ISO/IEC JTC1/SC22/WG5 N2212

The new features of Fortran 2023

John Reid

March 13, 2023

Abstract

The aim of this paper is to summarize the new features of the revision of Fortran 2018 that is planned for publication in 2023. It is known informally as Fortran 2023. We take as our starting point Fortran 2018 (ISO/IEC 2018) and its corrigenda (ISO/IEC (2021) and ISO/IEC (2023)). This paper supersedes ISO/IEC JTC1/SC22/WG5 N2194.

For an informal description of Fortran 2018, see Metcalf, Reid and Cohen (2018).

NB This article is not an official document and has not been approved by WG5 or PL22.3 (formerly J3).

Contents

1 Introduction

Fortran is a computer language for scientific and technical programming that is tailored for efficient run-time execution on a wide variety of processors. It was first standardized in 1966 and the standard has since been revised six times (1978, 1991, 1997, 2004, 2010, 2018). The first five revisions alternated between being minor (1978, 1997, and 2010) and major (1991 and 2004). Concern over implementations not being able to keep up with the revisions of the standard led to avoiding a major revison in 2018 and the plan is for the next revision to be minor.

The content of the revision was mostly chosen by WG5 at its meeting in 2019, see Resolution T1 in ISO/IEC JTC1/SC22/WG5 N2170. About 20 features were proposed by USA and are labelled as US 01, US 02, ... One feature was proposed by UK and is labelled UK 01. Additional small features were added by J3 in the course of working on the revision.

This document is based on the draft of the new standard that is current at the time of writing, J3/23-007. For further detail, the reader should consult this. Its introduction contains a list of the extensions to Fortran 2018. This document provides more detailed descriptions.

We use the convention of indicating the optional arguments of a procedure by enclosing them in square brackets in the argument list. We also use square brackets for other optional syntax and *[ ]*... for an item repeated an arbirary number of times, including zero.

2 Language elements

2.1 US 01 & 02. Allow much longer statement lines and overall statement length

In free source form, the limit on line length is raised to ten thousand characters and this applies to characters of any kind. The limit of 255 continuation lines is removed and the limit on statement length is raised to a million characters.

These relaxations are designed to support programs that are constructed mechanically. It not expected that they will be needed in programs written directly by people.

These are hard limits. Processors are required to issue warnings if they are breached.

2.2 US 14. Automatic allocation of lengths of character variables

When a deferred-length allocatable variable is defined by intrinsic assignment, as in the example

```
character(:), allocatable :: quotation
:
quotation = 'Now is the winter of our discontent.'
```

it is allocated by the processor to the correct length. This behaviour is extended to messages returned through `iomsg` and `errmsg` specifiers, writing to a scalar character variable as an internal file, and intent `out` and `inout` scalar character arguments of intrinsic procedures, such as in `call get_command(command)`.

2.3 US 16. The specifiers `typeof` and `classof`

The specifier `typeof` is now available to declare one or more entities to be nonpolymorphic with the type and type parameters of a previously declared entity, for example:

```
subroutine work(a)
   type(body) :: a(:)
   typeof(a) :: b(size(a))
```

A type parameter is deferred if it is deferred for the previous entity. The previous entity may have intrinsic type. It may be polymorphic, in which case its declared type is used; it must not be unlimited polymorphic or abstract.

The specifier `classof` is available to declare one or more entities to be polymorphic with the declared type and type parameters of a previously declared entity, for example:

```
subroutine work(a)
   class(*) :: a(:)
   classof(x), allocatable :: particle(:)
```

A type parameter is deferred if it is deferred for the previous entity. The previous entity must not be of assumed type or intrinsic type. It may be unlimited polymorphic.

For both `typeof` and `classof`, if the previous entity is an optional dummy argument it must not have a deferred or assumed type parameter.

2.4 US 22. Conditional expressions and arguments

Conditional expressions, expressions whose value is one of several alternatives, are added. A simple example is

```
value = ( a>0.0 ? a : 0.0)
```

which is a short way of writing

```
if (a>0.0) then
   value = a
else
   value = 0.0
end if
```

The general form of a conditional expression is

(*condition* ? *expression [*: *condition* ? *expression]*... : *expression*)

where each *expression* has the same declared type, kind type parameters, and rank. During execution, each *condition* in succession is evaluated until either

- one with the value true is found, in which case no further *condition*s are evaluated and the conditional expression takes the value of the following *expression* or
- all are found to be false, in which case the conditional expression takes the value of the final *expression.*

A conditional expression is a primary, just as is an expression in parentheses, and can be used in the same way as any other primary. In particular, it may be used as an *expression* in a conditional expression, that is, nesting is permitted. It is polymorphic if and only if one or more *expression*s are polymorphic. Its dynamic type is the dynamic type of the chosen expression. Its declared type, kind type parameters, and rank do not depend on which expression is chosen because they are required to be the same for each expression.

A conditional expression may also be used as an actual argument in a procedure reference, but for actual arguments there is a need to allow for variables and absent arguments. An argument may be a conditional argument, whose general form is

(*condition* ? *consequent [*: *condition* ? *consequent]*... : *consequent*) where each *consequent* is an expression, a variable, or `.nil.` to specify absence. On execution, as for a conditional expression, the *condition*s are evaluated in turn. If one that is true is found, its *consequent* is used and no remaining *condition* is evaluated; otherwise, the final *consequent* is used. Nesting of conditional arguments is not allowed, that is a *consequent* is not permitted to be a conditional argument, but a *consequent* may be a conditional expression.

We will refer to a *consequent* that is an expression or a variable as a **consequent argument**. Each consequent argument in a conditional argument must have the same declared type and kind type parameters. Each must have the same rank or each must be of assumed rank. Each must satisfy all the requirements for being an actual argument corresponding to its dummy argument, apart from those that can vary at run time. For example, each must be a variable if the dummy argument has intent `out` or `inout`.

To ensure that the resolution of a generic call does not depend on which consequent argument is chosen, if any consequent argument in a conditional argument is allocatable or a pointer, they must all be.

Here is an example using conditional arguments in a procedure reference:

```
call sub ( ( x>0? x : y>0? y : z ), ( edge>0? edge : mode==3? 1.0 : .nil.) )
```

where the interface of `sub` is

```
subroutine sub(x, bnd)
   real, intent(inout) :: x
   real, intent(in), optional :: bnd
```

2.5 US 23. More use of binary, octal, and hexadecimal constants

The use of binary, octal, and hexadecimal (boz) constants is very limited in Fortran 2018. They are allowed only in data statements and as actual arguments for appropriate intrinsic procedures. A boz constant will be allowed

- in an initialization of a named constant or variable of type integer or real,
- as the right-hand side of an intrinsic assignment where the variable is of type integer or real,
- as an integer value in an enum constructor (Section 9.3), oe
- as a value in an array constructor with a *type-spec* that specifies the type integer or real.

In an intrinsic assignment to an integer or real variable, the boz constant is converted to the type as by the intrinsic `int` or `real`, respectively, with the kind type parameter of the variable specified as its kind argument.

In an array constructor with a *type-spec* that specifies the type integer or real, the boz constant is converted as for intrinsic assignment. In the real case, the boz constant must be a valid representation for the specified kind of real.

3 Intrinsic procedures and intrinsic modules

3.1 US 03. Extracting tokens from a string

Two intrinsic subroutines, both simple (see Section 7.1), have been added to facilitate the extraction of tokens from a string. For example, if the separator characters are `" "` and `";"` the string

```
"one two three;"
```

is taken to contain the tokens `"one"`, `"two"`, `"three"` and `""`. The tokens can be extracted one at a time by the positions of the separators or all at once by an array of tokens or by arrays of starting and ending positions of tokens. No argument of intent `out` or `inout` is permitted to be a coarray or a coindexed object. Note that there are two overloaded forms of `tokenize`.

`call split (string, set, pos` *[*`,back`*]* `)` updates the integer `pos` to the position of the next (or previous) separator in `string`.

`string` is a scalar character object with intent `in`.

`set` is a scalar character object with intent `in`. It has the same kind value as `string` and holds a set of separator characters.

`pos` is a scalar integer variable with intent `inout`. If `back` is absent or is present with the value false, `pos` must have a value in the range `0` ≤ `pos` ≤ `len(string)` on entry and is given the position of the first separator in `string` after position `pos` or `len(string)+1` if there is no such separator. If `back` is present with the value true, `pos` must have a value in the range `1` ≤ `pos` ≤ `len(string)+1` on entry is given the position of the last separator in `string` before position `pos` or `0` if there is no such separator.

`back` is an optional scalar object of type logical and intent `in`.

`call tokenize (string, set, tokens` *[*`,separator`*]* `)` finds all the tokens in a string.

`string` is a scalar character object with intent `in`.

`set` is a scalar character object with intent `in`. It has the same kind value as `string` and holds a set of separator characters.

`tokens` is of type character and of the same kind as `string`. It has intent `out`. It is an allocatable array of rank one and deferred length. It is allocated by `tokenize` to have character length equal to the length of the longest token, lower bound one, and upper bound the number of tokens in `string`. It is assigned with the values of the tokens, in order.

`separator` is an optional argument of type character and of the same kind as `string`. It has intent `out`. It is an allocatable array of rank one and deferred length. It is allocated by `tokenize` to have character length one, lower bound one, and upper bound the number of separators in `string`. It is assigned with the values of the separators, in order.

`call tokenize (string, set, first, last)` finds the starts and ends of all the tokens in a string.

`string` is a scalar character object with intent `in`.

`set` is a scalar character object with intent `in`. It has the same kind value as `string` and holds a set of separator characters.

`first` is an allocatable integer array of rank one and intent `out`. It is allocated by `tokenize` to have lower bound one and upper bound the number of tokens in `string`. It is assigned with the starting positions of the tokens, in order. A token of zero length is taken as starting immediately after the token that precedes it or at position 1 if at the start of `string`.

`last` is an allocatable integer array of rank one and intent `out`. It is allocated by `tokenize` to have lower bound one and upper bound the number of tokens in `string`. It is assigned with the finishing positions of the tokens, in order. A token of zero length is taken as finishing immediately before its starting position.

3.2 US 04. Trig functions that work in degrees

The following are elemental functions that evaluate elementary mathematical functions for real arguments, working in degrees instead of radians. Each result is real with the kind type parameter of the first argument. They have been added because they have been widely implemented as extensions of the standard and are widely used.

`acosd (x)` returns the arc cosine (inverse cosine) function value for a real value x such that $|x| \le 1$, expressed in degrees in the range $0 \le$ `acosd(x)` ≤ 180.

`asind (x)` returns the arc sine (inverse sine) function value for a real value x such that $|x| \le 1$, expressed in degrees such that $-90 \le$ `asind(x)` ≤ 90.

`atand (x)` returns the arc tangent (inverse tangent) function value for a real value x, expressed in degrees in the range $-90 \le$ `atand(x)` ≤ 90.

`atand (y, x)` is the same as `atan2d (y, x)`, see below.

`atan2d (y, x)` returns the arc tangent (inverse tangent) function value in degrees for a pair of real values, x and y, with the same kind type parameter. They must not both be zero. The result is expressed in degrees in the range $-180 \le$ `atan2d(y,x)` ≤ 180. It has a value equal to a processor-dependent approximation to `atan2(y, x)`$\times 180/\pi$.

`cosd (x)` returns the cosine function value for real values x in degrees.

`sind (x)` returns the sine function value for real values x in degrees.

`tand (x)` returns the tangent function value for real values x in degrees.

3.3 US 05. Trig functions that work with half revolutions

The following are elemental functions that evaluate elementary mathematical functions for real arguments, working in half revolutions (180 degrees). Each result is real with the kind type parameter of the first argument. They have been added because they are mentioned in the IEEE standard for floating-point arithmetic and have been implemented as extensions of the Fortran standard.

`acospi (x)` returns the arc cosine (inverse cosine) function value for a real value x such that $|x| \leq 1$, expressed in half revolutions in the range $0 \leq$ `acospi(x)` ≤ 1.

`asinpi (x)` returns the arc sine (inverse sine) function value for a real value x such that $|x| \leq 1$, expressed in half revolutions in the range $-0.5 \leq$ `asinpi(x)` ≤ 0.5.

`atanpi (x)` returns the arc tangent (inverse tangent) function value for a real value x, expressed in half revolutions in the range $-0.5 \leq$ `atanpi(x)` ≤ 0.5.

`atanpi (y, x)` is the same as `atan2pi (y, x)`, see below.

`atan2pi (y, x)` returns the arc tangent (inverse tangent) function value in half revolutions for a pair of real values, x and y, with the same kind type parameter. They must not both be zero. The result is expressed in half revolutions in the range $-1 \leq$ `atan2pi(y,x)` ≤ 1. It has a value equal to a processor-dependent approximation to `atan2(y, x)`$/\pi$.

`cospi (x)` returns the cosine function value for real values x in half revolutions.

`sinpi (x)` returns the sine function value for real values x in half revolutions.

`tanpi (x)` returns the tangent function value for real values x in half revolutions.

3.4 US 06. `selected_logical_kind`

The function `selected_logical_kind` is added to match the existing functions for choosing a suitable kind for a type.

`selected_logical_kind (bits)` is a transformational function that returns as a default integer scalar the value of a kind type parameter of a logical type whose storage size in bits is at least `bits`, or -1 if no such type is available. If there is more than one such kind, the one with smallest kind value from those with smallest storage size is chosen.

3.5 Changes to `system_clock`

In Fortran 2018, users are free to use integers of any kind as actual arguments for `system_clock`. It was intended that this would allow the use of long integers for accurate timing on modern hardware with fast clocks. However, this left vendors unclear about how to accommodate default integers or even short (16-bit) integers — some provided an imprecise clock or caused an error return. There was also uncertainty over the effect of disagreement in kinds within a single call.

These problems are addressed by requiring that all integer arguments in a single call have the same kind. Furthermore, their decimal exponent range is required to be at least that of default integer, and implementations are recommended to support long integers (decimal exponent range at least 18). The processor is permitted to provide any number of clocks, including zero. Which clock is referenced depends on the kind of the integer arguments. Whether an image has no clock, has one or more clocks of its own, or shares a clock with another image, is processor dependent. It is recommended that a call with a real argument `count_rate` include one or more integer arguments in order to specify the clock.

3.6 Changes for conformance with new IEEE standard

Changes are made to the intrinsic module `ieee_arithmetic` for conformance with the new IEEE standard, ISO/IEC 60559:2020.

Four elemental functions have been added for the IEEE operations of maximum, maximumMagnitude, minimum, and minimumMagnitude:

```
    ieee_max (x, y)
    ieee_max_mag (x, y)
    ieee_min(x, y)
    ieee_min_mag (x, y)
```

where `x` and `y` are real with the same kind type parameter, return either `x` or `y`. The result is `x` if `x>y`, `abs(x)>abs(y)`, `x<y`, or `abs(x)<abs(y)`, respectively. It is `y` if `x<y`, `abs(x)<abs(y)`, `x>y`, or `abs(x)>abs(y)`, respectively. If either `x` or `y` is a NaN, the result is a quiet NaN. If `x=y`, and the signs are the same, the result is the value of either `x` or `y`. If one argument is negative zero and the other is positive zero, the result is positive zero. If both of `x` and `y` are signaling NaNs, `ieee_invalid` signals; otherwise, no exception is signaled.

The four elemental functions `ieee_max_num`, `ieee_max_num_mag`, `ieee_min_num`, and `ieee_min_num_mag` now conform to the operations maximumNumber, maximumMagnitudeNumber, minimumNumber and minimumMagnitudeNumber in ISO/IEC 60559:2020; the changes affect the treatment of zeros and NaNs, which are as for `ieee_max`, `ieee_max_mag`, `ieee_min`, and `ieee_min_mag` (see previous paragraph).

3.7 US 07 & 08. Additional named constants to specify kinds

These additional named constants are available in the module `iso_fortran_env` for kind type parameter values of types of given storage sizes in bits:

`logical8`	8-bit logical
`logical16`	16-bit logical
`logical32`	32-bit logical
`logical64`	64-bit logical
`real16`	16-bit real

They are default integer scalars. If the compiler supports more than one kind with a particular size, which one is chosen is processor dependent. If the compiler does not support a kind with a particular size, that constant has a value of -2 if it supports a kind with a larger size, and -1 if it does not support any larger size.

4 Interoperability with C

4.1 UK 01. Extend the intrinsic procedure `c_f_pointer` to allow its pointer result to have specified lower bounds

An extra optional argument `lower` has been added at the end of the argument list of the intrinsic subroutine `c_f_pointer`. It has intent `in`, is an array of rank one with the same size as the argument `shape`, and can be present only if `shape` is present. If present, it specifies the lower bounds of the pointer result, which otherwise all have the value 1.

This brings `c_f_pointer` into line with pointer assignment, which allows the lower bounds of an array pointer to be specified.

4.2 US 09. Procedures for converting between Fortran and C strings

On processors that support C character kind (`c_char` $\neq -1$), these procedures are added to the intrinsic module `iso_c_binding` for passing strings between C functions and Fortran procedures. Note that there are two overloaded forms of `c_f_strpointer`.

`f_c_string (string` *[*`, asis`*]* `)` where `string` is a character scalar of kind `c_char` and `asis` is an optional logical scalar, is a transformational function. It returns a character scalar of the same type and kind as `string` whose value is `string//c_null_char` if `asis` is present with the value true and is `trim(string)//c_null_char` otherwise.

`call c_f_strpointer (cstrarray, fstrptr`*[*`, nchars`*]* `)` is an impure subroutine with arguments:

`cstrarray` is a rank-one character array of kind `c_char` and character length one. It is an intent `in` argument. Its actual argument must be simply contiguous and have the `target` attribute.

`fstrptr` is a scalar deferred-length character pointer of kind `c_char`. It is an intent `out` argument whose character length `len` is the largest value for which `cstrarray(1:len)` contains no C null characters, `len` $\leq$ `size(cstrarray)`, and `len` $\leq$ `nchars` if `nchars` is present. `fstrptr` becomes pointer associated with the first `len` characters of the element sequence of `cstrarray`.

`nchars` is an optional integer scalar with intent `in`. If `cstrarray` is assumed-size, `nchars` must be present. If `nchars` is present, its value must be nonnegative and not greater than the size of `cstrarray`.

`call c_f_strpointer (cstrptr, fstrptr, nchars)` is an impure subroutine with arguments:

`cstrptr` is a scalar of type `c_ptr`. It is an intent `in` argument holding the C address of a contiguous array `s` of `nchars` characters. Its value must not be the C address of a Fortran variable without the `target` attribute.

`fstrptr` is a deferred-length character pointer of kind `c_char`. It is an intent `out` argument whose character length `len` is the largest value for which `s` contains no C null characters. It become pointer associated with the leftmost characters (in array element order) of the array `s`.

`nchars` is an integer scalar with intent `in`. Its value must be nonnegative.

5 Input-output

5.1 US 10. The `at` edit descriptor

The `at` edit descriptor has been added to give the effect of `trim` on the corresponding character list item during data output with the `a` edit descriptor. It is not available for input.

5.2 US 11. Control over leading zeros in output of real values

The leading zero mode controls optional leading zero characters in output with `f`, `e`, `d`, and `g` edit descriptors. When the mode is `print`, the processor produces a zero in any position that normally contains an optional leading zero. When the mode is `suppress`, the processor does not produce such a zero. When the mode is `processor_defined`, the processor has the choice.

The default mode for a file can be set on its `open` statement with a `leading_zero=` specifier with one of the values `print`, `suppress`, and `processor_defined`. If not specified, it is `processor_defined`. An `inquire` statement can inquire about the mode for a connection with a `leading_zero=` specifier which is assigned one of the values `print`, `suppress`, or `processor_defined` if the connection is for formatted i/o and `undefined` otherwise.

The `lzp`, `lzs`, and `lz` edit descriptors change the mode temporarily during the execution of an output statement to `print`, `suppress`, and `processor_defined`, respectively. They have no effect during the execution of an input statement.

5.3 Namelist

A public namelist group is allowed to have a private namelist group object.

6 Coarrays

6.1 US 12. Allow an object of a type with a coarray ultimate component to be an array or allocatable

An object of a type that has a coarray ultimate component is allowed to be an array and is allowed to be allocatable, but it remains not allowed to be a coarray (which would be confusing). This can happen at any depth of component selection so a type is allowed to have a potential subobject component that is a coarray.

An `allocate` statement for an object with a coarray potential subobject component is not an image control statement, that is, it involves only the executing image, because the coarray is unallocated so that no image is able to reference it. Fortran 2018 has the constraint that an `allocate` statement must not have a `source=` specifier that is an object with a coarray ultimate component because the need to copy the coarray if it is allocated would make the statement an image control statement. This is retained but has to be generalized to an object with a coarray potential subobject component. However, the corresponding constraint for a *type-spec* or `mold=` specifier is not needed and is removed, except for the case where the object is unlimited polymorphic (see the paragraph after next).

In an intrinsic assignment for an an object with a coarray potential subobject component, the component is subject to the rules of intrinsic assignment for a coarray so the statement is not an image control statement.

On the other hand, a `deallocate` statement for an object of a type that has a coarray potential subobject component involves the deallocation of all its coarray subobjects. It therefore needs to be an image control statement and involve synchronization of all the images in the current team. To make sure that an unlimited polymorphic object does not get a dynamic type with a coarray potential subobject component and cause its `deallocate` statement to become an image control statement, an `allocate` statement that allocates an unlimited polymorphic object is not permitted to have an object with a coarray potential subobject component in a *type-spec*, `mold=` specifier, or `source=` specifier.

An `allocate` statement for a coarray that is a potential subobject component of an object remains an image control statement and involves image synchronization. For an `allocate` or `deallocate` statement of such a coarray, there need to be the following additional conditions on the object itself to ensure that the coarray is properly aligned on the images. On each active image of the current team:

- if it is polymorphic, its dynamic type is the same,
- if it is an element of an array that is not a dummy argument, its position in array element

order in the array is the same,

- if it is an element of an array that is a dummy argument, its position in array element order in the ultimate argument is the same,
- if it is an unsaved local variable of a recursive procedure or an element of such a variable, the depth of recursion of that procedure is the same[1], and
- if it is a dummy argument whose ultimate argument is an unsaved local variable of a recursive procedure or an element of such a dummy argument, the depth of recursion of that procedure is the same.

Copy-in copy-out needs to be avoided for a dummy argument that is a non-allocatable array of a type with a coarray potential subobject component. Here, the actual argument is required to be simply contiguous or an element of a simply contiguous array.

6.2 US 13. Put with notify

Put with notify is popular in the SHMEM community as a very efficient synchronization technique for data transfers between images. The basic idea is to combine a 'put' (definition of a variable on a different image) with a notification mechanism that allows the receiving image to know that the data has arrived. It is especially efficient if network hardware can ensure that the data arrive before the notification occurs.

The derived type `notify_type` is defined by the intrinsic module `iso_fortran_env`. Its components are private and it is extensible. A scalar coarray of the type is a **notify variable**.

Here is a simple example of the new feature:

```
use iso_fortran_env
type(notify_type) nx[*]
:
me = this_image()
if (me == 1) then
   x[10, notify=nx] = y
else if (me == 10) then
   notify wait (nx, until_count=1)
   z = x
end if
```

Here `nx` is a notify variable. On every image it contains a count that has the initial value 0. After an image has assigned its value of `y` to `x` on image 10, the count value of `nx[10]` incremented to 1. Image 10 waits until the count of its `nx` is at least 1, then decrements the count by 1 and continues execution.

[1]This also clarifies the behaviour for the existing Fortran 2018 functionality.

The value of a notify variable includes its notify count, which always has the initial value zero and can be altered only by

- execution of an intrinsic assignment statement whose left-hand-side variable has an image selector that selects the image of the notify variable and has a `notify=` specifier that specifies the notify variable, or
- execution of a `notify wait` statement on the image of the notify variable.

The intrinsic assignment statement increases the count by one. The `notify wait` statement has a threshold value that is its `until_count` specifier if it appears or one otherwise and waits until the count is at least the threshold value and then decreases the count by the threshold value. Here is an example

```
use iso_fortran_env
type(notify_type) nx[*]
:
me = this_image()
if (me <= 4) then
   x(me)[10, notify=nx] = y
else if (me == 10) then
   notify wait (nx, until_count=4)
   z(1:4) = x(1:4)
end if
```

The effect of each update is as if the intrinsic subroutine `atomic_add` were executed for a variable that stores the notify count and has type integer with kind `atomic_int_kind`.

To ensure that the counts of notify variables are initialized to zero and not altered except as explained in the previous paragraph, the only additional ways a notify variable is permitted to appear in a variable definition context are in an `allocate` statement without a `source=` specifier, in a `deallocate` statement, or as an actual argument corresponding to a dummy argument with intent `inout` in a reference to a procedure with an explicit interface. Furthermore, a variable with a nonpointer subobject of type `notify_type` is permitted to appear in a variable definition context only in an `allocate` statement without a `source=` specifier, in a `deallocate` statement, or as an actual argument corresponding to a dummy argument with intent `inout` in a reference to a procedure with an explicit interface.

A named entity with declared type `notify_type`, or which has a noncoarray potential subobject component with declared type `notify_type`, must be a coarray. A component that is of such a type must be a data component.

The `notify wait` statement is not an image control statement. It takes the general form

`notify wait (` *notify-variable[* `,` *event-wait-spec-list]* `)`

where an *event-wait-spec* is an `until_count=` specifier, `stat=` specifier, or `errmsg=` specifier.

6.3 Error conditions in collectives

Fortran 2018 defines a collective subroutine as an "intrinsic subroutine that performs a calculation on a team of images without requiring synchronization". For performance reasons, it was intended that once an image has completed its calculation it should be free to execute other statements without waiting for the other images to do so. Fortran 2018 does not recognize that it might be the case that a collective is successful on one image while having an error condition on others. The revision acknowledges this and allows for different images to have different error conditions.

7 Procedures

7.1 US 15. Simple procedures

A pure procedure changes variables outside its scope only through its arguments. This allows it to be used in parallel constructs, where concurrency issues would otherwise prevent use. A simple procedure is a pure procedure that in addition is restricted to reference variables outside its scope only through its arguments. It represents an entirely local calculation. It may be executed independently by a thread that accesses program data only through the arguments. If all the intent `in` arguments are constants and there are no intent `inout` arguments, it may be performed by the compiler at compile time.

All the intrinsic functions are simple. All the module functions in all of the intrinsic modules are simple. The intrinsic subroutines `mvbits`, `split` (Section 3.1), and `tokenize` (Section 3.1) are simple. The intrinsic subroutine `move_alloc` is simple when the argument `from` is not a coarray. The module subroutines `c_f_procpointer`, `ieee_get_flag`, `ieee_get_halting_mode`, `ieee_get_modes`, `ieee_get_rounding_mode`, `ieee_get_status`, `ieee_get_underflow_mode`, `ieee_set_flag`, `ieee_set_modes`, `ieee_set_rounding_mode`, `ieee_set_status`, `ieee_set_underflow_mode`, and `ieee_set_halting_mode` are simple. The elemental operators `==` and `/=` for two values of one of the types `ieee_class_type` and `ieee_round_type` are simple.

A procedure may be specified as simple with the keyword `simple` in the prefix of its `function` or `subroutine` statement, for example

```
real simple elemental function convert(a)
   type(mine) :: a
   :
```

Of course, if `simple` is specified, neither `pure` nor `impure` may be specified. A simple procedure automatically has the property of being pure. A simple procedure may also be elemental. A dummy procedure or a procedure pointer may be specified to be simple. A type-bound procedure is simple if it is bound to a simple procedure. A deferred type-bound procedure is simple if its interface specifies it to be simple; any overriding type-bound procedure must then also be simple.

A simple procedure must satisfy all the requirements of a pure procedure. In addition,

- it must not reference a variable by use or host association,
- it must not reference a variable in a common block,
- all its dummy procedures must be simple,
- all its internal procedures must be simple,
- all procedures it references must be simple,
- when used in a context that requires it to be simple, its interface must be explicit and specify that it is simple, and
- it must not contain a `entry` statement.

8 Array features

8.1 US 17. Using integer arrays to specify subscripts and section subscripts

It is possible to use an integer array, known as a **multiple subscript**, to specify a sequence of subscripts, for example

```
A(@[3,5]) ! Array element, equivalent to A(3, 5)
A(6, @[3,5], 1) ! Array element, equivalent to A(6, 3, 5, 1)
A(@V1, :, @V2) ! Rank-one array section, the rank of A being
               ! SIZE (V1) + 1 + SIZE (V2).
```

A multiple subscript consists of a commercial at symbol followed by an integer expression that is an array of rank one and the array elements are interpreted as a sequence of subscripts.

A multiple subscript triplet uses rank-one integer arrays to specify a sequence of section subscripts. It has the form

`@` *[int-expr]* : *[int-expr] [* : *int-expr]*

where each *int-expr* is a rank-one integer array or an integer scalar. At least one must be an array and if there are more than one they must have the same size.

If the integer expressions are the arrays `l`, `u`, and `s` of size k, the effect is that of k section subscripts with the elements of `l` as lower bounds, elements of `u` as upper bounds, and elements of `s` as strides, for example

```
A(@[3,5]:[9,10]:[2,3]) ! Equivalent to A(3:9:2, 5:10:3)
```

If any *int-expr* is a scalar, its value is used for each bound or stride, for example

```
A(@[3,5]:[9,10]:2) ! Equivalent to A(3:9:2, 5:10:2)
```

Where an *int-expr* is omitted, the bound or stride is omitted from the equivalent section subscript, for example

```
A(@[3,5]:[9,10]) ! Equivalent to A(3:9, 5:10)
```

8.2 US 18. Using integer arrays to specify the rank and bounds of an array

Integer expressions that are rank-one arrays may be used to specify the rank and bounds of an array.

For an assumed-shape array, the rank and lower bounds may be determined by a rank-one integer array, for example

```
integer, dimension(3) :: lb_array = 0
real :: zz(lb_array+2:)
real, dimension(lb_array:) :: x, y
```

The size of the integer array must be constant and this determines the rank of each array being declared.

For an explicit-shape array, one or both sets of bounds may be determined by rank-one integer arrays of a constant size, and the size determines the rank. If both sets of bounds are specified in this way, the sizes must be the same. The lower bounds may be specified all to have the value of an integer scalar or by omission all to be one. The upper bounds may be specified all to have the value of an integer scalar. Here are some examples

```
real, dimension(lb_array:ub_array) :: z
real :: zz(lb_array+2:n), x(ub_array), y(0:ub_array)
real :: u(shape(w)) ! Array with the same shape as w
```

For allocating an allocatable array, one or both sets of bounds may be given by rank-one integer array expressions of size equal to the rank of the array. Also, one may be given as an array expression and the other as a scalar expression, in which case the scalar value is broadcast to all dimensions. Here are some examples:

```
real, allocatable, dimension(:,:,:) :: x, y, z
integer :: lower(3), upper(3)
:  ! Code that gives values to lower and upper
allocate(x(:upper), y(lower:upper), z(0:upper))
```

Note that this feature is not provided for cobounds.

Similarly, in a pointer assignment the set of lower bounds or both sets of bounds in a remapping may be given by integer array expressions. The sizes of the array expressions must equal the rank of the pointer array and if one is an array the other may be a scalar whose value is broadcast. Here are some examples:

```
real, pointer, dimension(:,:,:) :: x, y, z
integer :: lower(3), upper(3)
:  ! Code that allocates x and gives values to lower and upper
y(lower:) => x
z(0:upper) => x
```

8.3 Using an integer constant to specify rank

The rank of an assumed-shape or deferred-shape entity may be specified by an integer constant in a `rank` clause in a type declaration statement. For example,

```
subroutine ex(a)
real, rank(2) :: a  ! Equivalent to real :: a(:,:)
```

declares `a` to be an assumed-shape array of rank 2. The value of the integer constant is required to be non-negative and not greater that the maximum rank supported by the processor. The value zero specifies a scalar. With the `allocatable` or `pointer` attribute, it specifies a deferred-shape array; otherwise, it specifies an assumed-shape array with all lower bounds equal to one. Here are some examples

```
integer :: x0(10,10,10)
logical, rank(rank(x0)), allocatable :: x1 ! rank 3, deferred shape
complex, rank(2), pointer :: x2 ! rank 2, deferred-shape
logical, rank(rank(x0)) :: x3 ! rank 3, assumed-shape
real, rank(0) :: x4 ! scalar
```

8.4 US 20. Reduction specifier for `do concurrent`

A named variable may be declared in a `do concurrent` construct to have `reduce` locality within it, for example

```
real :: a, b, x(n)
:
a = 0.
b = -huge(b)
do concurrent (i = 1, n) reduce(+:a) reduce(max:b)
  a = a + x(i)**2
  b = max(b,x(i))
end do
```

allows the computations $\sum_{i=1}^{n} x_i^2$ and $\max_{i=1}^{n} x_i$ to be parallelized. We will refer to a variable that computes such a result as a **reduction variable**. It must be a named variable. Its name and the operator or function name must be specified in a `reduce` clause on the `do concurrent` statement.

A named variable with `reduce` locality must be of an intrinsic type that is suitable for its operator or function, and permitted to appear in a variable definition context. It must not be a coarray or an assumed-size array. It must not be asychronous or volatile. It must not be an optional argument.

A reduction variable is allowed to appear within its `do parallel` construct only in an intrinsic assignment of one of the forms *var* = *var op expr* and *var* = *expr op var* where *op* is one of the intrinsic operators `+`, `*`, `.and.`, `.or.`, `.eqv.`, or `.neqv.`, or as the left-hand side of an intrinsic assignment whose right-hand side is one of the intrinsic functions `max`, `min`, `iand`, `ieor`, or `ior` with the reduction variable as an argument. All occurences in the construct must have the same form.

The effect is as if each iteration has a separate reduction variable with exactly the same properties as its original, the 'outside' variable. If the outside variable is allocatable it must be allocated. If it is a pointer, it must have a target. The inside variable does not have the `allocatable`, `bind`, `intent`, `pointer`, `protected`, `save`, `target`, or `value` attribute, even if the outside variable does. Each inside variable is initialized at the start of execution of its iteration to

- `0` for `+`, `ieor`, or `ior`;
- `1` for `*`;
- all `1`s for `iand`;
- `.true.` for `.and.` or `.eqv.`;
- `.false.` for `.or.` or `.neqv.`;
- the least representable value of the type and kind for `max`; and
- the largest representable value of the type and kind for `min`.

On termination of the `do concurrent` construct the outside variable, or its target if it is a pointer, is updated by using the reduction operation to combine it with the values of all the inner variables. The updates may be performed in any order.

A `reduce` clause may be given a list of variables for a single reduction and there may be any number of `reduce` clauses, for example

```
do concurrent (i = 1, n) reduce(+:a, b, c) reduce(max:d, e, f)
```

9 US 21. Enumerations

9.1 Introduction

Fortran 2023 supports two variants of enumeration types. One variant extends the pre-existing C-interoperability feature of `enum` and the other one provides Fortran-specific enumeration types with additional semantics that are incompatible with those of C-style enums. We start with the description of the Fortran-specific variant.

9.2 Enumeration types

Enumeration types are added. Each is defined by the user to contain a set of named constants that are without any associated data. A scalar variable of the type has one of the constants as its value, which allows it to control an action, for example

```
enumeration type :: colour
   enumerator :: red, orange, green
end enumeration type
type(colour) light
:
if (light==red) ...
```

There may be more than one enumerator statement in an enumeration type declaration and the separators `::` are optional. The names of the types and their constants have exactly the same scoping rules as for other types and constants; for example, a procedure that contains the above declaration must not contain a `real` variable named `red`. An enumeration type is not a derived type but behaves like a derived type with no components, except that variables are not permitted to be polymorphic and the only values of scalar constants are the enumerators of the type definition. Apart from no polymorphism, variables have the usual properties and attributes. They may be arrays, coarrays, pointers, allocatable, etc. A component of a derived type may be declared to be of enumeration type. Enumeration types are not interoperable because there are no corresponding C types.

The intrinsic function `int` is extended to allow its main argument `a` to be of enumeration type and it returns the position of `a` in its type declaration as an integer.

In an intrinsic assignment statement with a variable (left-hand side) of enumeration type, the expression (right-hand side) must be of the same enumeration type. The intrinsic operations `==`, `/=`, `<`, `<=`, `>`, and `>=` are available in an expression when both operands are objects of the same enumeration type and the result is of type default logical. They work with the positions of the operands in the type declaration. No other intrinsic operations are available for one or both operands of enumeration type.

An enumerator may be accessed as an 'enumeration constructor' through its position in the type declaration. For example `colour(2)` has the value `orange`. This allows the enumerators to be accessed in a `do` loop such as:

```
do i = 1,3
   light = colour(i)
   :
end do
```

The general form of an enumeration constructor is *type*`(`*scalar-int-expr*`)` for any scalar integer expression *scalar-int-expr*. An enumeration constructor whose *scalar-int-expr* is constant may appear in a `data` statement.

An array constructor with the usual syntax is available to construct an array of an enumeration type, for example, `[red, (colour(i),i=2,3)]`. All the array elements specified must be enumerators of the same type.

Objects of an enumeration type may be used in a `select case` construct, for example

```
select case (light)
   case (red)
    :
   case (orange:green)
    :
end select
```

When a colon is present in a `case` statement the enumerators are selected by their positions in the type declaration.

For other kinds of sequential access to the enumerators of an enumeration type, the first enumerator of type `t` is available as `t(1)`, the intrinsic function `huge` is extended so that the last enumerator of the type of enumerator `a` is available as `huge(a)`, and there are two elemental intrinsic functions:

`next(a` *[*`,stat`*]*`)` returns the next enumerator after `a` in the sequence of the type of `a` or `a` itself if it is last in this sequence.

- `a` is of an enumeration type and has intent `in`.
- `stat` (optional) is an integer scalar with a decimal exponent range of at least four. If present, it has intent `out` and is given the value zero if `a` is not last or a processor-dependent positive value if it is last.

`previous(a` *[*`,stat`*]*`)` returns the previous enumerator to `a` in the sequence of the type of `a` or `a` itself if it is first in this sequence.

- `a` is of an enumeration type and has intent `in`.
- `stat` (optional) is an integer scalar with a decimal exponent range of at least four. If present, it has intent `out` and is given the value zero if `a` is not first or a processor-dependent positive value if it is first.

For either function, if `stat` is not present but would have been assigned a nonzero value if present, error termination is initiated.

An enumeration type that is declared in a module may be given the `private` or `public` attribute when it is declared, for example,

```
enumeration type, private :: colour
   enumerator :: red, orange, green
end type
```

The attribute may be given in a `private` or `public` statement, which overrides any declaration in the type definition. If declared in neither of these ways, the accessibility of the type is the default for entities of the module.

An object of enumeration type is not permitted in list-directed or namelist I/O. In formatted I/O, it corresponds to an `i`, `b`, `o`, or `z` edit descriptor. For output of a value of enumeration type, the position in its type definition is transferred. For input, the value must be positive and less than or equal to the number of enumerators in its type definition and the enumerator with this position is transferred.

9.3 Enum types

In Fortran 2018, an enumeration is an ordered collection of enumerators, which are named integer constants of a single kind. The integer type with this kind is interoperable with the enumeration type that the C companion processor chooses for the same set of enumerators in the same order. This has the disadvantage that the Fortran integer type has many other uses. The revision of the standard allows the definition of an enumeration to create a new type, known as an 'enum type', that also interoperates with the C enumeration type. The name of the type is declared on the `enum` statement. For example,

```
enum, bind(c) :: season
   enumerator :: spring=5, summer=7, autumn, winter
end enum
type(season) my_season
```

defines the enum type `season` and the variable `my_season` of this type. A component of a derived type may be declared to be of an enum type. The enumerators are just as in Fortran 2018. The name of the type and the names of its enumerators have exactly the same scoping rules as have other types and constants; for example, a procedure that contains the above declaration must not contain a `real` variable called `spring`.

Note that a scalar object of an enum type is not an integer but interoperates with a C integer that may have a value or be undefined. It is convenient to refer to this as the 'integer value of the object'. For an enumerator, the integer value was provided in the enumeration definition. The intrinsic function `int` is extended to allow its main argument `a` to be of enum type and return its integer value. An 'enum constructor' of the form *type-name*`(`*expr*`)`, where *expr* is a scalar integer expression or a boz constant, provides the object of enum type *type-name* that has integer value *expr*. For example, `season(7)` has the value `summer`. Values that represent no enumerator are permitted as long as they can be represented in the corresponding C integer type. An enum constructor whose *expr* is constant may appear in a `data` statement.

An integer expression is said to 'in type conformance' with an enum type if it contains a primary that is an enumerator of the type. In an assignment statement with a variable (left-hand side) of an enum type, the expression (right-hand side) must be of the same enum type or an integer expression that is in type conformance with it, for example

```
enum, bind(c) :: season
   enumerator :: spring=5, summer=7, autumn, winter
end enum
type(season) my_season, your_season
my_season = spring
your_season = autumn+1 ! winter
```

The intrinsic operations `==`, `/=`, `<`, `<=`, `>`, and `>=` are available when both operands are objects of the same enum type or one is of enum type and other is an integer expression that is in type conformance with it. They compare the integer values. No other intrinsic operations are available for objects of an enum type.

An array constructor with the usual syntax is available to construct an array of an enum type, for example, `[red, (colour(i),i=2,3)]` . All the array elements specified must be of a single enum type or be integer expressons that are in type conformance with it.

Enum types may be used in a `select case` construct, for example

```
select case (my_season)
   case (spring)
    :
   case (summer:)
    :
end select
```

When a colon is present in a `case` statement the enumerators are selected by their integer values.

An effective item of enum type in namelist or list-directed I/O is treated as if it were an integer with the item's integer value. In formatted I/O, an `i`, `b`, `o`, `z`, or `g` edit descriptor may be used.

An enum type that is declared in a module has the `private` or `public` attribute that is the default for its module unless it is included in a `private` or `public` statement.

10 Obsolete and deleted features

No more features have been added to the lists of obsolete and deleted features.

11 Acknowledgements

I would like to express my thanks to Bill Long, David Muxworthy, and Van Snyder for suggesting improvements to the first version, N2194, and to Steve Lionel and Reinhold Bader for their careful reading of this version of the paper. Reinhold suggested several corrections and improvements for this version.

References

ISO/IEC (2018), 'International Standard ISO/IEC 1539-1:2018 Information technology - Programming languages - Fortran - Part 1: Base language', *ISO/IEC*, Geneva.

ISO/IEC (2021), 'International Standard ISO/IEC 1539-1:2018/Cor 1:2021 Information technology - Programming languages - Fortran - Part 1: Base language - Technical Corrigendum 1', *ISO/IEC*, Geneva.

ISO/IEC (2023), 'International Standard ISO/IEC 1539-1:2018/Cor 2:2023 Information technology - Programming languages - Fortran - Part 1: Base language - Technical Corrigendum 2', *ISO/IEC*, Geneva.

Metcalf, M., Reid, J. and Cohen, M. (2018), *Modern Fortran Explained, Incorporating Fortran 2018*, Oxford University Press.