function.

This proposal would interact with the proposal to regularize type reference (2.2.1).

2.2.6 Extend LOGICAL type

If the LOGICAL type were extended by providing (1) a length parameter and (2) a SELECTED_LOGICAL_KIND intrinsic function, much of the functionality desired for a BIT data type could be realized. The SELECTED_LOGICAL_KIND intrinsic function should take an argument that gives the number of bits in a logical variable. There should be no assumption that independent logical variables, or different elements of a logical array, are packed, but consecutive elements of a logical string should be packed. The need for this wouldn't be as pressing if the proposal concerning subranges of integers (1.3.1) is adopted.

2.2.7 Protected types would be useful

Fortran 2003 has protected variables, but if they have the target attribute, their values aren't protected. A PROTECTED attribute should be provided for types, to indicate that objects of such types cannot be allocated or deallocated, or their values changed, except by procedures within the module where the type is defined. This would allow one to have a local (nonprotected) pointer step through a linked list of protected objects without being able to change them.

2.3 Improvements to generic system

2.3.1 Allow generic specifications to be partial applications

Introduction

I have a sparse matrix package that includes a MatrixAdd function, with interface

```
function MatrixAdd ( A, B, Subtract ) result ( Z )
  type(Matrix_T), intent(in) :: A, B
  logical, optional, intent(in) :: Subtract
  type(Matrix_T) :: Z
end function MatrixAdd
```

The functionality is that it adds $A+B$ unless the `Subtract` argument is present with the value `.true.`, in which case it subtracts $A-B$.

One cannot access this function with a defined operator. One could wrap it with additional functions that have only two nonoptional arguments, but this increases code bulk. Numerous studies have shown that the single most reliable predictor of lifetime cost of software is code bulk.

Proposal

Allow values to be specified for some arguments in an interface block, either in a [module] procedure statement, or in an interface body. Only the remaining arguments are visible, as arguments, when the procedure is accessed by using the *generic-spec*. After the values of some of the arguments are specified, the remaining arguments shall satisfy the present requirements.

For example, it would be useful to be able to declare something like

```
interface operator(+)
  module procedure MatrixAdd ! or MatrixAdd(subtract=.false.)
end interface
interface operator(-)
  module procedure MatrixAdd(subtract=.true.)
end interface
```

with the requirement that after specifying values to use for some arguments, in the interface, there remain one or two nonoptional arguments for which values are not specified, and these arguments meet the present requirements for defined-operator interfaces. In the functional programming community, this is called "partial application" or "Currying" (after Haskell Curry) of the `MatrixAdd` function.

The procedure `MatrixAdd` supports several different representations of sparse matrices, and has a lot of analysis to figure out where the nonzeroes of the output will be, and what representation to use. There are only a few places where it looks at the `Subtract` argument. It is undesirable to duplicate the code and specialize the two copies for the `Subtract = .true.` and `Subtract = .false.` cases, because that introduces the opportunity to create incorrect inconsistencies between them as a consequence of maintenance.

2.3.2 Add POINTER and ALLOCATABLE attributes to criteria for generic resolution

Introduction

I have a generic interface `Allocate_Test` that allocates an object, tests the status, and prints an error message if an error occurs. I cannot have a specific interface for a to-be-allocated argument that has the POINTER attribute, and another for a to-be-allocated argument with the same type, kind type parameters and rank that has the ALLOCATABLE attribute. So I need different generic names for allocatable objects and pointer objects. If I change an object from pointer to allocatable or vice-versa, I have to track down all of the `Allocate_Test` invocations for that variable and change them to the other one. Avoiding this labor was one of the justifications for the generic facility.

Proposal

Allow the POINTER and ALLOCATABLE attributes to be used for generic resolution. If the only difference between two specific interfaces is that one has neither the POINTER nor ALLOCATABLE attribute for some argument, and the other one has one of those attributes for the corresponding argument, the interface is ambiguous. If one has the POINTER attribute and the corresponding one has the ALLOCATABLE attribute the interface is not ambiguous, at least so far as that pair of specific procedures is concerned.

If a dummy argument has the POINTER or ALLOCATABLE attribute, the corresponding actual argument is required to have the same attribute. Therefore this change would not invalidate any existing program.

2.3.3 Allow additional optional dummy arguments for assignment or operator procedures

Introduction

One sometimes has procedures that have optional arguments that one would like to use to define assignment or operations. These cannot be used because of restrictions on the number of arguments such procedures are required to have. One could wrap these with additional procedures that have the required number of arguments, but this increases code bulk. Studies have shown that the most important factor contributing to development and maintenance expense is code bulk.

Proposal

Allow procedures that define assignment or operations to have optional arguments. Require a procedure that defines assignment to have at least two arguments, with any after the first two required to be optional. Require a procedure that defines an operation to have at least one argument, and if they have more than two, those after the second one are required to be optional. If a procedure for which all arguments after the first one are optional defines an operation, the operator can be used as either a binary operator or as an unary operator.

The generic resolution rules already handle optional arguments correctly.

2.3.4 Compound assignment/operation generics would be useful

Introduction

Some applications have complicated derived-type objects on which one wishes to define operations and assignment. In these cases, the result of the function that defines the operation will be an anonymous object of a derived type. Finalizers help to get these to work correctly, but don't address the performance problems that arise as a consequence of separating defined assignment from the defined operation, especially if assignment is a "deep copy." These could be ameliorated if a compound assignment/operation generic interface could be defined.

Proposal

Define a new variety of interface block, introduced by an INTERFACE statement that specifies compounded assignment and operation, e.g. INTERFACE COMPOUND(=,.MYMULT.). The first thing-o would have to be "=" so it may not be necessary to say so. On the other hand, saying so leaves room to extend it to pointer assignment.

These would be used in statements of the form *variable* = *expr* .MYMULT. *expr* or *variable* = .MYUNARY. *expr*.

Require all of the procedures named or described within the interface block to be subroutines with two or three arguments, with the first becoming associated with the *variable* and the second (and third) becoming associated with the *expr*(s). Also see 2.3.1 and 2.3.3, which would have impact on this specification.

2.3.5 Allow to define pointer assignment

Introduction

In some applications, data structures arise that are sufficiently complicated that one cannot point to a place in the program and say "this is the appropriate place to deallocate such-and-such entity." In these cases, one can frequently use *reference counters* to keep track of the number of pointers of which the object is a target, and deallocate the object when its reference count is reduced to zero. This presently requires that all pointer reassignment be done within subroutines. This camouflages the abstraction, thereby increasing maintenance costs. In addition, all that is necessary to break this abstraction is a pointer assignment statement that doesn't change the reference counter.

Proposal

Provide for defined pointer assignment in the same way as defined assignment is provided. This would allow to do the computations necessary to maintain reference counts within the procedure that defines the assignment, and would "cover up" intrinsic pointer assignment, thereby preserving abstraction.

2.4 Procedure improvements

2.4.1 Provide for a specification that there are no implicitly EXTERNAL procedures

Introduction

If one wishes to construct a program entirely from module procedures, it would be useful if the processor could be instructed to announce an error if there is a reference to a procedure that is not a module procedure, not an intrinsic procedure, and has not explicitly been given the external attribute.

Proposal

Add a specification to the IMPLICIT statement, say NOEXTERNAL, to indicate that no procedure is to be given the EXTERNAL attribute implicitly. I understand that some processors have an extension to allow an UNDEFINED specification to the IMPLICIT statement, to indicate both the effect of the NONE specification, and the NOEXTERNAL specification advocated here.

2.4.2 Optional `subroutine` *name* on `end` statements

Module procedures are required to end with `end subroutine` *name* or `end function` *name*. This is generally a good thing, but it makes it impossible to convert a FORTRAN 77 external procedure to a module procedure by using `include`. To cater for this desire, allow the `subroutine` *name* or `function` *name* part to be optional on an *end-subroutine* or *end-function* statement.

2.4.3 Improvements in usability of optional arguments

See also 2.3.3 about optional dummy arguments for assignment or operator procedures, and 2.8.1 about distfix if-then-else operators.

2.4.3.1 Disassociated pointer actual argument with optional nonpointer dummy

It should be allowed to associate a deallocated allocatable actual argument, or a disassociated pointer actual argument, with an optional nonpointer nonallocatable dummy argument, in which case it should be considered to be absent instead of an error. If one now wishes to achieve this effect, one needs an IF. . . ELSE IF. . . END IF sequence with 2^n branches to handle n arguments. One can declare the dummy arguments to be ALLOCATABLE or POINTER instead of OPTIONAL, and test for ALLOCATED or ASSOCIATED instead of PRESENT, but this degrades the generality of the procedure, and may have undesirable implications for optimization.

2.4.3.2 Default initial value for absent optional argument

A frequently requested feature is to be able to specify a default initial value for absent optional dummy arguments. It would seem to be easy to do for nonpointer nonallocatable scalars and explicit-shape arrays, especially if the proposal to provide for reinitialization on every invocation (2.11) is adopted. If the proposal for non-null initial targets for pointers (2.12.2) is adopted, it would be easy to provide a default target for absent optional pointer dummy arguments. For assumed-shape or -size arrays, or arguments with assumed length parameters, the assumed quantities can be taken from the initialization. If an absent optional allocatable dummy argument has a default initial value specified, it is initially allocated and has that value (and shape and length parameters if appropriate). The distfix if-then-else operator (2.8.1) would go some distance toward solving the same problems as this proposal, but it would still require an auxiliary named local variable.

2.4.3.3 Elements and sections of optional dummy arguments

It would be convenient if it were possible to use an array-subscripted optional dummy argument that is absent, or a section or an element of an absent optional dummy array argument, as an actual argument, and have the corresponding dummy argument be absent when control reaches the invoked procedure. If one wishes to achieve this effect at present, one needs an IF-ELSEIF-ELSE-ENDIF sequence with 2^n branches to handle n actual arguments.

It would be convenient if it were possible to use an array-subscripted optional dummy argument that is absent, or a section or an element of an absent optional dummy array argument, as a *data-target* in a pointer assignment statement, and have the *pointer-object* become disassociated.

A conspiracy of the distfix if-then-else operator (2.8.1) and allowing disassociated pointers as actual arguments associated with optional nonpointer dummy arguments (2.4.3.1) would go some distance toward solving the same problems as this proposal. E.g., one could use `present(a) ? a(i:j:2) : null()` as an actual argument.

2.4.3.4 Absent optional arguments in I/O control keyword specifiers

When an absent optional argument is used with a keyword specifier in an input/output specifier or a specifier in an allocate or deallocate statement, the specifier should be considered not to have appeared. The terminology for them ought, in parallel, to be changed to use "present" instead of "appear". If one wishes to achieve this effect at present, one needs an IF-ELSEIF-ELSE-ENDIF sequence with 2^n branches to handle n specifiers.

2.4.4 Intrinsic procedures

2.4.4.1 Linear algebra

We have `MaxLoc` and `MinLoc`, but what one usually needs for linear algebra is `MaxLoc(Abs(A))`. This can be gotten as written, but for those of us who don't trust the optimizer not to make a temporary variable for `Abs(A)`, it would be reassuring to have `MaxAbsLoc` (and for symmetry, `MinAbsLoc`). Similar arguments lead to a desire for `MaxAbsVal` and `MinAbsVal` to go along with `MaxVal` and `MinVal`.

Many algorithms in linear algebra have highest performance when blocked. The block dimensions depend on the characteristics of the cache. Provide intrinsic procedures to inquire the number of levels of cache, and the size and relative speed (or relative access time) of each level of cache. Main memory should be included in the hierarchy. It may be last, or next-to-last if virtual memory is included. The reported size should be in terms of an argument of the procedure, not in machine-dependent units such as "words" or "computer science" units such as "bytes." The reported relative speed (or relative access time) should take the cache data transfer width into account. Additional parameters of the cache, e.g. is it set-associative and if so how many ways?, is it write-through?, is it write-back? etc., are interesting but haven't as much effect as the first three.

2.4.4.2 ATAN2 is an anachronism

The `ATAN2` intrinsic function is not needed, given the power of generic resolution and optional arguments since Fortran 90. Define `ATAN(Y,X)` to have the functionality of `ATAN2(Y,X)`, and deprecate the latter.

2.4.4.3 Provide hyperbolic functions for complex type

The hyperbolic functions COSH, SINH and TANH aren't available for complex type. There are simple identities involving hyperbolic and trigonometric functions of real arguments, e.g. $\sinh z = \sinh x \cos y + i \cosh x \sin y$ and $\cosh z = \cosh x \cos y + i \sinh x \sin y$, or trigonometric functions of complex arguments, e.g. $\sinh z = -i \sin iz$, $\cosh z = \cos iz$, and $\tanh z = -i \tan iz$ but these are tedious to use. Furthermore, the processor might have better methods that depend on the architecture. In light of the last three identities, requiring the processor to provide hyperbolic functions for complex arguments does not impose a burden on developers, and would provide a convenience to users.

2.4.4.4 Procedures to compute functions that are "companions" efficiently

One frequently (Oh, OK, sometimes) needs to compute both cosine and sine, both hyperbolic cosine and sine, or both quotient and remainder. These pairs of functions are related in such a way that it is convenient to compute them together, and more efficient to do so than to invoke existing intrinsic functions or operations to compute them separately.

Many processors have such procedures lurking "under the covers" in their run-time libraries, and some exploit them when optimization is requested, but users can't count on this.

It would therefore be useful for the standard to specify intrinsic subroutines that compute both functions in each of these three "companion" pairs, say SINCOS, SINHCOSH and QUOTREM.

2.4.4.5 ASCII arguments for LGE etc.

It seems a bit inconsistent that LGE, LGT, LLE and LLT, which compare characters according to the ASCII collating sequence, do not allow arguments of SELECTED_CHAR_KIND('ASCII').

2.4.4.6 Help for optimization

Most small loops have higher performance if unrolled. Sometimes, there is a variable that prevents unrolling because it is scalar, but if it were an array having the same dimension as the amount of unrolling of the loop, the loop could be unrolled. Provide an intrinsic specification function that takes a DO construct label as input and reports the amount of unrolling of the loop as output.

2.4.4.7 Get an unused input/output unit number

Everybody eventually gets around to writing a "get an unused input/output unit number" routine. In writing it, we all wonder "what is the largest allowed unit number?" It would be useful to have an intrinsic procedure that returns an unused input/output unit number. One could also make a case to have a "maximum input/output unit number" named constant in the ISO_FORTRAN_ENV module. One might guess HUGE(0), but this may be unnecessarily large in some environments.

2.4.5 Allow the GENERIC statement as an alternative to an interface block

The GENERIC statement can be used within a type definition to specify the equivalent of an interface block. There is no reason it could not be allowed outside of a type definition as an alternative to an interface block.

2.4.6 KIND arguments whose values are used for generic resolution

It is not possible to write user-defined procedures for which the values of kind arguments are used for generic resolution, as is done for several intrinsic procedures.

For parameterized derived types, the values of their kind parameters, not the kind parameters of the kind parameters, are used to determine the kinds of objects of the type. This can be done because the kind parameters have the KIND attribute.

If it were allowed for dummy arguments to have the KIND attribute, as is allowed for derived type parameters, and if a restriction were put on the corresponding actual arguments that they must be initialization expressions, again as is required for derived type parameters, it would be possible to use the values of these arguments for generic resolution.

2.4.7 A "walkback" facility would be useful

Occasionally, at least for the purpose of producing informative error messages, it would be useful to know the path of procedure invocations that led to a particular place in a program. There are several ways this information could be provided. Although the values provided would be processor dependent, their types could be specified by the standard.

One possibility is to provide a type in ISO_FORTRAN_ENV, and a pointer variable of that type. The components of the type would be a deferred-length character pointer, an integer, and a pointer of that type. The pointer variable in ISO_FORTRAN_ENV should be defined to be NULL() while the execution sequence is in the main program. Whenever a procedure is called, a new object of the type is created, with its pointer component having the same pointer association status as the pointer variable in ISO_FORTRAN_ENV, and the pointer variable in ISO_FORTRAN_ENV is then associated with this new object. Upon return, the inverse process occurs. Such a type ought to be a protected type (2.2.7).

Another possibility is to provide an opaque type and three procedures in ISO_FORTRAN_ENV. One procedure initializes an object of the type to mean "tell me about the caller of the current procedure." Another procedure takes one object of the type and returns (optionally) a character variable, (optionally) the length of the data it hoped to have put into that character variable, (optionally) an integer that has processor-dependent meaning, and (optionally) a new value for the object of opaque type. A third procedure takes a variable of the opaque type and returns a logical value indicating whether there is a "next" object in the list.

Whether the values or procedures actually contain or do something useful should be processor dependent, with a note explaining what a processor is expected to do, and that there may be processor-dependent means to turn the facility off.

To start things off, it would be useful to have an intrinsic subroutine that returns the "line number" of its call – which would, of course, be a processor-dependent value. (For example, the "line number" could indicate a position in a scoping unit, a program unit, or a file.) This could be gotten by calling a procedure that does one step of the process described in preceeding paragraphs, and returns the "line number," but having a direct way to do it would be useful.

2.5 Allow empty CONTAINS part

Is there a good reason that a module, procedure or type definition has to have procedure definitions after the CONTAINS statement? What would be hurt by allowing not to have any? Sometimes when programs are generated automatically, they end up with an empty CONTAINS part, and then you have to go fix it manually, or whine (ineffectually) to the unsympathetic guys who wrote the processor, for which you have no source code.

2.6 A SWAP statement would be useful

One sometimes needs to exchange two variables, or more rarely two pointers. This requires declarating a temporary variable. But then the next one to maintain the code wonders "Is this temporary variable used anywhere else?" Also, simple compilers don't bother to ask themselves that question and answer it, so they don't produce as efficient a translation as they might otherwise. This dataflow question would be easier to answer if constructs were scoping units, as described in section 2.1.1, so that one could create a "very local" temporary variable. But this makes the program more bulky.

It would therefore be useful to have statements to exchange data and pointers. Examples of syntax of such statements are `A :=: B` and `A <=> B`. The former could be used wherever `A=B` and `B=A` are both permitted. The latter could be used wherever `A=>B` and `B=>A` are both permitted. These are simple to implement, simple to describe, improve readability of programs, and are more likely to be optimized by a simple compiler.

2.7 Allow array constructors and concatenation on the LHS of intrinsic assignment

It would occasionally be helpful if it were possible to put an array constructor on the left side of an intrinsic assignment, provided each *expr* that is an *ac-value* is allowed on the left side of an intrinsic assignment.

For example, if one needs to fill consecutive elements of an array *except for the middle one* with the results of a function, one could write

```
(/ ( a(i), i = 1, n/2-1 ), (a(i), i = n/2+1, n) /) = bfunc ( x, y, z )
```

instead of

```
b(1:n-1) = bfunc ( x, y, z )
a(1:n/2-1) = b(1:n/2-1)
a(n/2+1:n) = b(n/2:n-1)
```

It would occasionally be helpful if it were possible to put a character expression involving concatenation on the left side of an intrinsic assignment, provided each operand of the // is allowed on the left side of an intrinsic assignment.

For example, one could write

```
a // b // c // d = x
```

instead of

```
a = x(:len(a))
b = x(len(a)+1:len(a)+len(b))
c = x(len(a)+len(b)+1:len(a)+len(b)+len(c))
d = x(len(a)+len(b)+len(c)+1:len(a)+len(b)+len(c)+len(d))
```

It may be possible and reasonable to allow these things in input lists. I don't think it's reasonable to allow either of them in any other variable definition context.

If a distfix IF-THEN-ELSE operator (2.8.1) is provided, it could be allowed in variable definition contexts, provided its second and third operands are allowed, e.g.

```
allocate ( a(n), stat = present(status) ? status : myStat )
```

2.8 Operator improvements

2.8.1 A distfix IF-THEN-ELSE operator would be useful

One sometimes needs to select one thing or another to be used within an expression. At present, one creates a temporary variable, sets that variable with an if-then-else or where-elsewhere construct, then evaluates the expression using that variable.

Another use is to compute whether an actual argument is present. This cannot be done by creating a temporary variable. Instead, one uses an if-then-else construct with the argument textually present in one branch but not the other. If one wants to compute whether n actual arguments are present, one needs a complicated nest of if-then-else constructs with 2^n branches.

Other languages include a distfix if-then-else operator. For example, in C one can write `p ? x : y`, which is pronounced *if p then x else y*. If such an expression were to be the *target* in a pointer assignment, its result should be a target, not a value. This spelling could work in Fortran as well. Clunky alternatives might be `.IF. p .THEN. x .ELSE. y .ENDIF.`, or more briefly `p .THEN. x .ELSE. y`. For the case of computing whether an actual argument is present, the syntax might be `p ? x`, pronounced *if p then x is the actual argument, else the actual argument is not present.*

The difference for these operators as compared to existing operators is that only the first operand (`p` in the example) is initially evaluated. Then the second operand (`x` in the example) is evaluated if (where in the elemental case) `p` is true, else the third operand (`y` in the example) is evaluated if it appears. The first operand is required to be logical, while the others are required to be of the same type, type parameters and rank. If `p` is an array, `x` and `y` have to be the same shape as `p`. In the `p ? x` case, `p` would necessarily have to be a scalar. The result type, type parameters and rank are those of `x`. If `p` is a scalar, the shape of the result is `x` or `y` depending on whether `p` is true or fales. If `p` is an array, the shape of the result is the shape of `p`.

It would introduce substantial complication into the defined-operator discussion to allow to overload this operator. Fortunately, it seems unlikely that would be useful.

The functionality almost exists in the MERGE intrinsic function. The reason it isn't quite the same as what is described here is that the standard specifies that all arguments of a function are evaluated before the function is invoked. Also, we don't have a two-argument MERGE that causes its result not to exist (for purposes of argument association) if its first argument is false.

2.8.2 .ANDTHEN. and .ORELSE. operators

The standard presently allows a processor to short-circuit evaluation of logical expressions. For example, in `A .AND. B`, the processor is allowed not to evaluate `B` if `A` is false. It is sometimes desirable, however, to *require* that the processor not evaluate `B` if `A` is false, as opposed simply to *allowing* it not to. Here's an example:

```
if ( present(x) .and.  x /= 0 ) ...
```

One can't *depend* on the processor not trying to evaluate `x /= 0` if `x` is not present.

To support this desire, add an `.ANDTHEN.` operator, the semantics of which require the processor to evaluate the first operand first, and then prohibit it from evaluating the second operand if the first is false. The example becomes:

```
if ( present(x) .andthen.  x /= 0 ) ...
```

Similar considerations apply to the .OR. operator, leading to the desire for an .ORELSE. operator, in which the second operand is prohibited to be evaluated if the first is true.

These operators are, of course, even more useful elementally in WHERE statements and constructs. For example

```
where ( x > 0.0 .andthen.  log(x) < tol ) ...
```

2.8.3 Mixed-kind character concatenation

Concatenation operations in which one operand is of ASCII kind and the other is of ISO-10646 kind ought to be allowed, with the result being of ISO-10646 kind.

2.9 Improvements to array system

2.9.1 Extension to subscript

Suppose one has an array A of rank r.

It would be useful to be able to use a single subscript S of rank $k+1$ and extents $(r, n_1, ..., n_k)$ as a subscript for A. The elements of the rank-one sections in the first dimension of S are used consecutively as subscripts for A, resulting in a rank k array of extents $n_1, ..., n_k$. This provides a more general scatter/gather facility than the present vector subscript facility. This is not the same as using the elements of the rank-one sections in S as vector subscripts for A, which would result in a rectangular section of shape $(n, n, ..., n)$ (in the case S is of rank 2).

Example:

Suppose we have arrays A3 with dimensions (10,10,10) and S3 with dimensions (3,2). If we assume `S3 = reshape( (/3, 4, 5, 6, 7, 8/), (/3,2/) )` $= \begin{bmatrix} 3 & 6 \\ 4 & 7 \\ 5 & 8 \end{bmatrix}$, then `A3(S3)` is a rank-1 extent (2) array that can appear in a variable-definition context (except perhaps not as an actual argument associated with a dummy argument having INTENT(OUT) or INTENT(INOUT)); it specifies the same array as `(/ A3(3,4,5), A3(6,7,8) /)`, which cannot appear in a variable-definition context. This is different from A3(S3(1,:),S3(2,:),S3(3,:)), which can appear in a variable-definition context, but is an object with extents (2,2,2), not (2). The former is an arbitrary collection of elements, while the latter is a rectangular section.

As a degenerate case, one could allow a single subscript of rank one and extent equal to the rank of an array as the only subscript for that array. The elements of the first array are treated as the subscripts of the second array, resulting in accessing a single element of the second array. The result is a scalar, not an array of extent (1), or an array of extent (1,1,...,1).

Example:

Suppose we have arrays A3 with dimensions (10,10,10) and S3 with dimension (3). If we assume `S3 = (/ 3, 4, 5 /)`, then `A3(S3)` is the same as `A3(3,4,5)`, not `A3(3:3,4:4,5:5)`.

2.9.2 Extensions to array bounds declaration

2.9.2.1 Allow an array as a bounds specification

Introduction

Suppose one has a dummy array D and one needs an automatic array A with the same rank and extents as D. It would be convenient to be able to write `integer A(size(D))` instead of `integer A(size(D,1),size(D,2),...)`.

Proposal

Allow the bounds of an array to be given by a rank-1 array, both in the declaration of an explicit-shape array and in an ALLOCATE statement. If the upper bounds are given by an array, the lower bounds have to be default (i.e. 1) or given by a rank-1 array of the same extent as for the upper bounds. If the lower bounds are given by an array, the upper bounds have to be given by a rank-1 array of the same extent as for the lower bounds.

2.9.2.2 Allow any combination of assumed and explicit shape

Introduction

In many applications, one knows the values of some array bounds, but not all. In one application, I have a 2×2 matrix at every point along a path of indeterminate length. If I could declare this using `incoptdepth(2,2,:)`, I would have some confidence that the processor would optimize the MATMUL operations along the path, without needing to write `incoptdepth(1:2,1:2,j)`. At another point in the same application, I have an array that corresponds to the σ_-, π and σ_+ components of a Zeeman-split spectral line. The first dimension here is naturally `-1:1`.

Proposal

Allow any dimension of a pointer or allocatable array to be declared with explicit, assumed or deferred shape, independently of the others. Allow the last dimension of a pointer array to be specified by an asterisk. If the bounds for any dimension are given explicitly in the declaration, the same values shall be specified for those bounds in an ALLOCATE statement. If a pointer with such bounds is the left-hand side in a pointer assignment statement, and any bounds are specified, any bounds explicitly specified in its declaration shall have the same values in the pointer assignment statement.

2.9.2.3 Specify bounds of arguments, but allow/prohibit noncontiguity

Introduction

Sometimes one wants to specify a bound of a dimension of a dummy argument, but not require elements to be contiguous – so as not to trigger copy-in/copy-out argument passing. At other times one wants to require contiguity, but still be able to use assumed extent.

Proposal

Allow a dimension specification of the form [*low-bound*] : [*high-bound*] [: [*stride*]]. If the *stride* does not appear but the final colon does, the actual argument need not be contiguous – even if the *high-bound* expression appears. If the *stride* does appear, it shall be an initialization expression with the value 1, and the corresponding actual argument shall be contiguous.

Examples:

```
subroutine S ( A, B, C )
  real :: A( :, : )
  real :: B( :size(a,1):, : ) ! Doesn't require contiguous elements
  real :: C( ::1, : )         ! Requires contiguous elements.
  ...
```

2.9.3 More general rank remappings

Fortran 2003 allows the *data-pointer-object* in a pointer assignment statement to have higher rank than the *data-target* provided both bounds are specified for every dimension of *data-pointer-object* and *data-target* has rank one. This could be extended by allowing to specify both bounds for any consecutive sequence of dimensions of *data-pointer-object* provided the number of dimensions for which both bounds are not specified is one greater than the rank of the *data-target*.

Example:

In one application, I have a 3×3 matrix at every point along a path of indeterminate length. For reasons having to do with restrictions in the input/output package I am required to use, I have to store this as a rank-2 array in which the first dimension has extent 9. When It's time to use it – usually in MATMUL – I need to reshape it. It would be more convenient to write `P(:3,:3,:) => Q` or `P(:3,:3,:) => Q(:9,:)`. Notice that I cannot write `P(:3,:3,:) => RESHAPE(Q(:9,:),[9*size(Q,2)])`

In conjunction with the proposal in 2.9.2.2 to allow any combination of explicit and assumed shape, if P and Q were declared "`real, pointer :: P(3,3,:), Q(9,:)`" it would be nice if I could write simply `P => Q`.

2.10 Dynamic parameters

Automatic variables are convenient, but if one has several with the same (complicated) dimensions or lengths, it is tedious to declare them. It would be useful to have a class of parameters, identified explicitly by an attribute, say DYNAMIC, whose values are given by specification rather than initialization expressions. That is, they are allowed to depend on other entities in exactly the same way that dimensions and length parameters are. Dynamic parameters can obviously be allowed only within procedures. It should be prohibited for them to appear in variable definition contexts.

2.11 Initial values on every invocation

Introduction

Users who don't read the standard (or their textbooks) carefully are sometimes confused by initialization for variables. They don't realize that it happens exactly once, and not on every invocation. Leaving aside the confusion, it is sometimes desirable for initialization to happen on every invocation, and not automatically to imply the SAVE attribute.

Proposition

Provide an attribute for a variable, that can only be specified if it has initialization, that specifies that the initialization is performed every time the procedure is invoked. It should also be applicable to pointers. If the variable doesn't have the POINTER attribute, allow the initial value to be a specification expression. This attribute, and initialization in the presence of this attribute, ought not to imply the SAVE attribute.

2.12 Extensions to pointers and targets

2.12.1 Help for optimizers

Optimizers take advantage of the absence of the TARGET attribute to determine that a variable can't be changed by way of a pointer. This improves register utilization. If TARGET were extended with a list of pointers that were allowed to be associated with the target, more clues would be available to the optimizer. To be most useful, that is, to provide guarantees instead of promises from the programmer that could be violated, an attribute of a pointer that prohibits it from being a *data-target* or *proc-target* in a pointer assignment statement would be useful.

2.12.2 Non-null initial targets for pointers

It would be useful to be able to initialize pointers to targets other than NULL(). This is especially true for procedure pointers. Taken in combination with reinitialization on every invocation (2.11), this would allow a pointer to be initialized with a target that doesn't have the SAVE attribute.

2.13 Input/Output improvements

2.13.1 Option to output structure component names in NAMELIST

One can put structure component names in namelist input. This has documentary value. For output, the standard just says that the form is the same as for input. Since component names are optional in the input, most processors do not put them in the output. It would be useful to have a specifier that could be put in a namelist write statement (and maybe in the open statement as well) that specifies whether component names are required to appear in the output, required not to appear, or can appear at the whim of the processor.

2.13.2 Provide a way to know the end of a direct access file

It is sometimes useful to know the end of a direct access file. This could be done in at least two ways. One is to allow an END= specifier in a direct-access read statement, with the meaning that the branch is taken if one attempts to read a record that does not exist, and no records with larger record numbers exist. Another is to provide a named constant in ISO_FORTRAN_ENV that is the value of a status returned by the IOSTAT= specifier in the case that one attempts to read a record that does not exist, and no records with larger record numbers exist.

2.13.3 Relax unnecessary restrictions on input/output

2.13.3.1 SIZE= specifier

There appears to be no reason to require that SIZE= shall be specified only if ADVANCE= is specified and has the value “yes”. In fact, it would be useful if one could do it in other cases.

2.13.3.2 Nondefault characters

It would help our colleagues who need to use something other than the Latin alphabet if the prohibition against nondefault character kinds in character string edit descriptors, and in formats in general, were removed.

2.13.4 Simple generalization of numeric format specifiers

It is occasionally (OK, only rarely) useful to be able to input or output numbers in bases other than 2, 8, 10 or 16. To support that desire, extend integer format specifiers by allowing _*b* at the end (it doesn't work for real numbers because an exponent needs only a sign; the D or E isn't required). Require *b* to be a number between two and 16 (or maybe allow it to be as large as 36).

2.13.5 Control of the case of E or D in output

To control the case of the "E" or "D" that appears in output of real numbers, define a CASE mode of formatting, and LC and UC edit descriptors. A CASE mode of formatting would also apply to output of component names during namelist output (2.13.1).

2.14 Module improvements

2.14.1 More control over accessibility

I have a few modules that have enormous numbers of things that I don't want to enumerate in an ONLY list. Sometimes, I don't want them all. To cater for that desire, it would be useful to have an accessibility attribute that says "The things named here are accessible only by adding the ONLY attribute to the USE statement that accesses the module" (which would be the default if there's no list), and an attribute of USE that says "get everything except what's mentioned here."

2.14.2 Delete possibility that nonsaved module variables disappear

The standard presently allows that unsaved module variables may disappear when no procedure in the module is executing. Whether this happens is processor dependent. To my knowledge, no processor does this, and no user has asked a vendor to provide it.

Allowing for it complicates the discussion of undefinition in Section 16, and the discussion of finalization in Section 4, more than would be necessary without it. Because of this ambiguity whether a variable becomes undefined, there was difficulty in specifying when it comes back into existence. This complication was one of the reasons that initializers weren't put into Fortran 2003.

Since it's processor dependent, no portable program could depend on it. It's even hard to imagine how a nonportable program could depend on it.

It should be removed. Module variables should all be saved variables.

2.14.3 Allow a local name to be the same as a module name

If a module name appears in a USE statement in a scoping unit, no other entity in the scoping unit can be declared with the same name. The module name isn't used for anything else, so there's no harm in allowing it. There *is* harm in prohibiting it: If you already have a certain local name, and later need to use a module of the same name in the same scoping unit where that name appears, your only recourse is to change all occurrences of the local name to something different.

There also appears not to be a good reason that an entity declared in a module can't have the same name as the module. I end up sticking `_M` on the end of module names just so as not to conflict with procedure names – frequently the only procedure name – in the module.

2.14.4 Allow a USE inside of a type definition

Sometimes, one needs to reference something gotten by use association from inside of a type definition. In Fortran 95 the only possibility is a named constant, but in Fortran 2003 it will also make sense to want to access procedures. If the type definition is at module scope, then the USE is too. When processing module information, many processors read the module information for any USEs encountered at module scope, instead of putting that information in the using module's module information file (which would have the potential to cause enormous module information files). But they don't usually read module information for modules accessed by USE statements that aren't at module scope. So if we could put a USE statement inside of the type definition, we could potentially speed up some compiles.

2.15 Provide for the definition of pronouns

One sometimes has a subexpression that appears several times within a statement. One may wish to extract that subexpression into a temporary variable, so as not to need to trust the optimizer to eliminate all but one evaluation of it. But a temporary variable causes other problems, as discussed above in section 2.6. By allowing to define a pronoun that has statement scope, or a scope that extends from its definition to the end of a statement, these problems would be avoided. Its type would be taken from the expression that defines its value. For example, one might replace:

```
a(3*i+1) = b(3*i+1)
```

by

```
a( s @ (3*i+1)) = b(s)
```

I have indicated in the example a proposal that pronouns be defined with a special character, "`@`" in the example.

2.16 Extensions to C interoperability

2.16.1 Provide for assumed shape and optional arguments

The C interoperability facility of Fortran should include specification of C functions and/or C structs that allow to create actual arguments and use dummy arguments that have assumed shape, assumed type parameters, or are optional. The specified mechanisms should apply only to Fortran procedures that have the BIND(C) attribute.

3 Facilities that ought perhaps to be defined in optional parts

The facilities described in this section might perhaps best be done as an optional part or parts, so as not to require their support on systems where it would be difficult.

3.1 Provide support for associated variables

Many operating systems provide support for what are called "associated variables". These are variables that are associated with a file. They can be implemented efficiently on machines that have segmented memory management, simply by allowing to specify that a segment consists of one or more variables, and that a certain file is the swap file for that segment. This is most useful if the variable, or its last component (but not both) can have indefinite size.

Allow something like

```
TYPE :: Type1
  character(30) :: Name
  character(60) :: Address
  character(30) :: City
  character(10) :: Zipcode
  character(2) :: State
END TYPE Type1
```

```
TYPE(Type1) :: Var1(*) ! An attribute, say MAPPED, could be required here
OPEN ( ACCESS='mapped', FILE='AddressBook', STATUS='old', IOSTAT=ier ) Var1
```

or

```
TYPE :: Type2
  INTEGER :: NumTimes
  REAL :: Lat, Lon, TimeStart, TimeStep
  REAL :: MagField(3,*) ! This could require an attribute, say MAPPED, on the type
END TYPE Type2
TYPE(Type2) :: Var2
OPEN ( ACCESS='mapped', FILE='TimeSeries', STATUS='new', IOSTAT=ier ) Var2
```

after which references to `Var1` or `Var2` access the files `AddressBook` or `TimeSeries`, respectively.

The type `Type2` would have to behave in some ways like a SEQUENCE derived type.

3.2 Provide support for dynamically linked libraries

Many systems provide for what are called "dynamically linked libraries" or "shared libraries" or "shared objects." These are libraries of procedures that are not included into the executable file, but are rather added to the program during its execution. Many Fortran processors exploit these facilities. There should be support for programs to define the libraries that are accessible, and the entry names in those libraries that are to be accessed, by program variables.

The ALLOCATE statement could be used to specify that a library of dynamically-linked procedures is to be incorporated into a program and that a certain procedure pointer is to be associated with a procedure of a specific name within the specified library. The DEALLOCATE statement could be used to specify that a library of dynamically-linked procedures previously incorporated by an ALLOCATE statement should be removed from the program.

Here is an example:

```
ALLOCATE ( LIBRARY = '/mypath/mylibrary', &
    & ASSOCIATE = ( proc_ptr_1, 'entry_1'), &
    & ASSOCIATE = ( proc_ptr_2, 'entry_2'), STAT = oops )
```

Of course, it should be possible to specify the libraries and entry points by variables. Otherwise, the facility offers nothing new.