Subject: Wish List for Fortran after 2015
From: Van Snyder
References: 97-114r2 98-171r2 00-317 03-258 04-125r1 04-139r1 04-140r1 04-141r1 04-146r1 04-147 04-154 04-155 04-157 04-158 04-159 04-160 04-162 04-163 04-164 04-167 04-168r1 04-169 04-171r1 04-172 04-173 04-174 04-175 04-176 04-179 04-180 04-181 04-182 04-184 04-187 04-188 04-190 04-191 04-194 04-195 04-196 04-197r1 04-198 04-199 04-200 04-201 04-203 04-204 04-206 04-207 04-216 04-218 04-380r1 04-390 04-391r1 04-393 04-403 05-195 05-369 06-110 06-187 07-007r2 08-197 08-201 08-204 11-256 11-259 12-104 N1257 N1688 N2113

Herein, I list the wishes of my sponsors for features of Fortran, some having been desired since before Fortran 90. It is hoped that at least some of these will be implemented in future revisions of the Fortran standard. Other than being divided into *important* and *merely useful* groups, the discussion is not in any particular order, except that the proposal for *"Physical" or "engineering" units of measure* is the most important. 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, reliable, robust and efficient software.

1.1 “Physical” or “Engineering” Units of Measure

Introduction

The proposal for which authorization to proceed as an ISO Technical Specification was requested in June 2016 was N2113.

Incorrect use of physical units is a common error in scientific or engineering software. Other common errors are mismatching the types of actual and dummy arguments, and subscript bound violations. Explicit interfaces largely solve the latter problems, but do nothing directly for the former. (One can use derived types to provide a physical units system, at the expense of redefining intrinsic functions, operations, and assignment, for all combinations of units – a tremendous job for mechanics, saying nothing about thermodynamics, electronics, ... – and then you hope for inlining. If done using type parameters or integer components, it can distinguish length from time, but not kilograms from pounds.) 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 NASA contracts for over thirty years, and they *always* specified SI units.)

Proposal

Define a new UNIT or MEASURE attribute or type parameter (call it what you will) that can be specified for any numeric variable or named constant. Literal 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. Square root, or maybe even exponentiation of a unit by a rational number, would be useful. In the context of a unit definition, the integer literal constant 1 is considered to be the unitless unit.

Each unit declaration creates one or more generic *units conversion functions* having the same name as the unit, that takes an argument with any related unit, and converts it to have units specified by the name of the function. It also creates a function that casts unitless values to have the specified unit. 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 relationship 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*) as an attribute in their declarations, or alternatively by separate UNIT declaration statements.

Examples:

```
UNIT :: INCH, SECOND
UNIT :: CM, INCH = 2.54 * CM
UNIT :: CM_PER_INCH = CM / INCH
REAL, PARAMETER, UNIT(CM_PER_INCH) :: CONVERT = 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
REAL, UNIT(CM) :: C
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
C = CM(L)                     ! VALID -- Clearer than the previous statement
L = SQRT(A) * 5.0e-3          ! VALID -- exercise for reader
```

1.2 Improvements to the type system

1.2.1 A more complete type system should be provided

Enumerations that are true types, and their enumerators

References: 98-171r2, 04-125r1 and 04-139r1.

Enumerations that are new types should be provided. 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, enumerators of different types can have the same names. Enumeration types should be extensible, with the extension type inheriting the enumerators of its parent type, and possibly adding more enumerators. MAX and MIN should work for ordered enumeration types.

Subranges of integers

Reference: 04-140r1.

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 that checking of subscript bounds can be done at compile time: If the bounds of an array are specified by a subrange name, and if in that case the only subscripts are 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 constant subrange of 0:1.

1.2.2 New types – not type synonyms – from existing types

Reference: 04-146r1.

For a brief time, there was a work plan for Fortran 2008 to define new type names that are synonyms for existing types. The inspiration for this was the `typedef` statement from C.

For derived types, we already have a similar method: extend an existing type without adding any components. Rather than providing a synonym, this provides a compatible type. There is nothing that can be done using a synonym that cannot be done using a compatible type, but there are important things that can be done using a compatible type that cannot be done using a synonym. The most important of these is generic resolution.

Two simple extensions of this principle would provide all the desired functionality.

1. Allow to extend intrinsic types (but continue to prohibit to extend sequence types). Whether declaration of additional components is allowed, and whether declaration of

type-bound procedures is allowed, can be decided in due course. Specify that an extension of an intrinsic type (perhaps only an extension that does not have additional components) is an intrinsic type. Thereby, descriptions in the standard that refer to the term "intrinsic type" still apply. Further, existing type-compatibility rules also continue to apply.

2. During definition of a type extension, allow to specify a value for a type parameter of the parent type. All objects of the new type, or any extension of it, have that value for that type parameter. A different value for that type parameter cannot be specified in an object declaration, or a further type extension.

To illustrate the use of these two principles combined, consider a procedure to evaluate a special function, say a Fresnel integral, for which approximations that provide different degrees of accuracy have different costs. One might provide extensions of type REAL that have different requested decimal precision, e.g.,

```
  type, extends(real(selected_real_kind(4))) :: Real4
  end type
  type, extends(real(selected_real_kind(6))) :: Real6
  end type
  ...
  type, extends(real(selected_real_kind(14))) :: Real14
  end type
```

In an application that needs four digits of accuracy for some computation and six for others, objects of these types can be used to select an algorithm, by generic resolution, that provides the appropriate balance of performance and accuracy. This is not possible using type synonyms.

Another reason to want compatible types instead of synonyms is that declarations using, e.g. SELECTED_REAL_KIND(6) and SELECTED_REAL_KIND(12) might result in objects of different kinds on some platforms, and the same kinds on other platforms. The latter results in failure to compile generic interfaces in which arguments are distinguished by these type parameters, which in turn inhibits portability.

A symplified syntax, that inherently prevents declaration of additional components, and type bound procedures, could use a keyword different from EXTENDS, e.g.

```
  type, new(real(selected_real_kind(4))) :: Real4
```

or

```
  new type :: Real4 => real(selected_real_kind(4))
```

This syntax might be allowed even if the EXTENDS syntax is also allowed for the purpose of adding type-bound procedures. If it is desired to allow that but to prohibit component declarations, a constraint would suffice.

1.3 Updaters

Reference: Correspondence preceding 1986 Albuquerque meeting, 97-114r2, 04-141r1.

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 (e.g., Mesa, POP-2 and CURL) 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. A reference to one can only appear in a variable-definition context.

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, thereby guaranteeing 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 data object that has INTENT(INOUT) or unspecified intent, copy in / copy out semantics are used.

1.4 Uniform syntax

Reference: Correspondence preceding 1986 Albuquerque meeting.

Uniform referential syntax to array elements, updaters and structure components increases software mutability, and therefore decreases maintenance costs. (The alternative is to follow Parnas's advice to encapsulate everything in procedures — which provides uniform referential syntax with higher labor expense and lower performance). To move in that direction, it ought to be allowed to access a structure component using the syntax *component(structure)*. This should be classed as a generic updater. To avoid a conflict with an array of the same name as *component* an array ought to be considered to be a generic updater. Access to type-bound procedures or procedure pointer components with PASS_OBJ could require the structure to be first, or to be in the same place in the reference as the corresponding dummy argument. For symmetry with this syntax, array components should be referenced as *component(structure,subscript,...)*. A small step in the direction of uniform syntax is to allow type-bound functions to be refenced without parentheses enclosing the empty argument list. An anti-symmetric change to allow a scalar to be referenced with an empty subscript list would make it impossible to know, and impossible for the syntax to depend upon, whether a reference to an entity of a derived-type object is a reference to a component or a type-bound function or updater. To complete the ability to convert an entity between an array, structure component and updater, an integer interval type, using the same constructor as a subscript triplet, is needed to allow an updater reference with the same syntax as an array section reference.

1.5 Exception generation and handling

Reference: 04-154, N1257, August 2016 *Fortran Forum*.

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, or within a WHERE or FORALL statement or construct, or a DO CONCURRENT construct, and the exception is handled outwith the 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). John's proposal could be improved by using enumerators of enumeration types (see 1.2.1) instead of integers to identify exceptions.

1.6 Coroutines

Reference: 04-380r1.

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 allow easy access to extra information in 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*.

A simpler use that has nothing to do with reverse communication, in the sense that the invoked procedure needs the caller to perform computations, is to preserve the activation record for later use. Suppose, for example, that one needs to perform several related computations of different sizes, for which one can compute the maximum storage requirements for intermediate variables. One way to avoid visiting the storage allocator for each variable for each different problem is to use automatic or allocatable variables allocated with the maximum size necessary, and write the code for the different size problems either inline, or in internal subprograms that have access to the automatic or allocatable variables by host association. Another alternative is to pass the variables as arguments. The former can lead to enormous modules; the latter to enormous argument lists. Another alternative is for the variables to be allocatable variables at module scope, but this is not thread safe. Automatic local variables in a coroutine consitute a simpler solution.

Another use is to to specify a coroutine in, e.g., a PROCESS= specifier in a data transfer statement, to process the item list. This would allow to set most of the material concerning defined input/output in obsolescent font, and eventually delete it.

Coroutines can also serve as the foundation for reliable parallel programming. For example, Ada's protected variables are actually coroutines. See, e.g., **Concurrent and Real-Time Programming in Ada** by Alan Burns and Andy Wellings.

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

Proposal

Provide coroutines, which are adequate to allow developing packages that provide for reverse communication, without causing the excruciating mess within the package that is presently required.

In addition to statements to define them, coroutine support would require two executable statements, e.g., SUSPEND and RESUME. If a coroutine 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, and a new activation record is created. Otherwise, the coroutine was previously entered, was last exited by a SUSPEND statement, and is being re-entered by a RESUME statement, in which case control resumes at the next executable statement after the SUSPEND statement, using the existing activation record. 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.

Activation records for coroutines cannot be on the stack. For thread safety they should be carried from CALL to RESUME by the calling scoping unit. Therefore, RESUME can only appear in the same scoping unit as the corresponding CALL, unless CALL and RESUME are both invoked using the same procedure pointer.

It is useful to provide mechanisms to determine whether a coroutine is suspended, so that it doesn't need so to specify to the invoker using an argument or global variable, and to terminate a suspended coroutine (e.g., to reclaim its memory in case the calling procedure decides it doesn't want it to continue), so that it isn't necessary to resume it so it can terminate itself by executing a RETURN statement.

Whether RESUME requires to repeat the actual arguments, or the argument association is automatically maintained while the coroutine is suspended, can be determined later.

Here's an example of simple usage to evaluate an integral.

```
call integrate ( a, b, tolerance, answer, error )
do while ( suspended(integrate) )
  answer = f(answer)
  resume integrate ( a, b, tolerance, answer, error )
end do
```

1.7 Generic programming

References: 05-195, 07-007r2.

A macro system would be quite valuable within Fortran. A macro system was agreed for inclusion in Fortran 2008 (07-007r2), but was removed before development was completed. Either it should be reinstated, or an alternative, such as parameterized modules (05-195), should be developed.

The original motivation for this was to make it easier to construct a derived type with kind parameters, together with a consistent set of type-bound procedures. A way to do this that is less powerful and less flexible than macros or parameterized modules is to allow to define procedures and interfaces within type definitions. This would require the processor to instantiate procedures with the appropriate spectrum of kind type parameters whenever a variable of the type is declared, and preferably to avoid duplicates. Interface blocks within type definitions would provide the same functionality as deferred bindings.

Another alternative that is simpler than macros or parameterized modules is to allow to define a procedure abstraction that has kind parameters. The definition of a subprogram would include a list of names, perhaps after a new word such as ABSTRACT in the prefix. It would be required to declare those names within the body of the subprogram to be integers with the KIND attribute. One would put the abstract subprogram's name, together with constant expressions giving values to its kind parameters, as the *proc-interface* in a PROCEDURE statement. If the POINTER attribute appears, this serves only to specify an explicit interface. In a *procedure-stmt* or *procedure-declaration-stmt* without the POINTER attribute, this creates an instantiation of the abstract subprogram. A *type-bound-procedure-stmt* specifies that an instantiation is to be created where an object of the type is declared.

1.8 An assertion system would have several benefits

References: 04-142, 04-143, 04-144, 04-219, 04-376, 04-414, 04-417r1.

As described in 04-142, 04-143, and 04-144, an assertion system would aid optimization, reduce labor costs, and improve program reliability.

In the system described in the references, an ASSERT statement is executable. An ASSERT declaration that holds throughout the program unit or construct in which it appears would also be useful. The advantage of a declaration as compared to an executable statement is that dataflow analysis (from the ASSERT statement to the situations to which it applies) would not be necessary to gain performance benefits.

Certain "system" considerations cannot easily be described by logical expressions but can have profound impact on performance. One example is "the *variable* and *expr* in intrinsic assignment do not overlap." These could be described using an "intrinsic function," but it would not be a function according to the current definitions in that it is important that its arguments are *not* evaluated, and not even required to be defined (see also 2.7.12.17). Two varieties of such a "function" ought to be provided, one that takes no arguments, and one that takes two arguments and applies only to assignments in which the first is the *variable* and the second is the *expr* in an intrinsic assignment.

1.9 Structured parallelism would assist to exploit modern architectures

References: 97-114r2, 00-317, 06-187.

Define a new execution construct:

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

fork                      is  FORK [ IF ( logical-expr ) ] [ parallel-construct-name ]
                                  [ specification-part ]
                                  block
```

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 that does not have IF (*logical-expr*) or for which *logical-expr* is true is executed exactly once, and ones that have IF (*logical-expr*) and for which *logical-expr* is false are not executed. After all of the required 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, replace FORK statements that have IF (*logical-expr*) with IF constructs, and then ignore the PARALLEL, FORK, and END PARALLEL statements.

This construct is functionally equivalent to a DO CONCURRENT construct with a SELECT CASE construct inside it, but the syntax makes it more obvious, and might make life easier for an optimizer.

OpenMP provides the functionality described here, and more. Some of the additional functionality of OpenMP could be provided by allowing all constructs, or at least a *fork*, not just BLOCK constructs, to have specification parts (see 2.1.1).

The proposal in 97-114r2 is for an ASYNCHRONOUS construct that can begin execution when control reaches its initial statement, but the sequence of execution can be split, and then simultaneously proceed to the first executable construct after the END ASYNCHRONOUS statement. A statement is provided to wait for completion of an ASYNCHRONOUS construct. A modern reincarnation of that proposal ought to allow to declare a variable of EVENT_TYPE in the statement that initiates the construct, and to wait for completion of the construct using the EVENT WAIT statement. See 2.1.6.

In any case, a variable that is not declared within the construct should not be allowed in a variable definition context, except within a CRITICAL construct, which needs to be extended beyond images to sequences of execution. Similarly, an impure procedure can only be invoked from within a CRITICAL construct. An alternative is a new PARALLEL attribute for procedures, which indicates that any alteration to a nonlocal variable or dummy argument occurs within a CRITICAL construct.

2 Useful

2.1 Improvements to constructs

2.1.1 All constructs should be allowed to have specification parts

Reference: 04-155, 08-197, 08-201.

Introduction

A BLOCK construct can have a specification part. Allowing all constructs to have specification parts would be more concise.

Proposal

Allow all constructs to have specification parts. 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, but only for a *nonlabel-do-stmt* or a *block-do-construct*, *viz.* `DO INTEGER::I = 1, 10`, as can be done in a FORALL construct.

For those constructs that have multiple branches, *viz.* IF and SELECT, each branch should have a separate specification part.

2.1.2 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 the cases. This can be done by using extra select case constructs before, within or after the main one.

Proposal

Allow to group some subset of the case blocks in a select case construct. Thereby, one wouldn't need to write the repetitive select case constructs. 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, the processor gets away without automatically constructing an extra select case construct. Example:

```
select case ( expr )
case ( c1 ); ...
block ! alternatively "cases" or "case group" or ...
  stuff before c2 and c3, but only if expr is in one of them.
  The processor replaces the ‘‘block’’ with ‘‘case ( c2, c3)’’,
  then the extra stuff, then another ‘‘select case ( temp )’’
  where ‘‘temp’’ has the value of ‘‘expr’’ in the outer explicit
  select case statement.
  case ( c2 ); ...
  case ( c3 ); ...
end block
  ‘‘end block’’ becomes ‘‘end select’’ in the case of stuff before
  c2 and c3.
  Stuff after c2 or c3.  If there’s no stuff before c2 and 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
```

2.1.3 More general discrete ranges in case selectors

Introduction

Reference: 97-114r2, 04-157.

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

Proposal

Allow the same generality for a *case-selector* as for a *section-triplet*. If integer intervals are provided (1.4), it should be allowed to specify the range in a case selector using a named integer interval constant.

If ordered enumerators are provided (see 1.2.1) a similar syntax should be allowed for them.

2.1.4 REAL ranges in SELECT CASE constructs

Introduction

Reference: 97-114r2.

Arithmetic IF is deprecated with no equivalent that doesn't involve a temporary variable. Problems with temporary variables are described in section 2.9.

Proposal

Allow a REAL *case-expr* in a SELECT CASE statement. Allow a REAL constant in a CASE statement. Extend the *case-value-range* in a CASE statement to allow open or closed ranges, using *expr rel-op* * *rel-op expr*, *expr rel-op* *, or * *rel-op expr*, where *rel-op* can be <. <=, `.lt.` or `.le.`, and * refers to the value of *case-expr* in the SELECT CASE statement. Thereby, `if ( y * (myfunc(a) - cos(z)**2 ) ) 10, 20, 30` can be replaced by

```
  select case ( y * (myfunc(a) - cos(z)**2 ) )
  case ( * < 0.0 ) ! was statement label 10
    ...
  case ( 0.0 )     ! was statement label 20
    ...
  case ( 0.0 < * ) ! was statement label 30
    ...
  end select
```

An alternative is construct-scope variable association (see 2.3.3).

2.1.5 CASE .AND.

Introduction

Reference: 97-114r2.

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, by putting it into a procedure, or by using an auxiliary logical variable to control executing it after the CASE construct. One can't even get there with a GO TO statement.

Proposal

Allow .AND. *scalar-logical-expr* or IF (*scalar-logical-expr*) after the (*case-value-range-list*) on a *case-stmt*, with the meaning that the *block* after that *case-stmt* is executed if the value of the *case-expr* is one of the *case-value*s and the *scalar-logical-expr* is true, and the one after the CASE DEFAULT statement is executed otherwise.

2.1.6 Asynchronous block construct

A BLOCK construct with the ASYNCHRONOUS attribute would allow explicit specification of parallelism opportunities that are not as structured as those that could be accomodated by a PARALLEL or FORK/JOIN construct (1.9). For example, use

```
 BLOCK [, ASYNCHRONOUS (event-variable) ]
...
END BLOCK
```

An event is posted to *event-variable* when execution of the construct is completed.

A variable that is referenced or defined within the construct and that is not a construct entity is a pending storage sequence affector, and shall not be defined or referenced until an EVENT WAIT statement that references the *event-variable* is executed, except by an atomic subroutine.

It is necessary to remove the requirement that an event variable be a coarray, and that the ATOM argument of an atomic subroutine be a coarray. If the definitions of segments are not revised to include sequences before and after EVENT POST and EVENT WAIT statements that do not wait for a coarray *event-variable*, it would continue to be necessary to require the *event-variable* in an EVENT POST statement to be a coarray or a coindexed object.

It might also be useful to allow lock variables and critical sections to synchronize between asynchronous blocks.

2.1.7 More interaction between iterations of DO CONCURRENT

If a variable is defined in an iteration of a DO CONCURRENT construct, it should be allowed to reference or define it by another iteration by using it as the ATOM argument to an atomic subroutine.

It should be allowed to use critical sections to synchronize iterations of a DO CONCURRENT construct.

2.1.8 Procedure pointer or constant selector in ASSOCIATE

Introduction

The reason for the ASSOCIATE construct was to provide abbreviated names to reference objects that otherwise would have long designators, and especially to provide synonyms for designators containing expressions that ought to be identical, even after maintenance. This desire applies equally to procedure pointers and data objects.

The ASSOCIATE construct was modeled on argument association, wherein a procedure pointer or a constant is a perfectly good actual argument.

It makes no sense to prohibit the *selector* in an ASSOCIATE construct to be a procedure pointer or a constant.

A *selector* in a SELECT TYPE construct is required to be polymorphic. This quite reasonably precludes procedure pointers or constants.

Proposal

Allow a procedure pointer or a constant to be a *selector* in an ASSOCIATE construct. As with all *selector*s, it shall be associated with a target. The corresponding *associate-name* is a procedure designator or constant.

2.1.9 Bounds for associate name

Introduction

The lower bounds of an assumed-shape dummy argument can be declared. The lower bounds of a pointer can be specified in pointer assignment. The bounds of a pointer whose target is either rank one or simply contiguous can be specified. Construct association was modeled on argument association and pointer assignment.

Proposal

It should be possible for an *associate-name* to have a *bounds-spec-list* or *bounds-remapping-list*, subject to conditions similar to those in pointer assignment. If a *bounds-spec-list* is specified, the number of *bounds-spec*s shall be equal to the rank of the selector. If a *bounds-remapping-list* is specified, the selector shall be of rank one or shall be simply contiguous.

2.1.10 Construct labels ought to be local to the construct

Reference: 04-158.

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. This is incompatible with the proposal for an intrinsic function that requires how much a loop is unrolled (see 2.7.12.18).

2.1.11 Iterators

An iterator is a special kind of procedure that produces all the elements of a set, one at a time. An iterator is related to a coroutine (1.6) in that it is suspended when it provides a set element, and it is resumed after the point of suspension when another set element is needed. Iterators can only be accessed within the control statement of a loop. An iterator-controlled loop might have a syntax such as ITERATE (*variable* = *iterator-reference*) . . . END ITERATE. Unlike the case of a DO construct, it isn't necessary for *variable* to be scalar, and it can have any type. It could even be polymorphic, or allocatable. It gets its value from the *iterator-reference* as if by an assignment statement, including the possibility of defined assignment.

An iterator procedure would be bracketed by special statements such as ITERATOR ([*dummy-arg-name-list*]) . . . END ITERATOR. Before the first iteration of the loop body, the loop control enters the iterator at its first executable construct. The iterator produces a value by assigning to its result variable (exactly as does a function) and then executing a SUSPEND statement. It indicates that there are no more results by executing a RETURN or END statement. An iterator's activation record would also be destroyed by explicit loop termination as described in subclause 8.1.6.6.4 of 10-007r1.

It would be additionally useful to allow iterators to return tuples, and to allow the *variable* to be a scatter (2.10).

A CONCURRENT ITERATE construct should bw provided. The iterator is resumed as if within a critical section (not a CRITICAL construct as currently defined.)

2.1.12 DO constructs that test at the end would be useful

Reference: 04-159.

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 allowing only END DO.

2.1.13 A subconstruct done on explicit exit would be useful

Reference: 04-160.

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.

Proposal

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

```
do ...
  ...
  if ( <condition-1> ) exit
  ...
  if ( <condition-n> ) 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.

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 Parameter declarations inside of TYPE definitions

Reference: 04-162.

One sometimes needs to declare a parameter for the purpose of declaring components 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 outwith the type, in contexts other than variable definition contexts (subclause 16.6.7 of 12-007), by using component selection notation (provided they are not private). Dynamic parameters or specification variables(2.3.4), that depend on length parameters of the type, would also be useful.

2.2.2 Automatic types

Kind type parameter values of an object have to be constant expressions. Length type parameters cause components whose extents or length parameters depend upon them to behave like allocatable components, except possibly if they are the last component of a structure. There could be a middle ground, at least for types that are defined within subprograms: An entity analogous to automatic variables, but at the type level. In this case, bounds or length parameters of components would be given by specification expressions, which would be accessed within the type definition by host association, and "inherited" by objects of the type. The coefficients necessary to access components of automatic types could be calculated in the same way that processors calculate coefficients necessary to access multidimensional automatic objects, rather than creating arrays of allocatable arrays. As with automatic variables, specification expressions are only evaluated when the procedure is invoked, not, for example, when objects of the type are allocated.

2.2.3 Turn off intrinsic assignment

Reference: 97-114, 03-258r1, 04-163, 04-167, 13-214, 13-348, 14-138r1, 14-139

It is sometimes desirable to prohibit changes to objects of a derived type, except within the scoping unit where the type is defined. Examples include variables of type(LOCK_TYPE), and variables of type(EVENT_TYPE) proposed in TS 18508. This can at present be done for user-defined types 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, can't be done with a PURE subroutine, and doesn't affect variable-definition contexts other than the assignment statement. 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." A variable of limited type should also be prohibited as a dummy argument with the VALUE attribute, or in any other context where it gets a value *as if* by intrinsic assignment, except within the inclusive scope where the type is defined. This would be subsumed by protected types (2.2.7).

It would be similarly useful to be able to delcare that pointer assignment is not defined for objects of specified user-defined types. See 2.6.6.

2.2.4 Extend the type parameter system to types

Reference: 04-164.

The type system would be more useful if a new variety of type parameters, that are types, were provided.

2.2.5 Specify rounding as an attribute

The IEEE modules control rounding, but the answer to interpretation f95/000104 says that a processor can retain intermediate results in representations different from the kind specified by subclause 7.1.9 (of 09-007). There is no specification of *when* a value has to be represented as specified by subclause 7.1.9. Thus there is no requirement to use the same rounding mode to compute a value and later to round it to the representation specified by subclause 7.1.9. If the rounding to be used when a value is assigned to a variable, including to a component or a function result variable, were specified by an attribute, there would be no question.

The attributes might be spelled ROUND(UP) (toward $+\infty$), ROUND(DOWN) (toward $-\infty$), ROUND(ZERO) (toward zero), ROUND(NEAREST), or ROUND(DEFAULT), the latter not being equivalent to no specification. If a rounding mode is not specified, when a value is assigned to a variable it is rounded according to the rounding mode in effect as specified by IEEE_SET_ROUNDING_MODE, which is not guaranteed to be the rounding mode in effect when the value was calculated. If a variable has a specified rounding mode, processors that encode the rounding mode in the instruction, as opposed to a processor status register, could generate more efficient code.

This interacts with operator-specified rounding (2.11.3).

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 that has a length type parameter instead of a kind type parameter 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 (1.2.1) were adopted.

2.2.7 Protected types

Reference: 04-167, 14-165.

Fortran 2008 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. Not appearing in a variable definition is not quite right. They can be the *variable* in a defined assignment (but not an intrinsic assignment), provided the defined assignment is provided by the module where the type is defined. Explicit interface should be required if a dummy argument is of a protected type. Subobjects of them cannot be actual arguments associated with dummy arguments that have unspecified intent, and cannot be actual arguments to procedures with implicit interface. A local pointer could step through a linked list of protected objects without being able to change them. Assignment could be turned off (2.2.3), simply by not providing a defined assignment.

2.2.8 Protected components

Reference: 13-215.

Define a PROTECTED attribute for components of non-sequence types. If a type has a component with the PROTECTED attribute and the type definition is accessed by use association:

- a variable of the type shall not appear in a variable definition context (16.6.7), and
- a component with the PROTECTED attribute, or a subobject thereof, shall not appear in a variable definition context (16.6.7), a pointer association context (16.6.8), or as the *data-target* or *proc-target* in a *pointer-assignment-stmt*.

2.2.9 Allow kind type parameter of PDT in literal constants

A kind type parameter name of a type being defined cannot be used as the *scalar-int-constant-name* in a literal constant that appears within the type. This prohibits using, for example `0.001_RK` as an initializer for a component of real type with kind RK given by a kind type parameter name of a type being defined. This is unhelpful if the initializer cannot be exactly represented in binary (as, for example, 0.001 cannot be exactly represented in binary). One could specify, for example, 0.001d0, but this is not helpful if the processor offers a kind with more precision, as an extension.

2.2.10 Set types

Set types would be useful. The universe for a set type could be specified by an integer range, a named integer subrange (1.2.1), or an ordered enumeration type (1.2.1). Operations on set variables include union, intersection, inquiry whether a value is a member of a set, adding a value to a set, removing a value from a set, set difference, and symmetric set difference. Input and output should be provided. Whether a set universe can be specified by a specification expression, or only by a constant expression, can be decided in due course.

2.2.11 Objects of abstract type

Objects of abstract type exist: the parent component of an extension of an abstract type. It should be possible to access objects of abstract type, or components thereof, but not to invoke a deferred type-bound procedure using one as a *data-ref*. There is no harm in allowing to declare variables of abstract type, so long as it is prohibited to invoke a deferred type-bound procedure using one as a *data-ref*. Move C611 in 15-007r2 to be a constraint on R1223. Require the target of a pointer assignment to be polymorphic if it's type is abstract and the *data-pointer* is polymorphic, so a similar constraint is needed on R737. It also needs to be a constraint on R630 if *allocate-object* is polymorphic. C403 prohibits components of abstract type, but an extension of an abstract type has a component of abstract type. Move C403 to be a constraint on R501 and R626.

2.2.12 Select subobject of function result

Introduction

It would be useful to allow selecting a subobject of a function result, primarily to avoid the necessity of a temporary variable. A high-quality optimizer might exploit this to improve efficiency compared to returning the full object, storing it in a temporary variable, and then extracting a subobject.

Proposal

Add an alternative for *part-ref*:

R612	*part-ref*	**is**	*part-name* [(*section-subscript-list*)] [*image-selector*]
		or	*function-reference* [(*section-subscript-list*)]

If *function-reference* appears and the reference has no actual arguments other than a passed-object dummy argument and *section-subscript-list* appears, parentheses shall appear as part of *function-reference* if the function has optional dummy arguments (2.7.3).

This also has the effect of allowing function composition (2.7.1).

2.2.13 Self component

It is not possible to access a polymorphic object as if it were a nonpolymorphic object of its declared type, even using a SELECT TYPE construct. It is occasionally desirable to do so, especially to initialize the declared-type part of the object. It is possible to initialize the declared-type part of the object one component at a time, but this is tedious if the object has more than a few components.

Define a nonpolymorphic "self" component of each type that has the same name as the type. If a component having the same name as the type is declared, the "self" component is not available. Thereby, if one has a polymorphic variable V of declared type T, one could access the declared-type part of it using V%T.

2.2.14 Dynamic type of ancestor type of polymorphic object

One occasionally wants or needs to access a procedure bound to the type of an ancestor type of the dynamic type of a polymorphic object. This might occur if the overridden procedure produces the same values for the ancestor component as the overriding procedure, but does it more efficiently. Presently, the only way to do so is to use a SELECT TYPE construct, with a CLASS IS block for each interesting dynamic type, and then to use an ancestor component to access the desired procedure. If one adds more extensions of the base type, these need to be specified in additional CLASS IS blocks. An alternative is to add an ANCESTOR (N) block, that causes the associate name to have the type of the N^{th} ancestor of the declared type of the selector. In order to ensure that this is the desired type, the SELECT TYPE construct containing the ANCESTOR (N) block should be within a CLASS IS block that specifies the expected dynamic type of the selector, or an intrinsic function, say TYPE_DEPTH, should be provided, having a result that is the number of extensions from the declared type of the argument to its dynamic type. An alternative to ANCESTOR (N) is DESCENDANT (N), which selects the type that is the N^{th} extension of the declared type of the selector, on the type-extension path to the dynamic type of the selector.

2.2.15 ASYNCHRONOUS attribute for components

It would be helpful if a component of a type could be declared to be ASYNCHRONOUS in the type definition, and a component of an object of declared type could be declared to be ASYNCHRONOUS using an ASYNCHRONOUS statement.

2.2.16 Describe COMPLEX as a sequence type

The standard might be simplified if COMPLEX were described as an intrinsic sequence derived type.

2.2.17 Delete comma from * char-length selector

The comma after * *char-length* in a *char-selector* is an anacronism that could be deleted.

2.2.18 Allow VALUE for the passed-object dummy argument

There appears to be no reason to prohibit the passed-object dummy argument to have the VALUE attribute.

2.3 Improvements to variables

2.3.1 Conditional declarations

One occasionally needs a temporary array with automatic parameters or bounds that depend upon optional arguments. It is possible to enclose declarations or those sorts of variables within BLOCK constructs within IF constructs (at the expense of two more levels of indentation if that's your style), but because these constructs have to be properly nested, this strategy quickly becomes intractable as the number of optional variables increases. One might be tempted to try to use MERGE to set a length or bound to zero if an optional argument is absent, but this doesn't work because the arguments have to be evaluated before the MERGE function is invoked, and it violates the definition of restricted expressions (see 2.3.5.1). If one really wants a scalar, say one of derived type that might require a lot of memory, declaring it to be a rank-one array of extent either zero or one causes problems.

Conditional expressions (2.11.1) offer a verbose solution that can be used to create zero-size automatic objects (again see 2.3.5.1). A more terse alternative is an "attribute" that specifies whether a declaration of a local variable of a subprogram is to be respected, say IF(*logical-expr*) or COND(*logical-expr*).

Example: `INTEGER, IF(present(A)) :: PERMUTE(size(A))`

An alternative to IF(*logical-expr*) as the attribute is OPTIONAL(*logical-expr*).

The rules about when they can be referenced would be the same as for optional dummy arguments. The PRESENT intrinsic should be applicable to them. If they appear as actual arguments associated with optional dummy arguments, they behave as if they were optional dummy arguments.

2.3.2 Initially-allocated allocatable arrays

Within a subprogram, allow an explicit-shape non-dummy variable, whose length type parameters, if any, are not deferred, to have the ALLOCATABLE attribute. When the procedure is invoked, it is initially allocated with the specified bounds and length type parameters, and with dynamic type the same as declared type. When and if it is deallocated, its shape becomes deferred and any length type parameters whose values are given by specification expressions become deferred. Thereafter it behaves in all ways like an ordinary allocatable variable. If it has the SAVE attribute, the specified dimensions and length parameters are only used the first time the procedure is invoked.

2.3.3 Statement- or construct-scope variable association

Reference: 97-114r2.

One sometimes has a subexpression that appears several times within a statement. One might 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 in section 2.9. By allowing to create a variable that has statement scope, these problems would be avoided. It behaves like an associate name in an associate construct: Its type, type parameters, shape, and whether it is allowed to appear in a variable-definition context would be taken from the expression that defines it. For example, one might replace:

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

or

`a(3*i+1) = a(3*i+1) + 1` by `lhs @ a(3*i+1) = lhs + 1`

I have indicated that statement-scope variables be defined with a special character, "`@`" in the example. The second example illustrates that it is not simply a value. It would be reasonable to require its "declaration" (i.e., within the statement) to appear before any references to it.

Construct-scope association would be useful in block IF constructs:

```
  if ( ( v @ ( y * ( myfunc(a) - cos(z)**2 ) ) ) < 0.0 ) then
    ... including references to v
  else if ( v == 0.0 ) then
    ... including references to v
  else ! v > 0.0
    ... including references to v
  end if
```

and similarly within SELECT CASE constructs.

2.3.4 Dynamic parameters a.k.a. specification variables

Reference: 04-200.

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, or a class of variables, identified explicitly by an attribute, say SPECIFICATION, whose values are given by specification rather than constant expressions. That is, their values can depend upon other entities in exactly the same way that dimensions and length parameters can. Such entities can obviously be allowed only within procedures. If they're called dynamic parameters, it should be prohibited for them to appear in variable definition contexts. Ordinary variables that get a new initial value on every invocation (see 2.3.6) could serve this purpose, with a restriction that they are declared with an initial value before they appear in a specification expression.

2.3.5 Less restrictive restricted expressions

Reference: 13-208r1.

2.3.5.1 Inquiry functions arguments

The definition of restricted expression prohibits inquiry functions to reference optional arguments. This is a pointless restriction. Augment item (9)(b) of 7.1.11p2:

(9) a specification inquiry where each designator or function argument is ...

  (b) a variable whose properties inquired about are not

    (i) dependent on the upper bound of the last dimension of an assumed-size array,

    (i′) dependent on the association status or allocation status of a dummy argument with the OPTIONAL or INTENT(OUT) attribute,

    (i″) dependent on the bounds or extents of an assumed-shape dummy argument with the OPTIONAL attribute,

    (i‴) dependent on the dynamic type of a polymorphic dummy argument with the OPTIONAL attribute,

The rules concerning absent dummy arguments (items (4) and (9) in 12.5.2.12p2) prohibit an absent dummy argument from being an argument to an intrinsic function other than PRESENT. It would seem to be harmless to allow it to be the argument of an inquiry function, provided the properties inquired are not assumed or deferred bounds or length parameter values.

2.3.5.2 Optional arguments for specification functions

An optional dummy argument cannot be an actual argument to a specification function, even if the corresponding dummy argument of the specification function is optional. Specification functions are required to be pure, and pure procedures are required to have explicit interface in contexts that require references to pure procedures. Compile-time checkability and production of diagnostics would not be measurably compromised by allowing an optional dummy argument to be an actual argument corresponding to an optional dummy argument of a specification function. Replace item (11) of 7.1.11p2:

(11) a reference to a specification function, where each actual argument that corresponds to a nonoptional dummy argument is a restricted expression, and each actual argument that corresponds to an optional dummy argument is a restricted expression or an optional dummy argument,

2.3.5.3 Less restrictive specification functions

7.1.11p5 in 10-007r1 prohibits a specification function from being an internal function, and from having a dummy procedure argument. It would be sufficient to prohibit a specification function from being internal to the scoping unit containing a reference to it, and to require that if it has a dummy procedure argument, the associated actual argument not be internal to the scoping unit containing a reference to the function to which it is an argument.

2.3.6 Initial values on every invocation

Reference: 04-201.

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.

Proposal

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. If the variable does have the pointer attribute, allow it to be what Fortran 2008 allows plus local variables that have the target attribute, and allow bounds in the target to be specification expressions. This attribute, and initialization in the presence of this attribute, ought to be inconsistent with the SAVE attribute. See also the proposal for specification variables (2.3.4).

2.3.7 Allow absent optional arguments to be used

If a subprogram uses a default value when an optional argument is absent, and the value of the argument when it's present, it is necessary to declare an auxiliary variable, and either copy the argument value to it if the argument is present, or give both the argument and auxiliary variable the TARGET attribute and associate a pointer with one or the other of them. It would be simpler if there were an attribute that could be applied to optional arguments that do not have assumed shape or assumed length parameters that creates a local variable with the same names and characteristics.

2.3.8 Relax restrictions on pointer part-names

C618 in 10-007r1 prohibits a *part-name* with the pointer or allocatable attribute from appearing after an array *part-ref* within a *data-ref*. This could harmlessly be relaxed to prohibiting an array *part-ref* with the pointer or allocatable attribute from appearing after an array *part-ref*, which is potentially as irregular as a vector-subscripted array; therefore, it would be reasonable to apply the same restrictions as for vector subscripts in 6.5.3.3.2.

2.3.9 Extensions to the VALUE attribute

A dummy argument with the VALUE attribute is a local variable that gets its initial value from the corresponding actual argument. It therefore ought to work as much as possible like a local variable, other than having an initial value.

There is no problem for a dummy argument with the VALUE attribute to be a pointer: It gets its initial pointer association status from the corresponding actual argument, if the actual argument is a pointer, or becomes pointer associated with the actual argument if the actual argument is not a pointer.

There is no problem for a dummy argument with the VALUE attribute to be allocatable: If the actual argument is not allocatable the dummy argument is initially allocated and gets its value from the corresponding actual argument. If the actual argument is allocatable, the dummy argument gets its initial allocation status and value (if allocated) from the corresponding actual argument.

There is no problem for a dummy argument with the VALUE attribute to be optional (except maybe for BIND(C) procedures), and yet to exist even if the corresponding actual argument is absent, unless it has assumed shape. If it is absent and neither allocatable nor a pointer its initial value – except for default initialized subcomponents – is undefined (but that doesn't prevent the procedure from defining it). If it is absent and allocatable it is initially not allocated (but that doesn't prevent the procedure from allocating it). If it is absent and a pointer its initial pointer association status is undefined (but that doesn't prevent the procedure from defining its pointer association status).

There is no problem for a dummy argument with the VALUE attribute to be VOLATILE. Its volatility commences after it gets its value.

2.3.10 Assumed length parameter, implied shape

We allow an asterisk for a length parameter for a dummy argument or for a named constant. In the case of a named constant, this means "get the length from the data." This would be useful for variables as well as for named constants. One use is the case of an error message that gets something put into it at a known place, or a place found with INDEX. What one currently does is to define a named constant with the desired value, taking its length from the value, and then define a variable for which both the length and value are taken from the named constant:

```
  character(len=*), parameter :: Message_K = "The message initial value"
  character(len=len(Message_K)) :: Message = Message_K
```

Another use is the case of the desire to have what is effectively a named constant, but with the TARGET and SAVE attributes, so that it can be the actual argument that is associated with a dummy argument that has the TARGET attribute, and thereafter the target of a pointer that one wishes not to become undefined when the referenced procedure returns.

We allow an asterisk as an upper bound in an *implied-shape-spec* for a named constant, to mean "get the extents from the data." This is probably not as useful as for length parameters, as described in the previous paragraphs, but if the extension to variables is made for length parameters, this ought to be provided as well, for consistency.

2.3.11 Allow default-initialized variables in DATA

If one has an array having a derived type, and most of the elements ought to have a default value, but others ought to have specific values, it would be useful to allow objects of default-initialized types to appear in DATA statements. The intent is that the specification in the DATA statement overrides default initialization. Without this, one has two alternatives: a first-time flag, and assignment statements for the elements that need a value other than the default value, or explicit initialization for all elements. Neither one is as clean as a few DATA statements. Interpretation F08/0062 established that it would be onerous to ask a processor to initialize part of a structure using default initialization, and part using a DATA statement. For consistency with that interpretation, it would not be unreasonable to prohibit a subcomponent of an object of a type that has default initialization to appear in a *data-stmt-object-list*.

2.3.12 Make COMMON always SAVE

If COMMON is not removed, the standard could be simplified a little bit if COMMON blocks and variables in COMMON always have the SAVE attribute.

2.3.13 Allow nondefault character names for BIND entities

The C standard has a mechanism to allow external names to consist of characters that are not in ISO 8859-1. Remove the requirement that that the NAME in a *language-binding-spec* is of default character kind.

2.3.14 Allow default-initialized variables in DATA

There is no reason to prohibit to initialize default-initialized variables in DATA statements. Simply specify that the *data-stmt-value* takes precedence over default initialization.

2.4 Improvements to array system

2.4.1 Extension to subscript

Reference: 04-195.

Allow an array A of rank $r > 1$ to be subscripted with a single subscript S with extents $[r, n_1, ..., n_k]$, or a rank-one array A to be subscripted by a single subscript S with extents $[n_1, ..., n_k]$ (not $[1, n_1, ..., n_k]$). In all cases, the shape of the result is $[n_1, ..., n_k]$. Fortran 2008 has only the $r = 1, k = 0$ and $r = 1, k = 1$ cases.

If $r > 1$, the elements of the rank-one, extent r, sections in the first dimension of S are used consecutively as subscripts for A. 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 $[n_1, n_1, ..., n_1]$ (in the case S is of rank 2).

Example:

Assume A3(:,:,:) and `S3 = reshape( [(i,i=3, 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 (but see 2.10).

This is different from `A3(S3(1,:),S3(2,:),S3(3,:)) = A3([3,6],[4,7],[5,8])`, 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.

If $k = 0$, S has extent $[r]$ and its elements are treated as the subscripts of A, resulting in accessing a single element of A. The result is a scalar, not an array of extent [1], or an array of extent [1,1,...,1]. This makes the result values of FINDLOC, MAXLOC and MINLOC directly usable as subscripts, i.e., without first being assigned to an array, whose elements are then used individually.

Example: Assume `S3 = [ 3, 4, 5 ]`. Then `A3(S3)` is `A3(3,4,5)`, not `A3(3:3,4:4,5:5)`.

The $r = 1$ case would be useful to compute THRESHOLD in the Ising model in C.13.2.3.3p9: `THRESHOLD = P(COUNT)`.

2.4.2 Extensions to array bounds declaration

2.4.2.1 An array as a bounds specification

Reference: 04-196, 03-216, 03-224, 03-240.

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 a rank r array to be given by a rank-1, extent r, array, in the declaration of an explicit-shape array, in an ALLOCATE statement, and in place of the *bounds-spec-list* or either *...-bound-expr* in a *bounds-remapping*. If the upper bound is given by an array, the lower bound has to be default or given by a conformable expression. If the lower bound is given by an array, the upper bound has to be given by a conformable expression.

This is related to extensions to subscripts (2.4.1).

Allow the cobounds of a corank c coarray to be given by a rank-1, extent $c - 1$, array, both in the declaration of a coarray and in an ALLOCATE statement. If the upper cobound is given by an array, the lower cobound has to be default or given by a conformable expression. If the lower cobound is given by an array, the upper cobound has to be given by a conformable expression.

2.4.2.2 Any combination of assumed and explicit shape

Reference: 04-197r1.

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 every reference. The declaration would also imply contiguity in the first two dimensions (but see 2.4.2.3), which the reference `incoptdepth(1:2,1:2,j)` does not. 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. 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.

2.4.2.3 Uncouple bounds specification from contiguity

Reference: 04-198.

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 dummy argument is not assumed to be contiguous in that dimension – even if the *high-bound* expression appears. If the *stride* does appear, it shall be a constant expression with the value 1, implying the dummy argument is contiguous in that dimension.

Examples:

```
subroutine S ( A, B, C )
  real :: A( :, : )
  real :: B( :size(a,1):, : ) ! Doesn't require contiguous elements
  real :: C( ::1, : )         ! Requires elements in each column to be
                              ! contiguous, but the first element in a
                              ! column need not be contiguous to the
                              ! last element in the previous column.
  ...
```

It probably makes sense to prohibit a contiguous dimension after one that isn't declared to be contiguous.

2.4.3 Conversions between array element order and subscripts

It would be useful to have intrinsic functions to convert between array element order and subscripts. One function would have an argument that consists of an array element reference. Its result would be the array element order of that element. Another would have one argument that is an array and another that is the array element order position of an element. Its result would be a rank-one array having an extent equal to the rank of the array argument, with its value consisting of the subscripts that correspond to that array element order position. The value of the array need not be defined in either case. Similar functions (IMAGE_INDEX and THIS_IMAGE) are already provided for the relationship between cosubscripts and an image index. IMAGE_INDEX should have accepted a coindexed object instead of a coarray and an array of cosubscripts. THIS_IMAGE should accept an argument that specifies an image index rather than only calculating the cosubscripts for the invoking image. Unfortunately, because the definition of THIS_IMAGE was changed from two functions, one having a COARRAY argument and an optional DIM argument, to three functions, this is not possible, so another function is needed to provide the cosubscripts for a specified image and coarray.

2.4.4 More general rank remappings

Reference: 04-199.

Fortran 2008 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* is contiguous or 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 less 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 2.4.2.2 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`.

2.4.5 SIZE(disassociated) = SIZE(deallocated) = SIZE(absent) = 0

An alternative to creating automatic array variables conditionally (assuming conditional declarations such as described in section 2.3.1 are not done) is to simplify creating them with zero size by defining the SIZE inquiry intrinsic function to return zero if its first argument is a disassociated pointer, deallocated allocatable variable, or absent optional argument. As a companion to this, the values of all elements of the array returned by the SHAPE intrinsic function ought to be zero if its first argument is a disassociated pointer, deallocated allocatable variable, or absent optional argument. It might additionally be useful if UBOUND were to return values one less than the corresponding result values of LBOUND, to indicate zero extent in each dimension.

2.5 Avoiding and taming pointers

If one wishes to view a rank-one array as a higher-rank array, one needs to give the rank-one array the TARGET attribute and associate a pointer with it. If one wants to use a particular part of an array, repeatedly, but not drag around the subscripts, one gives the array the TARGET attribute and associates a pointer with the desired part. Both practices subvert optimizers.

2.5.1 Range attribute

Reference: S8.99 (April 1986).

Allow to give an array a RANGE attribute that means an executable RANGE statement can specify the effective bounds when a dimension doesn't have a subscript. If a range has been specified for an array, accessing it with a subscript that is outside the range is an error, even if the subscript is within the bounds of the array. Provide intrinsic functions to inquire the range bounds, size and shape. This requires a descriptor that is equivalent to a pointer or allocatable descriptor for nonpointer nonallocatable arrays, and an extra descriptor for pointer, allocatable or dummy argument arrays, but the adverse impact on quality of optimization is probably substantially less than the POINTER and TARGET attributes. The RANGE attribute isn't argument associated (but it is USE or host associated): If an actual argument has the RANGE attribute, the dummy argument doesn't automatically get it. If the dummy argument has the RANGE attribute, the actual argument doesn't have to have it. If they both have the RANGE attribute, setting the range for the dummy argument does not change the range of the actual argument. If the actual argument has a range and the dummy argument has assumed shape, the extents of the dummy argument are taken from the actual argument's range, not its shape. Example: executing `RANGE(A(1:2,1:2,:))` says that when `A(:,:,I)` is referenced the object is a 2×2 array. Where `A` is a dummy argument, this is potentially useful if the optimizer can use a dataflow analysis to follow constant expressions from the RANGE statement to references. Additional possibility: Allow bounds in the RANGE attribute for a dummy argument, which have the same effect as execution of a RANGE statement. Whether the effect of executing a RANGE statement within a BLOCK has a dynamic scope of the block is TBD. This would require saving ranges of variables that appear in RANGE statements in a BLOCK and restoring their ranges upon exit, or using a different RANGE descriptor within the BLOCK. Whether an associate name has the RANGE attribute if the selector has it is TBD. If it does, whether setting the range of the associate name affects the selector is TBD.

2.5.2 Views

If one could specify that a variable is a view of a subobject or another variable of the same type and kind, one could have some of the functionality of pointers without their adverse impacts on optimization. The view variable has a rank no less than the viewed object, which might be an array section of lower rank than the viewed variable. If the rank of the view variable is greater than the rank of the viewed object, the viewed object shall be contiguous.

A view component that can view another component of the same object is useful to avoid pointer components or the TARGET attribute. If the component it views is allocatable the view component is considered not to be allocated until an executable statement specifies the view, for example the bounds, as if in a rank-remapping pointer assignment, if the view is of a higher rank than the viewed component, or a section if the view is of a lower rank.

The similarity to EQUIVALENCE is that the relationship between view variable names and viewed variable names is a specification. The differences from EQUIVALENCE include that the types and type parameters shall be the same, a view variable can view a dummy argument, an automatic variable, maybe an allocatable variable (TBD), and maybe a pointer's target (TBD), and the viewed object can have bounds given by specification expressions. This poses less difficulty for optimizers than a pointer and a variable with the TARGET attribute, because one knows that a view variable only views one viewed variable, and a viewed variable is viewed by a specified set of view variables (unless one is nutty enough to give it the TARGET attribute). View variables should be allowed to have the RANGE attribute (2.5.1). If a viewed object is a variable that has a range (i.e., not a section), the view variable's bounds are taken from the range. Whether the viewed object can be a coarray is TBD (a selector in an ASSOCIATE construct cannot be).

Example: in

```
real, intent(in) ::  A(:,:)
view(a(:,i)) ::  A_COL(:)  !  Maybe (:)  isn't necessary
```

A_COL views the I'th column of A. I has to meet the requirements for a specification expression. Changes to the value of I after the *specification-part* is elaborated do not affect which column of A is viewed by A_COL.

Attributes of a VIEW object are related to attributes of viewed objects in the same way as for associate names and selectors.

A VIEW declaration in a BLOCK construct is equivalent to an associate name in an ASSOCIATE construct. One could get the effect of VIEW variables in program units by putting the entire *execution-part* inside an ASSOCIATE construct (demonstrating that there is not tremendous additional implementation difficulty), but this strategy becomes intractable if applied to a large number of variables, unless the limit on the number of lines in a statement is removed.

2.5.3 LIMITED or PROTECTED attribute for pointers

If one has an INTENT(IN) dummy argument, one ought not to associate a pointer with that argument or a subobject of it, or with the target of a pointer component of it, and then use that pointer in a variable-definition context; this is in fact prohibited in a pure procedure. A LIMITED or PROTECTED attribute for pointers would mean "This pointer cannot appear in a variable-definition context." It would also mean its association status cannot be copied to a pointer without the attribute. The constraints on pointer associations within pure procedures could thereby be relaxed to require this attribute, rather than prohibiting pointer association entirely.

2.5.4 Coordinating pointers and targets

Optimizers take advantage of the absence of the TARGET attribute to determine that a variable can't be changed by way of a pointer. 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* in a pointer assignment statement, or an actual argument associated with a pointer dummy argument that does not have INTENT(IN), would be useful.

2.5.5 Reference-counted pointers

An attribute of pointers that indicates the target has a reference counter would be useful. In a pointer assignment or argument association, the target or corresponding dummy argument would be required to have the attribute. Such an attribute is not useful for non-pointer targets, so the only ways such a pointer gets a target are by allocation and pointer assignment. Array sections don't have the attribute. Counted pointers spring into existence disassociated, not undefined. Like allocatable variables, their association status is never undefined. If they have a target, and their association status is changed, the count of the target is decremented, and if zero, it is deallocated. When a counted pointer is allocated, its target's count is set to 1. After pointer assignment, if the pointer is not disassociated, the count of its target is incremented.

2.6 Improvements to generic system

2.6.1 Optional dummy arguments for assignment or operator procedures

Reference: 04-169, 13-213.

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.

The requirement in 12.4.3.5.2 in 10-007r2 that the procedure that defines a defined operation have exectly two nonoptional dummy arguments prevents both of these uses.

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 it has 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. If a procedure has at least two arguments, and the second is not optional, it can only define a binary operation.

The generic resolution rules already handle optional arguments correctly.

2.6.2 Generic specifications as partial applications

Reference: 04-168r1.

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.

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 (and it increases code bulk).

Proposal

Allow actual argument values to be specified by constant expressions for some dummy 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 actual arguments are specified, the dummy arguments for which no value is specified for the corresponding actual argument shall satisfy the present requirements. In the functional programming community, this is called "partial application" or "Currying" (after Haskell Curry).

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
```

Since these interfaces define operators, it would be necessary that after specifying values to use for some arguments, in the interface, there remain one or two arguments for which values are not specified, and these arguments meet the present requirements for defined-operator interfaces.

2.6.3 Compound assignment/operation generics

Reference: 04-171r1.

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 (at least — see 2.6.1) 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 2.6.1 and 2.6.2, which would have impact on this specification.

2.6.4 Procedure declaration statement in generic definitions

It would be useful to allow a *procedure-declaration-stmt* in an *interface-block*, provided it specifies explicit interface.

2.6.5 Distfix defined operations

Introduction

Some operations have three operands. For example, an inline if-then-else operation, having functionality similar to the MERGE intrinsic function, would have three operands, and might be written *logical-expr* .THEN. *expr* .ELSE. *expr*, where the second and third *expr* have the same type, type parameters, and rank.

Proposal

Specify the syntax and precedence within expressions, in clause 7, for distfix defined operators. Specify the syntax to declare a distfix defined operation, for example by allowing two defined operators in the INTERFACE statement. Specify that a function that defines a distfix operation have at least three arguments, with any after the third optional. See also 2.6.1.

2.6.6 Define pointer assignment

Reference: 04-175.

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, including type-bound 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. It is necessary, however, to be able to control where intrinsic pointer assignment works because otherwise the ultimately necessary pointer assignment cannot be done.

This is related to limiting where intrinsic assignment works (2.2.3), and reference-counted pointers (2.5.5).

2.6.7 KIND arguments whose values are used for generic resolution

See 2.7.14.

2.6.8 Resolve generic without invoking a procedure or evaluating arguments

See 2.7.15.

2.7 Procedure improvements

2.7.1 Invoke type-bound function or function pointer component using a type constructor or the result of a function reference

A function reference of the form `f2(f1(x))` is allowed. References of the form `x%f1()%f2()` or `f1(x)%f2()`, i.e., equivalent compositions of function references, ought to be allowed.

See also 2.2.12

2.7.2 Allow type-bound procedure to have array passed-object argument

If a type-bound procedure has a passed-object dummy argument, it is required to be a scalar. This requirement appears not to serve a useful purpose. There are cases, such as sparse matrix-vector multiply, that cannot be posed as elemental operations.

2.7.3 Reference a type-bound function or function pointer component without parentheses

Allow to reference a type-bound function, or a function pointer component, without parentheses if it has no actual arguments other than the passed-object dummy argument, or none at all if it has no passed-object dummy argument. This precludes 2.7.4 because it would be ambiguous whether the appearance of `x%f` as an actual argument is the function or a reference to it if the interface of the referenced procedure is implicit or the dummy argument corresponding to `x%f` is a function or function pointer for which the result is a function having the same characteristics as `x%f`.

2.7.4 Allow type-bound procedure as actual argument or procedure pointer target

Reference: 13-219. See also 2.7.12.1. This precludes 2.7.3 in some cases.

Introduction

Neither *proc-target* nor *actual-argument* is allowed to be a type-bound procedure, denoted by *data-ref*%*binding-name*. This makes it impossible to use a procedure that is accessible only as a type-bound procedure as an actual argument or a procedure pointer target.

Proposal

Replace *procedure-name* and *proc-component-ref* in R1223 with *procedure-designator*, with appropriate adjustment of C1235. Replace *procedure-name* and *proc-component-ref* in R740 with *procedure-designator*.

2.7.5 Allow actual argument corresponding to polymorphic dummy to be type compatible

Introduction

It is unhelpful that the actual argument corresponding to a polymorphic dummy argument is required to be polymorphic, and that they have the same declared types or both be unlimited polymorphic.

Proposal

Do not require an actual argument that corresponds to a polymorphic dummy argument to be polymorphic. This affects only allocation.

Allow the declared type of an actual argument that corresponds to a polymorphic dummy argument to be type compatible with the declared type of the dummy argument rather than requiring they have the same declared types. This is already the case for nonpolymorphic arguments, so nothing new need be said. This affects only allocation.

If no *type-spec* appears in an ALLOCATE statement for a polymorphic dummy argument, it is allocated with the declared type of the actual argument. If a *type-spec* appears it shall specify a type that is type compatible with the declared type of the actual argument if the actual argument is polymorphic, or of the same declared type otherwise. A *type-spec* shall appear if the actual argument is unlimited polymorphic.

2.7.6 Invoke elemental subroutine within WHERE construct or statement

Allow to invoke an elemental subroutine within a WHERE construct or statement provided the shape of each array argument is the same as the *mask-expr*. A constraint should require that every array argument have the same rank as the *mask-expr*.

2.7.7 COPY mode of argument association

Provide a COPY mode of argument association. With INTENT(IN) it's the same as VALUE; with INTENT(OUT) it's the same as copy-out. Otherwise it's the same as copy-in/copy-out. Before copying out and after the procedure completes execution, the designator is determined. This means that subscripts, length parameter values, and function references within the actual argument designator are (re)evaluated after the procedure completes execution. If a function result has the COPY attribute, evaluation of the function is allowed to affect other entities in the statement wherein it is invoked.

2.7.8 Specify arguments to which elementality applies

Allow to specify attributes, say SCALAR and ELEMENTAL, for elemental procedure dummy arguments that control which ones have to have a specified rank, and which ones are elemental in the current sense. If the procedure is a function, its shape is the same as the elemental actual arguments of maximum rank.

Example:

```
elemental function F ( N, P, X )
  integer, intent(in), scalar :: N
  real, intent(in) :: P(n+3) ! or P(:)
  real, intent(in), elemental :: X
end function F
```

A reference of the form `y(1:m) = f ( n, p(1:n+3), x(1:m) )` would compute the same thing as `forall(i=1:m) y(i) = f ( n, p(1:n+3), x(i) )`

The SCALAR and ELEMENTAL attributes can be specified only for dummy arguments of elemental procedures. Actual arguments corresponding to SCALAR dummy arguments have to be scalar. Actual arguments corresponding to ELEMENTAL dummy arguments all have to have compatible shape. Actual arguments corresponding to dummy arguments with the dimension attribute have to conform in the usual nonelemental way.

2.7.9 Improvements in usability of optional arguments

See Section 2.6.1 about optional dummy arguments for assignment or operator procedures, 2.11.1 about distfix if-then-else operators, and 2.3.5.2 about optional arguments for specification functions.

2.7.9.1 Default initial value for absent optional argument

Reference: 04-179.

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 specification variables (2.3.4) or reinitialization on every invocation (2.3.6) is adopted. Since the proposal for non-null initial targets for pointers was 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 (2.11.1) would go some distance toward solving the same problems as this proposal, but it would still require an auxiliary named local variable.

This would interact with the proposal for specification variables in 2.3.4, and possibly with extensions to the VALUE attribute in 2.3.9.

2.7.9.2 Optional arguments for final subroutines

There doesn't seem to be a good reason not to allow final subroutines to have optional dummy arguments. Instead of requiring exactly one nonpolymorphic, nonpointer, nonallocatable argument of the type, allow any number of arguments, with at most one nonoptional. Where bound as a final subroutine, the first nonpolymorphic, nonpointer, nonallocatable argument of the type is the one that is finalized, and all the others are required to be optional. This would allow a final subroutine to be a type-bound subroutine that could be used explicitly to force finalization before the object becomes undefined (which requires its passed-object argument to be polymorphic), or to be used for more than one type.

2.7.9.3 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 absent optional dummy argument as a subscript in an actual argument, with the effect that the corresponding dummy argument is considered to 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 (2.11.1) and allowing disassociated pointers as actual arguments associated with optional nonpointer dummy arguments 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.

2.7.9.4 Absent optional arguments in statement specifiers

Reference: 04-180.

When an absent optional dummy argument is used with an optional statement specifier in an input/output control list, or an ALLOCATE, DEALLOCATE, SYNC, or LOCK 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.

2.7.9.5 Absent optional argument as target gives null pointer

In Fortran 2008 a disassociated pointer can be associated with an optional nonpointer nonallocatable dummy argument, in which case the dummy argument is considered not to be present.

To complete this picture, allow an absent optional nonpointer dummy argument with the TARGET attribute to be the target in a pointer assignment statement, in which case the pointer becomes disassociated.

2.7.9.6 Absent optional argument in I/O lists

If an absent optional argument is an *input-item* or *output-item* the effect is as if the item had not appeared.

2.7.9.7 Optional dummy argument as actual argument to several intrinsics

Reference: 10-187r2.

Several intrinsic functions, e.g. IALL, have a nonoptional DIM argument that is prohibited to be an optional dummy argument. Interpretation F08/0038 agrees the restrictions are pointless, but failed at the WG5 level because it was viewed as a feature change, not an error correction.

2.7.10 Lifetime of procedure pointer and host instance of its target

Reference: 16-007r2.

It was proposed during the development of Fortran 2003 that the lifetime of the host instance of a procedure pointer could have been constrained not to exceed the lifetime of the host instance of its target, rather than becoming undefined when the host instance of its target completes execution.

[Somewhere in Clause 8 or 19]

"The scope where a data entity or procedure pointer is established is the scope where it is declared but it is not a dummy argument or accessed by use or host association, and is not a potential subobject component of an object that is a dummy argument or accessed by use or host association."

[173:7+ C1030+]

C1030a (R1033) If the *proc-pointer-object* is established other than in an internal procedure or the inclusive scope containing the *pointer-assignment-stmt*, and an internal procedure is a *proc-target* in the inclusive scope containing the *pointer-assignment-stmt* or any inclusive scope contained within or containing that scope, then the *proc-target* shall not be an internal procedure, and shall not be a procedure pointer that is established within an internal procedure or within an inclusive scope that contains an internal procedure.

C1030a prohibits one harmless but potentially useful case. Suppose A is a constrained pointer, B and C are pointers that are not constrained, D is a module procedure, and E is an internal procedure. Then

```
B => D ! is OK because B is not constrained.
C => E ! is OK because C is not constrained.
A => B ! is harmless because B is associated with D, but is prohibited
       ! because B is local and E appears as a <proc-target> in the
       ! same scope.  A dataflow analysis might prove A cannot become
```

```
        ! associated with E in this particular case, but not in all
        ! cases, for example if a <proc-pointer> is a subobject of a
        ! data target or an array element.
```

A similar constraint for a *data-pointer-object* would be overly restrictive, and therefore harmful, because the *data-target* can be allocated, and therefore would not necessarily be outlived by the *data-pointer-object*.

2.7.11 Deeper nesting of internal procedures

It would be useful to allow internal procedures to have internal procedures, with a maximum nesting depth specified to a ridiculously large value (say, 32) by the standard.

2.7.12 Intrinsic procedures

Reference: 13-219. See also 2.7.4.

2.7.12.1 More liberal actual arguments for inquiry functions

If one has a system of types connected either by allocatability, pointers, or embedding, one sometimes cannot use inquiries that seem not to cause any operational problem. Suppose `x` has an array or pointer component `c`, which in turn has an array or pointer component `b`, which in turn has an array or pointer component `a`. One cannot inquire `kind(x%c%b%a)` if both `b` and `c` are arrays, or if either one is deallocated or disassociated. Such inquiries would be possible if C618 in 10-007r1 were revised

C618 (R611) Except as an actual argument to the intrinsic functions BIT_SIZE, DIGITS, EPSILON, HUGE, KIND, NULL, PRECISION, RADIX, RANGE, or TINY, or the inquiry functions in the intrinsic module IEEE_ARITHMETIC, there shall not be more than one *part-ref* with nonzero rank, and a *part-ref* to the right of a *part-ref* with nonzero rank shall not have the ALLOCATABLE or POINTER attribute.

and if a sentence were inserted in 13.1p2

> In the argument to any of the intrinsic functions BIT_SIZE, DIGITS, EPSILON, HUGE, KIND, NULL, PRECISION, RADIX, RANGE, or TINY, or the inquiry functions in the intrinsic module IEEE_ARITHMETIC, a component at any level of component selection may be a pointer that is not associated with a target, or an allocatable component that is not allocated.

In a reference to NULL in which any pointers in the argument are not associated, the result is as if those pointers were associated to an object of the declared type.

2.7.12.2 Linear algebra

References: 04-181, 04-182, 05-269.

2.7.12.2.1 `MaxAbsLoc` and `MinAbsLoc`

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`.

2.7.12.2.2 More than two arguments for DOT_PRODUCT

One occasionally needs to compute SUM(A*B*C), SUM(A*B*C*D).... The DOT_PRODUCT intrinsic function could be generalized to do this. Hopefully, processors would not create an array temp to do so.

2.7.12.2.3 Control conjugation of first argument of DOT_PRODUCT

One occasionally needs to compute a dot product of complex vectors, in which the first is not conjugated. Add a CONJG=*logical-expr* optional argument to DOT_PRODUCT. If it is not present, or is present and the value of the *logical-expr* is true, the first argument is conjugated.

2.7.12.2.4 Exact DOT_PRODUCT and EXACT_SUM

An exact dot product, and exact sum, accumulated in fixed point, allow to solve some problems that cannot be solved if the accumulation is carried out in floating point. An exact dot product is part of the proposed interval arithmetic standard being prepared by IEEE P1788. Although interval arithmetic benefits from an exact dot product, an exact dot product does not require interval arithmetic. Add an EXACT=*logical-expr* optional argument to DOT_PRODUCT and SUM. If it is not present, or is present and the value of the *logical-expr* is false, the result is accumulated using floating-point arithmetic of the same kind as the result value. If it is present and the value of the *logical-expr* is true, the result is accumulated using complete arithmetic, and then rounded to floating point.

Provide EXACT_DOT_PRODUCT and EXACT_SUM intrinsic functions, with results of type(COMPLETE), where COMPLETE is derived type, defined in ISO_FORTRAN_ENV. The results of these functions contain the exact result, not a rounded floating-point result. Define addition, subtraction, negation, and conversion to and from REAL of various kinds, for type(COMPLETE).

2.7.12.2.5 More than two arguments for MATMUL

One occasionally needs to compute the vector-matrix-vector product $x^T Ay$. The MATMUL intrinsic function could be extended to allow more arguments. The requirement that the two-argument form cannot accept two rank-one arguments should be relaxed, to allow any sequence of compatible matrices and vectors to be multiplied. Aho, Hopcroft and Ullman provide an algorithm in **The Design and Analysis of Computer Algorithms** on pages 67-69 to minimize the number of operations in such a product. To solve this problem explicitly for a product of n factors and then compute the product using a sequence of two-argument MATMUL invocations would require an IF ... ELSE IF ... END IF construct with C_{n-1} branches where C_n is the n^{th} Catalan number given by $\frac{(2n)!}{(n+1)!n!}$, which is the number of ways to parenthesize a product of n factors.

2.7.12.2.6 MASK argument for DOT_PRODUCT and MATMUL

One occasionally needs to compute an inner product using some subset of the operands, with the subset specified by a mask. The DOT_PRODUCT intrinsic function could be generalized to include a MASK argument. The [two-argument] MATMUL intrinsic function could be generalized to include a MASK argument, which would specify the columns of the first argument, and rows of the second argument, to be used to compute the product.

2.7.12.2.7 OUTER_PRODUCT

One occasionally needs to compute the outer product of two vectors, i.e.,

```
   forall(i=1:m,j=1:n) c(i,j) = a(i) * b(j).
```

A new OUTER_PRODUCT(A,B) intrinsic function that takes rank-one arguments A and B and returns a rank-two result C(size(A),size(B)) would be useful.

2.7.12.2.8 Cache characteristics inquiry

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 architecture" units such as "bytes." The reported relative speed (or relative access time) should take the cache data transfer width into account. Additional characteristics 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.

2.7.12.3 Compute "companion" functions efficiently

Reference: 04-173.

One occasionally 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.

This is related to scatters and tuples (2.10), which would allow these procedures to be functions.

2.7.12.4 More mathematical intrinsic functions

Reference: 04-184.

The following appear in applications, and have better round-off characteristics for x near zero when implemented directly rather than as written here: $e^x - 1$, $\log(x+1)$, $x - \log(x+1)$, $(x - \sin(x))/x^3$, $(1 - \cos(x))/x^2$, $(\sinh(x) - x)/x^3$, $(\cosh(x) - 1)/x^2$ and $1/\Gamma(x+1) - 1$. The function $x - 1 - \log(x)$ has better round-off characteristics for x near one when implemented directly rather than as written here. These should be provided for both real and complex arguments. The first two are the ones most commonly found in applications.

2.7.12.5 AVAILABLE

Reference: 15-173.

Provide an intrinsic function that has the effect of either ALLOCATED or ASSOCIATED, depending upon whether the argument is allocatable or a pointer.

2.7.12.6 DIM

Allow X and Y arguments to be integer or real, independently. Result characteristics are determined as if for the result of X-Y; compare to DOT_PRODUCT.

2.7.12.7 Wait for asynchronous EXECUTE_COMMAND_LINE

Add an ID argument that can be used in a WAIT statement to inquire whether an asynchrously-executed command has completed execution. The IOSTAT variable in the WAIT statement would be the exit status.

2.7.12.8 Caseless INDEX, SCAN, and VERIFY

One sometimes needs to do the calculations provided by INDEX, SCAN, and VERIFY without regard to case. One must copy one or both strings and convert them to upper case (or lower case). This requires an auxiliary variable and the cost of copying and conversion. It would be helpful if INDEX, SCAN, and VERIFY had an optional logical argument, assumed false if not present, that causes the computations to be done without regard to case if it is present and true.

2.7.12.9 MATMUL

Remove the requirement that MATRIX_A and MATRIX_B cannot both be rank 1. This provides a slightly different dot product, which doesn't conjugate its first argument if it's complex.

Add optional arguments to specify whether the first or second argument is conjugated or transposed. This would give MATMUL the full functionality of *GEMM.

2.7.12.10 TRANSPOSE

Add an optional argument to specify whether the result is conjugated.

2.7.12.11 UPPER_CASE and LOWER_CASE

UPPER_CASE and LOWER_CASE intrinsic functions, with obvious functionality, would be useful.

2.7.12.12 FINDALL

Reference: 15-171.

Provide an intrinsic function that returns a rank-2 array of all the locations within an array having a specified value. The extent of the first dimension of the result is equal to the rank of the array, and the extent of the second dimension is the number of elements of the array that are equal to the specified value.

2.7.12.13 Extensions to FINDLOC

Reference: 15-170.

If FINDLOC had an optional ORDERED argument, which if true specifies that the values of the array elements, taken in array element order, are either nonincreasing or nondecreasing, it would allow a more efficient algorithm to be used. Without knowing that the elements are ordered, one must use an $O(n)$ algorithm. Knowing they are ordered, one can use a secant method, an $O(\log\log n)$ algorithm, for integer or real types, and binary search, an $O(\log n)$ algorithm, for all other ordered types.

FINDLOC should be extended to derived types that have a type-bound == operator taking two operands of the type, allowing the obvious $O(n)$ algorithm, and to ones that in addition have a type-bound <= operator taking two operands of the type, allowing binary search if the ORDERED argument is present and true.

2.7.12.14 Equivalent of MERGE for pointers

Reference: 15-172.

The equivalent of MERGE for pointers would occasionally be useful.

Something like `MERGE_PTR ( TSOURCE, FSOURCE, MASK )` but a transformational instead of elemental function, returning a pointer associated with TSOURCE (if not a pointer) or having the same association status as TSOURCE (if a pointer), or one associated with FSOURCE (if not a pointer) or having the same association status as FSOURCE (if a pointer), depending upon MASK (a scalar). TSOURCE and FSOURCE shall both be data objects or both be procedure objects. If data objects they shall have the same declared type, kind type parameters and rank, each one shall either be a pointer or have the TARGET attribute, and the result is a pointer with the same declared and dynamic type, type parameters and rank as the selected target. If and only if both are simply contiguous, the result is simply contiguous. If and only if both are polymorphic, the result is polymorphic. If procedure objects they shall have the same abstract interface.

2.7.12.15 Extensions to MOVE_ALLOC

Reference: 15-174.

Allow to specify the lower bounds of TO during MOVE_ALLOC, analogously to pointer assignment with lower bounds specifications. This would be useful if all you wish to do is reuse the amount of space.

Allow the FROM argument to MOVE_ALLOC to be a reference to a function that has an allocatable result value.

2.7.12.16 Getting dynamic type name

For purposes of producing messages, a function that returns the name of the dynamic type of an object would be useful. The result ought to be an allocatable default character scalar with deferred length.

2.7.12.17 Lazy argument and operand evaluation

Certain intrinsic functions, especially MERGE (and MERGE_PTR), could be executed more efficiently, and be useful in more circumstances, if evaluation of their actual arguments were lazy instead of eager. That is, if an actual argument were not evaluated until its value is needed. For example, it would be possible to use MERGE(SIZE(A), 0, PRESENT(A)) within a specification expression.

Similarly, it would be useful if the second operands of the new operators .ANDTHEN. and .ORELSE. described in Section 2.11.2 were specified to have lazy evaluation semantics.

2.7.12.18 Help for optimization

Reference: 97-114r2.

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. Allow a reference to the function in specification expressions. This is incompatible with the proposal that construct labels are local to their constructs (see 2.1.10).

2.7.13 Procedures in intrinsic modules

2.7.13.1 Extensions to C_F_POINTER

There is no mechanism for a pointer to get values for deferred length parameters if it gets its target by execution of C_F_POINTER. Add an optional LEN argument to C_F_POINTER. Require that it not be present unless FPTR has a deferred length parameter. Require that it be present if FPTR has a deferred length parameter and is not of type character. If FPTR is of type character with a deferred length parameter, and LEN is not present, the value of the length parameter of FPTR is one.

The lower bounds of FPTR cannot be specified if it is an array. An auxiliary pointer is necessary. Add an optional LBOUND argument, with the same shape as SHAPE, that specifies the lower bounds if present. If absent, the lower bounds are all 1.

2.7.14 KIND arguments whose values are used for generic resolution

Reference: 04-188.

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 shall be constant expressions, again as is required for derived type parameters, it would be possible to use the values of these arguments for generic resolution.

Dummy arguments with the KIND attribute would be required to be of integer type, and to have initialization expressions that specify the value the corresponding actual argument is required to have.

2.7.15 Resolve generic without invoking a procedure or evaluating arguments

Reference: 04-391r1

Rationale

With care and diligence, one can develop a program so that related sets of variables, constants and function results are parameterized by a single kind type parameter. In order to change the kind of that set of entities, one need only change one named constant's definition — almost: Generic procedures cannot be actual arguments or procedure pointer targets. Thus, if one needs to change the program, in addition to changing the single named constant definition, one needs to find all places where a specific procedure that operates on the entities in question is an actual argument or procedure pointer target, and manually edit those appearances.

Alternatively, one needs to write a specific procedure, with arguments appropriately parameterized, which in turn references the generic procedure, and pass that specific procedure as an actual argument. This is less onerous than before Fortran 2008, because of the ability to pass internal procedures as actual arguments. Nonetheless, it increases code bulk, which increases lifetime cost.

It would be helpful to have a facility to resolve a generic name to a specific procedure without evaluating any arguments or invoking a procedure.

Proposal

Allow an actual procedure argument to be a generic identifier provided the referenced procedure has explicit interface, and the dummy argument corresponding to the procedure actual argument has explicit interface. Generic resolution of the actual argument to a specific procedure would be based upon characteristics of its associated dummy argument.

Allow a procedure pointer target to be a generic identifier provided the procedure pointer has explicit interface. Generic resolution of the pointer target to a specific procedure would be based upon characteristics of the procedure pointer.

Alternative proposal

If an actual argument is *generic-name*, and an interface body with that name is accessible at the point the procedure is invoked, the procedure that is invoked shall have explicit interface. The dummy argument associated with a *generic-name* shall be a dummy procedure with explicit interface. The specific procedure from the interface that is compatible with the dummy argument is associated with the dummy argument. If the dummy argument is a subroutine the *generic-name* shall identify an interface that provides a subroutine that is compatible with the dummy argument's interface, and that subroutine is associated with the dummy argument. If the dummy argument is a function the *generic-name* shall identify an interface that provides a function that is compatible with the dummy argument's interface, and that function is associated with the dummy argument.

If a *proc-target* in a procedure pointer assignment statement is *generic-name*, and an interface body with that name is accessible at the point where procedure pointer assignment appears, the *proc-pointer* shall have explicit interface, the *generic-name* shall identify an interface that provides a specific procedure that is compatible with the pointer object's interface, and the *proc-pointer* becomes associated with that specific subroutine or function.

Original proposal from 04-391r1

Given exemplars of actual arguments, resolve a generic name to a specific procedure without invoking the procedure or evaluating its arguments.

There are at least two ways to do this. One is to provide a syntax that is suggestive of procedure reference, but does resolution instead. One possibility for this is to enclose an actual argument list in square brackets or curly brackets instead of round brackets. E.g.,

```
  call solver ( myVec, myJacobian, myModel[myVec,myJacobian] )
```

Another is to provide an entity that looks like an intrinsic function but that has the important distinction that its arguments aren't evaluated. Indeed, this entity that has the appearance of a function reference isn't even invoked during program execution. It is entirely resolved to a procedure by the processor during translation. E.g.,

```
  call solver ( myVec, myJacobian, resolve(myModel,myVec,myJacobian) )
```

RESOLVE would be an inquiry "function" that takes a *generic-spec* as its first "argument," and doesn't evaluate its other arguments — because it's an inquiry function.

It should be possible to resolve a type-bound generic reference, e.g., `resolve(a%b)`.

The macro facility might solve this problem, but it's not obvious that it does so in a readable way.

No matter what syntax is used, it should be allowed to use the result either as an actual argument or a procedure pointer target.

2.7.16 "Walkback" facility

Reference: 06-110

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. This is not inconsistent with a program having more than one image, but it is not, however, thread safe (wouldn't work with the PARALLEL construct in 1.9), and probably doesn't work with coroutines (1.6). 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 preceding paragraphs, and returns the "line number," but having a direct way to do it would be useful.

2.8 ASSIGN statement

Add a statement that contains an assignment statement, in which

- it is not necessary that the variable be allocatable if it is polymorphic,
- whether the dynamic types and length type parameter values are the same, and whether shapes are conformant, is checked,
- optional STAT= and ERRMSG= specifiers are allowed, and
- an error condition occurs if the variable is not allocatable and either the dynamic types are different, the shapes of the variable and expression do not conform, or corresponding length type parameters have different values, or the variable is allocatable and the values of any corresponding nondeferred length type parameters differ.

2.9 SWAP statement

Reference: 97-114r2, 04-190.

One sometimes needs to exchange two variables, or more rarely two pointers. This requires declaring 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 can be made easier to answer if one uses a BLOCK construct to contain the declaration of the temporary variable, and would be even easier if all constructs could have specification parts, as described in section 2.1.1, so that one could more easily 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.

Swap statements, at least those that swap values, should be allowed in FORALL and WHERE.

2.10 Scatters and Tuples

Reference: 04-191, 13-217.

Array constructors, string concatenation, structure constructors, and the MERGE intrinsic are gathers. Allowing them on the left side of an intrinsic assignment, provided each *expr* that is an *ac-value* in an array constructor, each *level-2-expr* or *level-3-expr* that is an operand of a *concat-op* in a *level-3-expr*, each *expr* in a *structure-constructor*, and the TSOURCE and FSOURCE arguments of MERGE, are *variable*s, would be a scatter, which is occasionally just as useful as a gather. 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 a *variable*. 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))
```

A structure constructor in a variable-definition context would allow to scatter a subset of its accessible components, provided each *component-data-source* is a *variable*. For example, assuming the type of `pointStru` is `stru`, one could write

```
stru(x=a,color=c) = pointStru
```

instead of

```
a = pointStru%x
c = pointStru%color
```

Unlike the constructor case, in a scatter one wouldn't need to supply a *component-spec* for every public component that does not have default initialization.

It is possible to combine these scatters. For example, an array constructor or concatenation could be the *expr* in a *component-spec* in a structure scatter, or vice versa, even with different structure scatters for different array elements.

Even though these things have the same syntax when used in value-reference and variable-definition contexts, it would be clearer in the standard to use different syntax terms and different names for them in variable-definition contexts.

It might be possible and reasonable to allow these things in input lists. Until we allow vector subscripted arrays as actual arguments associated with dummy arguments with INTENT(OUT) or INTENT(INOUT), we shouldn't allow these things there either. I don't think it's reasonable to allow any of them in any other variable definition context.

If a distfix IF-THEN-ELSE operator (2.11.1) is provided, it could be allowed in variable definition contexts, provided its second and third operands are *variable*s, e.g.

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

Similarly, a reference to MERGE could be allowed in a variable definition context if it were to be defined to be an updater (1.3), provided actual arguments corresponding to its TSOURCE and FSOURCE arguments are *variable*s.

Sometimes one needs to group several things together, but a derived type is too big a hammer, and sometimes that just won't work.

Other languages provide for tuples, that is, ordered sequences of variables or expressions. Fortran provides these in their full generality only in input/output lists, and in a semantically restricted but syntactically identical form in array constructors. It would be useful if they were more generally available. The first scatter example above is a special case, consistent with array constructor semantics. A more general form would allow arbitrary mixtures of types. Tuples should be allowed in assignment statements, as actual and dummy arguments, and as function results. This would allow "companion" procedures (2.7.12.3) to be functions instead of subroutines. Tuple dummy arguments and function results could be declared something like this:

```
subroutine S ( Z )
  integer :: X(n)
  real :: Y
  tuple :: Z = (x,y)
```

Elements of tuples that are dummy arguments or function results have to be scalar or explicit shape, and not allocatable.

It should be possible to assign between a tuple and a derived-type object, provided the characteristics of the object's components and the tuple's elements agree. It might be useful to allow an actual argument of derived type to be associated with a tuple dummy argument, but vice-versa is probably undesirable. The same effect can be achieved, at least for INTENT(IN), with a structure constructor, provided the "arguments" of the constructor can be a tuple (so they can be provided by a function that returns a tuple result), and maybe for other intents if the structure constructor actual argument is a scatter.

Unlike objects of derived type, tuples of tuples are just tuples, not some hierarchically compound entity, similarly to the way that array constructors made up of arrays construct a single rank-one array.

A tuple can be a scatter. If one appears in a variable-definition context, all of its elements have to be permitted in a variable-definition context (e.g., they could be scatters).

2.11 Operator and expression improvements

2.11.1 Distfix IF-THEN-ELSE operator

Reference: 04-393, 13-234.

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-ELSEIF-ELSE constructs with 2^n branches. Alternatively, one can create n pointers, and either disassociate them, or associate them with the desired-to-be-present actual arguments. This requires the potential actual arguments to have the TARGET attribute and be variables, which compromises optimizers, and prohibits such actual arguments to be expressions.

Another use is to compute whether a specifier in an I/O statement is to be used. If one wants to compute whether n specifiers are present, one needs a complicated nest of if-then-else constructs with 2^n branches. One cannot use null pointers to simulate the absence of specifiers. Also see 2.7.9.4.

Another use is to compute whether a list item in an I/O list is considered to appear. If one wants to compute whether n list items are present, one needs a complicated nest of if-then-else constructs with 2^n branches, perhaps with a different format statement in each one. One cannot use null pointers to simulate the absence of list items. Also see 2.7.9.4.

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 the actual argument is x, else the actual argument is not present.* For a pointer target, `p ? x` would be a handy shorthand for `p ? x : NULL()` or `MERGE_PTR ( x, NULL(), p)` (2.7.12.14).

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. If the third operand does not appear, the declared and dynamic type of the result are those of the second operand, and the result is polymorphic if and only if the second operand is polymorphic. Otherwise, if the second and third operands are type compatible, the declared type of the result is that of the one that is not an extension. If they are not type and kind compatible, the result is unlimited polymorphic. The result is nonpolymorphic if and only if the third operand does not appear and the second is not polymorphic, or if the second and third operands are both nonpolymorphic and are type and kind compatible. If the result is polymorphic, the first operand is required to be scalar.

This is related to lazy evaluation. See 2.7.12.17.

If `p ? x : y` appears in a variable definition context, `x` and `y` shall be allowed in that variable definition context. If the variable definition context is an actual argument `p` shall be a scalar.

See 2.10. Expressions within `x` and `y` are not evaluated until necessary.

If `p` is an array, `x` and `y` shall be conformable to it and the shape of the result is the shape of `p`.

If `p` is a scalar and `p ? x : y` is not an actual argument, the shape of the result is that of `x` or `y` depending upon whether `p` is true or false. If `p` is a scalar and `p ? x : y` is an actual argument, the shape of the result is scalar if both `x` and `y` are scalars, and the shape of the one that is an array otherwise. This definition gives more latitude to optimizers than requiring the second definition in all cases.

In the `p ? x` case, `p` shall be a scalar. The result type, type parameters and rank are those of `x`. In a variable definition context, `x` shall be allowed in that variable definition context.

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

A tiny bit of the functionality almost exists in the MERGE intrinsic function. The reason it isn't 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, the TSOURCE and FSOURCE arguments of MERGE have to be the same shape (not just conformable), a reference to MERGE cannot appear in a variable definition context, and we don't have a two-argument MERGE that causes its result not to exist (for purposes of argument association or pointer targeting) if its first argument is false.

2.11.2 .ANDTHEN. and .ORELSE. operators

References: 04-390 and 04-403.

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 (or to arrange things in the opposite order and not evaluate `A` if `B` 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, and sometimes it's important to specify the order. For example, in

```
    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 (see 2.7.12.17). 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 ) ...
```

2.11.3 Operators that specify directed rounding

Reference: 04-172.

The way to specify directed rounding in expressions in Fortran since 2003 is by using a procedure from the IEEE module. This isn't terse, and is unkind to the code generators for architectures that encode the rounding as part of the instruction instead of in a processor status register. Operators that specify rounding, at least toward $+\infty$ and $-\infty$, e.g. $+>$ and $+<$, ought to be provided.

This interacts with rounding specifications in type declarations (2.2.5).

2.11.4 Mixed-kind character concatenation

Reference: 04-194, 13-210.

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. In general, character concatenation ought to be allowed if one operand is of a kind for which the set of values is a subset of the values of the kind of the other one, with the kind of the result being the kind of the operand having the larger set of values.

2.11.5 Character array concatenation

Problem

It's easy to convert a string into an array. For example, to convert a string into an array of one-character elements use `[ ( string(i:i), i=1,len(string) ) ]`. But the inverse isn't so easy, and one occasionally needs to do it:

```
  do i = 1, size(array)
    string(i:i) = array(i)
  end do
```

Potential solutions:

Prefix unary concantenation

Introduce a prefix unary concatenation operator //, whose operand is required to be an array. It concatenates the array elements, in array element order, producing a string having a length equal to the product of the element length and the number of elements: `string = // array`. To concatenate two of these, one would write `string = (// array1) // (// array2)`.

PRODUCT intrinsic

Extend the PRODUCT intrinsic so that it concatenates character array elements.

New intrinsic

Provide a new intrinsic, say CONCATENATE, that concatenates character array elements. Include a DIM argument and use it as in the PRODUCT intrinsic. Maybe even include a MASK argument when DIM does not appear.

2.11.6 Substrings of character array sections

Substrings of character array sections cause no technical problems that are not immanent in arrays and array sections. Consider

```
  character(len=2) :: X(4) = [ 'ab', 'cd', 'ef', 'gh' ], Z(2)
  character(len=4) :: Y(2)
  equivalence ( X, Y )
  z = x(1:3:2)      ! permitted, giving z == [ 'ab', 'ef' ]
  z = x(1:3:2)(1:2) ! prohibited, but would reference exactly the same
                    ! character storage units
  z = y(:)(1:2)     ! prohibited, but would reference exactly the same
                    ! character storage units
  z = y(1:2)        ! [ 'abcd', 'efgh' ], not a substring
```

The constraint should be "if *substring-range* appears, *parent-string* shall not be a whole array."

2.11.7 Specify roundings of operator and intrinsic function results

Provide operators with specified rounding modes, e.g. +>, −> or >+, >−, at least for rounding toward $+\infty$ and $-\infty$. Provide functions, e.g. ROUND_UP, that specify the rounding mode used to evaluate their arguments (so they're actually pragmas, not functions). It can be specified in due course whether the function or operator wins in, e.g. ROUND_UP (a −< b), or if functions that specify different rounding are composed. Alternatively to those operators and pragmas, and if we get true enumeration types, define an enumeration type in 14.4, or, better, in 13.9, with enumerators for the rounding modes, and add an argument of that type to (almost) all intrinsic functions that have real results. Then REAL(a-b,round=up) would be the same as ROUND_UP(a-b). Thereby, REAL would be a pragma, not a function.

2.11.8 A "strict" mode for floating-point arithmetic

References: 04-218, F03/0084, F03/0121.

Provide a specification that within a range of executable statements, arithmetic operations and intrinsic elementary functions conform to strict rules. The REAL intrinsic function ought to do what the standard says it does, instead of nothing, within a strict range. Changes of rounding mode ought to have effect within a strict range. It would also make sense to specify the treatment of signed zeroes within that range (see 2.11.9), while it might not make sense to do so outwith it.

One syntactic way to do this is with an attribute for subprograms. Another is as a construct. It would be useful to have both; if only one is acceptable, I prefer the construct. An attribute for objects is a bad idea because it raises the question what happens if an operator has two operands with different strictness, and would require specifications how to propagate strictness.

The 1995 Ada standard specifies a strict numeric mode in section G.2. The foundation for it is the *fundamental interval*, which for any number is bounded by the two consecutive floating-point numbers that surround it. Accuracy of arithmetic operations and intrinsic elementary functions is specified in terms of that interval, and the radix and round-off level (epsilon) for floating-point numbers.

The specification amounts to four or five pages in the Ada standard (which was published with smaller pages than the Fortran standard). The same could be done in the Fortran standard.

2.11.9 Distinguish positive from negative real zero

Reference: 04-216 (Walt Brainerd).

The treatment of negative zero required by 4.4.2.3 in N1826 compels behavior that is inconsistent with common expectations. If all but a few intrinsic functions are prohibited from recognizing the difference between positive and negative zero, then the following results are required (all of which, to meet common expectations, should be different):

```
AIMAG((0.0, -0.0)) -> 0.0
CMPLX(-0.0, -0.0) -> (0.0, 0.0)
CONJG((0.0, -0.0)) -> (0.0, -0.0) !
DBLE(-0.0) -> 0.0d0
DPROD(0.0, -0.0) -> 0.0d0
MAX(-1.0, -0.0) -> 0.0
MIN(1.0, -0.0) -> 0.0
REAL(-0.0d0) -> 0.0
ASIN(-0.0) -> 0.0
```

```
ATAN(-0.0) -> 0.0
ATAN2(-0.0,1.0) -> 0.0
SIN(-0.0) -> 0.0
SINH(-0.0) -> 0.0
TAN(-0.0) -> 0.0
TANH(-0.0) -> 0.0
TRANSFER(-0.0, 0.0) -> 0.0
MERGE(-0.0, 1.0, .true.) -> 0.0
```

There are other cases that are also arguably controlled by the existing constraint (in which -0.0 appears as a value inside an array argument, for example). None of the above results meet common expectations.

See *Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit* by William Kahan in **The State of the Art in Numerical Analysis**, Eds. Iserles and Powell, Clarendon Press, Oxford, 1987.

2.11.10 Simplified range tests

A specification of a range of the form $a \odot x \odot b$, where either $\odot$ is independently $<$ or $\leq$, is common in applied mathematics. In Fortran one is required to write for example `P <= X .and. X < Q`. Where `X` is an expression, one who doesn't trust the optimizer not to evaluate it twice will create a temporary variable. Where `P`, `X`, and `Q` are conformant arrays, the temporary variable is an array.

It would be helpful if the shorthand *expr*$_1$ *op-a* *expr*$_2$ *op-b* *expr*$_3$ were available to denote *expr*$_1$ *op-a* *expr*$_2$ `.AND.` *expr*$_2$ *op-b* *expr*$_3$, with *op-a* and *op-b* independently allowed to be either `<` or `<=`, and a specification that *expr*$_2$ is only evaluated once.

2.12 Input/Output improvements

2.12.1 Inter-program data transport via Input/Output statements

Reference: 97-114r2, 08-204.

The problem

Transporting a structure from one program to another via MPI or PVM or some other C-based library is tedious, error prone, and fragile.

Asychronous data transport using procedure libraries, especially MPI, is problematical.

Coarrays cannot help with inter-program data transport; they're only for intra-program data transport.

Proposal

- Add a specifier to OPEN, say MORE= or EXTRA=, with a value that is a default character scalar expression. Its use is processor dependent.
- Add notes in Clause 9 or in Annex C urging processors to
  - interpret the specifier to facilitate inter-program communication via input/output statements. Most systems already provide this, but the program at one end is specialized, e.g. NFS, not something general.
  - interpret the specifier to indicate additional information about structured files, such as group, dataset or attribute names in HDF files.

  - interpret the specifier to indicate encodings such as XDR for unformatted files, or allow the ENCODING= specifier for unformatted files, with processor-dependent meaning (unless XDR is standardized).

A new specifier is not strictly necessary, since the interpretation of the FILE= specifier is processor dependent, but on most systems the file names are quite general so a processor might have trouble noticing something special.

2.12.2 Unformatted internal input/output

Background

Reference: 11-256, 11-259.

Internal input/output is limited to synchronous formatted input/output using character internal files.

Proposal

Extend *internal-file* to variables of any type, continuing to restrict formatted I/O using internal files to character variables. For variables of other types, only unformatted I/O is permitted. In either case, the internal file can be a coarray. The internal file is used in array element order.

In the unformatted internal file case, the dynamic type and kind of every effective list item shall be the same as the dynamic type and kind of the internal file. Care is needed here when describing expansion of list items of derived type: Expansion stops when components of the type of the internal file are reached. Whether data are transferred as if by intrinsic assignment, or using defined assignment, can be decided in due course.

Internal I/O can be asynchronous or synchronous. Existing rules concerning access during asynchronous transfer continue to apply.

Extend the ID= specifier in a control information list to allow variables of type(EVENT). This applies to READ, WRITE, INQUIRE and WAIT statements. ID= variables of type(EVENT) could equally be used for external file I/O.

Use the WAIT statement rather than a new statement for notify/query synchronization.

A WAIT statement with an event that appears in a WRITE statement completes when the data in the I/O list have been transferred to the internal file.

A WAIT statement with an event that appears in a READ statement completes when the variables in the I/O list have been transferred from the internal file.

2.12.3 Allow NAMELIST statements in BLOCK constructs

If there is not a technical reason to prohibit NAMELIST statements in BLOCK constructs, allow them.

2.12.4 Derived-type object working like NAMELIST name

Allow a derived-type variable (expression) in the position of a *namelist-name* in a READ (WRITE) statement, provided it has no ultimate pointer or allocatable components. The component names (but not subcomponent names) are used in the same way as *namelist-group-object*s. Subcomponents require qualification by component names, in the same way that components or subcomponents of *namelist-group-object*s require qualification with the *namelist-group-object* name.

2.12.5 Option to output structure component names in NAMELIST

Reference: 04-203.

One can choose to put or not to put structure component names in namelist input. Providing them 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, whether processors put them in the output is processor dependent (most do not). 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.

2.12.6 Always allow semicolon as separator

10.10.2p3 first item, 10.10.3p8, 10.10.4, and 10.11.3.3p4 in 15-007r2 all specify that a semicolon is a separator in list-directed or namelist input if and only if the decimal mode is "comma". It would be helpful if it were always allowed as a separator. This would require quoting a character list item that includes a semicolon.

2.12.7 A way to know the end of a sequential file

One can detect the end of a sequential file by reading it and either branching to the END= label, or examining the IOSTAT= variable. It would be useful if the INQUIRE statement included an ATEND=*scalar-logical-variable* specifier.

2.12.8 A way to know the end of a direct access file

Reference: 97-114r2.

It is sometimes useful to know the record number of the record with the largest record number in a direct access file. This could be done in at least three 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. Another is an inquiry specification, say LAST_RECORD = *variable*, in the INQUIRE statement, that provides that record number.

2.12.9 Absent optional arguments in optional control list specifiers

See 2.7.9.4.

2.12.10 Relax unnecessary restrictions on input/output

2.12.10.1 Nondefault characters as character string edit descriptors

It would help our colleagues who need to use something other than the Latin alphabet if the *default-char-expr* in a *format* were instead allowed to be default kind, ASCII kind, or ISO 10646 kind. As well, the characters in character string edit descriptors should be allowed to be default kind, ASCII kind, or ISO 10646 kind.

2.12.10.2 Allocatable components in I/O lists

Allocatable list items are allowed by 9.6.3p4 in 15-007r2. Where derived-type list items are treated as lists of components, instead of prohibiting allocatable ultimate components in I/O lists, insist that they are allocated. If a derived-type list item in an unformatted I/O list has an allocatable ultimate component, expand it to a list as is done for formatted I/O.

2.12.10.3 WAIT by ID= or INQUIRE by ID= or PENDING=

It shouldn't be necessary to specify a unit when doing a wait with ID=. It shouldn't be necessary to specify a unit when doing an inquire with ID= or PENDING= specifiers. The value of the expression in an ID= specifier uniquely identifies a particular transaction, so the unit can only be the one in that transaction. INQUIRE with PENDING= alone would inquire about all pending transfers, regardless of unit.

2.12.11 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).

2.12.12 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 changeable mode of formatting, and LC and UC edit descriptors. A CASE changeable mode of formatting would also apply to output of namelist names and component names during namelist output (2.12.5).

2.12.13 More precise description of which records become undefined

Subclause 9.6.4.1, paragraph 8, states "If execution of the program is terminated during execution of a WRITE or PRINT statement, the contents of the file become undefined." Surely, at least for a PRINT statement, it's sufficient that the current record becomes undefined.

2.12.14 ASYNCHRONOUS prefix for I/O statements

The ASYNCHRONOUS= specifier is reqired to appear with a constant expression having the value "yes" or "no". It would be simpler and easier to read if OPEN, READ and WRITE statements had an optional ASYNCHRONOUS prefix that implies ASYNCHRONOUS="yes". If the prefix appears, the specifier shall not appear.

2.12.15 Allow output to nonchild units during DIO

Allow output to nonchild units within defined output subroutines.

2.12.16 Ignore ID=, POS= and REC= in internal I/O statements

Ignore ID=, POS= and REC= specifiers in internal I/O statements, instead of prohibiting them.

2.12.17 Allow INQUIRE to refer to an internal file

Prohibiting an INQUIRE statement to inquire using the unit argument of a defined input/output procedure if it represents an internal file, it is not possible to inquire the state of changeable modes that have been changed by format specifiers.

2.12.18 Namelist group objects and END=

Namelist group objects should not become undefined if an end-of-file condition occurs before the namelist name is recognized in the input file.

2.13 Module improvements

2.13.1 Declare accessibility of generic identifier within interface blocks

Allow *access-spec* on an *interface-stmt* that has a *generic-spec*, within the *specification-part* of a module.

R1503	*interface-stmt*	**is**	INTERFACE
		or	INTERFACE [:: [*access-spec*]] *generic-spec*
		or	ABSTRACT INTERFACE

C1503a (R1503) An *access-spec* shall not appear except within the *specification-part* of a module.

2.13.2 Declare accessibility of specific procedures within interface blocks

Allow *access-spec* on a *procedure-stmt* within an *interface-block* within the *specification-part* of a module.

2.13.3 Explicit qualification of names gotten by USE association

When maintaining a program unit, it is handy to be able easily to find from where a name is gotten. With the exception of implicitly-declared external names, USE names, and names that submodules get from their ancestors by host association, one can always tell by looking at the current scoping unit from whence a name arrived in it. If one uses the ONLY clause in USE statements one can see from whence the listed names arrived.

In this regard, it would be helpful to allow, in a reference, to qualify a name accessed by USE association with the module from whence it came, *viz.* [*module-name*%]*use-name*. Whether this form is allowed for an *only-use-name* can be decided in due course. The processor could be required to check that the use name actually was accessed from the specified module. This could not be confused with a reference to a component or type-bound procedure because an object is not allowed to have the same name as a module in a scope where the module name appears in a USE statement.

It would also be useful to allow a specification on the USE statement that requires explicit qualification of all names accessed from the module, for example, USE, QUALIFIED. This would allow entities of the same name to be accessed from several modules, provided they are all accessed with explicit qualification required.

This proposal conflicts with 2.13.5

2.13.4 Modules need initialization parts

Reference: 04-174.

Provide for an *initialization-part* that consists of an *execution-part* and perhaps some more syntax, somewhere in a module, that is specified to be executed exactly once before any procedure within the module is executed, or before any part (including an initialization part) of a program unit that accesses it by use association is executed.

There are three reasons to do this: convenience, clarity and safety. Convenient because the initialization gets done without user code needing to invoke it, and without the initialization part needing to have an explicit “first time flag” to prevent executing it twice. Clear because it puts initialization in a consistent place, specified by the standard. Safe because it guarantees the initialization gets done without needing to depend on scoping units that access the module to invoke the initialization.

The proposal in 04-174 foundered because of interaction with nonsaved module variables. There are no longer such herrings.

Another objection was the need to examine a first-time flag (or invoke the initialization part, which checks a first-time flag) in every procedure in a module that has an *initialization-part*, or that accesses such a module by use association, on the chance that the module is not connected to the main program by a sequence of use associations. This could be addressed by prohibiting a module with an *initialization-part* to be accessed by use association in an external subprogram or a block data subprogram. With this restriction, a conspiracy of compiler and linker could arrange for initialization parts to be executed before module variables can be accessed. If two varieties of initialization part were provided, one arranged to be executed by a conspiracy of the compiler and linker, and the other tested implicitly by a first-time flag in every module procedure and every procedure that accesses the module by use association, modules with only the latter kind of initialization could be allowed to be accessed by use association from an external subprogram or a block data subprogram.

2.13.5 Local name can be the same as a module name

Reference: 04-206.

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 need to use a module whose name is the same as a name that appears as a local name in the using scope, your recourses are to

- change all occurrences of the local name,
- change the used module's name and every USE statement that references it,
- create a new module of a different name, use the desired one there, and use the desired entities from the new module, or
- use the module in another module where it's not otherwise needed just so you can use things from it in places that have local names that are the same as the desired module name.

The first two are expensive and tedious. The last two are ugly kludges.

The foregoing parts of this proposal conflict with 2.13.3.

An alternative for the standard is to replace *module-name* in *use-stmt* with [*local-module-name* =>] *module-name*, and then allow entities within the inclusive scope where the *use-stmt* appears to have the same name as *module-name*.

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.

2.13.6 USE inside of type definitions

Reference: 04-207.

Sometimes, one needs to reference named constants, types, and procedures gotten by use association, from within a type definition. 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. Also, this would put the module name and the names of all accessed entities in the scope of the type definition, so they wouldn't collide with names outside that scope. There need not be a facility to access entities accessed by use association within a type definition outwith the type definition. See 2.13.5.

2.13.7 Access spec for enumerations

It would be useful to allow an *access-spec* on an ENUM statement, to override default accessibility for all the enumerator names in the enumeration. Independently, it would be useful to allow an *access-spec* on an ENUMERATOR statement, to override default accessibility.

R459	*enum-def-stmt*	**is**	ENUM , *enum-attrib-list*
R459a	*enum-attrib*	**is**	BIND(C)
		or	*access-spec*

C499a (R459) BIND(C) shall appear in the *enum-attrib-list* of an *enum-def-stmt*.

R460	*enumerator-def-stmt*	**is**	ENUMERATOR [[, *access-spec*] ::] *enumerator-list*

2.13.8 The PROTECTED attribute ought to imply PUBLIC

The PROTECTED attribute ought to imply the PUBLIC attribute. There is no reason for a private protected variable or procedure pointer.

2.14 IMPORT with renaming would be useful

For the same reasons that USE with renaming is useful, IMPORT with renaming would be useful, especially within constructs.

2.15 Include should be user-defined

Reference: 97-114r2, 04-147.

The quoted text in an include line is presently specified to be interpreted in a processor-defined way. It should be user defined. Something as simple as moving a program from Windows to Linux shouldn't require to edit all the `include` lines to change \ to /. The method by which the processor allows a user to specify the mapping from the character constant in an `include` line to the text the processor includes should be processor defined.

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 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 systems 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 extent in its final dimension.

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), MAPPED :: Var1(*) ! MAPPED means it's not an assumed-size array,
                               ! even if it's a dummy argument
OPEN ( ACCESS='mapped', FILE='AddressBook', STATUS='old', IOSTAT=ier ) Var1
```

or

```
TYPE, MAPPED :: Type2 ! MAPPED types cannot be extended
  INTEGER :: NumTimes
  REAL :: Lat, Lon, TimeStart, TimeStep
  REAL :: MagField(3,*)
END TYPE Type2
TYPE(Type2) :: Var2 ! Variables of MAPPED type can only be scalars
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 types of mapped variables should be allowed to be SEQUENCE or BIND(C) derived types. A variable of mapped type cannot be an array, because then it wouldn't be the last dimension that has indefinite extent. A mapped type cannot be extended, because then the last component wouldn't be the one with indefinite final extent. A mapped variable or a variable of mapped type cannot be referenced before it is opened, or after it is closed. It cannot be ALLOCATABLE or have any ALLOCATABLE or POINTER ultimate components. It might also be necessary to prohibit it to have the POINTER or TARGET attribute. VOLATILE might be important. Polymorphic mapped dummy arguments, and mapped coarrays, are probably workable and useful.

3.2 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.

3.3 Define interfaces and functionality for special mathematical functions

References: N1688, 11-288.

Clause 14 of part 11 of ISO 31 describes several special functions. Interfaces for and functionality of procedures to evaluate most of them, and several not mentioned there, ought to be provided.

If provided as an optional part, processors would not be required to provide them. If they are provided, this part would specify their interfaces and functionalities. This was discussed and rejected at J3 meeting 195, joint with the 2011 WG5 meeting, in Garching.

3.4 Automatic differentiation

For minimization, zero finding, and nonlinear least-squares methods at least, one needs to compute derivatives. Computing derivatives by numerical methods is unstable: the larger the step size in the independent variable, the more the derivative is an average, while the smaller the step size, the more cancellation in the result. If there are several independent variables, each one needs to be perturbed independently, meaning that the state needs to be evaluated $n + 1$ times to compute derivatives with respect to n independent variables.

Computing derivatives analytically and then transforming the mathematical result to software is tedious and error prone.

There have been several preprocessors that produce a procedure that computes the derivative of some subset of an input procedure's outputs with respect to some subset of its inputs.

It would be useful if the syntax and semantics of this capability were standardized.