Fortran 2008 Overview

March 11, 2024

1 Introduction

This document describes those parts of the Fortran 2008 language which are not in Fortran 2003. These are all supported by the latest release of the NAG Fortran Compiler.

The compiler release in which a feature was made available is indicated by square brackets; for example, a feature marked as '[5.3]' was first available in Release 5.3.

2 Overview of Fortran 2008

Fortran 2008 is a major revision to Fortran 2003: the new language features can be grouped as follows:

- SPMD programming with coarrays;
- data declaration;
- data usage and computation;
- execution control;
- intrinsic procedures and modules;
- input/output extensions;
- programs and procedures.

3 SPMD programming with coarrays [6.2, 7.0]

3.1 Overview

Fortran 2008 contains an SPMD (Single Program Multiple Data) programming model, where multiple copies of a program, called "images", are executed in parallel. Special variables called "coarrays" facilitate communication between images.

Release 6.2 of the NAG Fortran Compiler limited execution to a single image, with no parallel execution. Release 7.0 of the NAG Fortran Compiler can execute multiple images in parallel on SMP machines, using Co-SMP technology.

3.2 Images

Each image contains its own variables and input/output units. The number of images at execution time is not determined by the program, but by some compiler-specific method. The number of images is fixed during execution; images cannot be created or destroyed. The intrinsic function `NUM_IMAGES()` returns the number of images. Each image has an "image index"; this is a positive integer from 1 to the number of images. The intrinsic function `THIS_IMAGE()` returns the image index of the executing image.

3.3 Coarrays

Coarrays are variables that can be directly accessed by another image; they must have the `ALLOCATABLE` or `SAVE` attribute or be a dummy argument.

A coarray has a "corank", which is the number of "codimensions" it has. Each codimension has a lower "cobound" and an upper cobound, determining the "coshape". The upper cobound of the last codimension is "`*`"; rather like an assumed-size array. The "cosubscripts" determine the image index of the reference, in the same way that the subscripts of an array determine the array element number. Again, like an assumed-size array, the image index must be less than or equal to the number of images.

A coarray can be a scalar or an array. It cannot have the `POINTER` attribute, but it can have pointer components.

As well as variables, coarray components are possible. In this case, the component must be an `ALLOCATABLE` coarray, and any variable with such a component must be a dummy argument or have the `SAVE` attribute.

3.4 Declaring coarrays

A coarray has a *coarray-spec* which is declared with square brackets after the variable name, or with the `CODIMENSION` attribute or statement. For example,

```
REAL a[100,*]
REAL,CODIMENSION[-10:10,-10:*] :: b
CODIMENSION c[*]
```

declares the coarray `A` to have corank 2 with lower "cobounds" both 1 and the first upper cobound 100, the coarray `B` to have corank 2 with lower cobounds both -10 and the first upper cobound 10, and the coarray `C` to have corank 1 and lower cobound 1. Note that for non-allocatable coarrays, the *coarray-spec* must always declare the last upper cobound with an asterisk, as this will vary depending on the number of images.

An `ALLOCATABLE` coarray is declared with a *deferred-coshape-spec*, for example,

```
REAL,ALLOCATABLE :: d[:,:,:,:]
```

declares the coarray `D` to have corank 4.

3.5 Accessing coarrays on other images

To access another image's copy of a coarray, cosubscripts are used following the coarray name in square brackets; this is called "coindexing", and such an object is a "coindexed object". For example, given

```
REAL,SAVE :: e[*]
```

the coindexed object `e[1]` refers to the copy of `E` on image 1, and `e[13]` refers to the copy of `E` on image 13. For a more complicated example: given

```
REAL,SAVE :: f[10,21:30,0:*]
```

the reference `f[3,22,1]` refers to the copy of `F` on image 113. There is no correlation between image numbers and any topology of the computer, so it is probably best to avoid complicated codimensions, especially if different coarrays have different coshape.

When a coarray is an array, you cannot put the cosubscripts directly after the array name, but must use array section notation instead. For example, with

```
REAL,SAVE :: g(10,10)[*]
```

the reference `g[inum]` is invalid, to refer to the whole array `G` on image `INUM` you need to use `g(:,:)[inum]` instead.

Similarly, to access a single element of `G`, the cosubscripts follow the subscripts, e.g. `g(i,j)[inum]`.

Finally, note that when a coarray is accessed, whether by its own image or remotely, the segment ordering rules (see next section) must be obeyed. This is to avoid nonsense answers from data races.

3.6 Segments and synchronisation

Execution on each image is divided into segments, by "image control statements". The segments on a single image are ordered: each segment follows the preceding segment. Segments on different images may be ordered (one following the other) by synchronisation, otherwise they are unordered.

If a coarray is defined (assigned a value) in a segment on image I, another image J is only allowed to reference or define it in a segment that follows the segment on I.

The image control statements, and their synchronisation effects, are as follows.

`SYNC ALL` synchronises with corresponding `SYNC ALL` statement executions on other images; the segment following the n^{th} execution of a `SYNC ALL` statement on one image follows all the segments that preceded the n^{th} execution of a `SYNC ALL` statement on every other image.

`SYNC IMAGES (`***list***`)`
synchronises with corresponding `SYNC IMAGES` statement executions on the images in *list*, which is an integer expression that may be scalar or a vector. Including the invoking image number in *list* has no effect. The segment following the n^{th} execution of a `SYNC IMAGES` statement on image I with the image number J in its *list* follows the segments on image J before its n^{th} execution of `SYNC IMAGES` with I in its *list*.

`SYNC IMAGES (*)`
is equivalent to `SYNC IMAGES` with every image no. in its *list*, e.g. `SYNC IMAGES ([(i,i=1,NUM_IMAGES())])`.

`SYNC MEMORY`
This only acts as a segment divider, without synchronising with any other image. It may be useful for user-defined orderings when some other mechanism has been used to synchronise.

`ALLOCATE` **or** `DEALLOCATE`
with a coarray object being allocated or deallocated. This synchronises all images, which must execute the same `ALLOCATE` or `DEALLOCATE` statement.

`CRITICAL` **and** `END CRITICAL`
Only one image can execute a `CRITICAL` construct at a time. The code inside a `CRITICAL` construct forms a segment, which follows the previous execution (on whatever image) of the `CRITICAL` construct.

`LOCK` **and** `UNLOCK`
The segment following a `LOCK` statements that locks a particular lock variable follows the `UNLOCK` statement that previously unlocked the variable.

`END` **statement**
An `END BLOCK`, `END FUNCTION`, or `END SUBROUTINE` statement that causes automatic deallocation of a local `ALLOCATABLE` coarray, synchronises with all images (which must execute the same `END` statement).

`MOVE_ALLOC` **intrinsic**
Execution of the intrinsic subroutine `MOVE_ALLOC` with coarray arguments synchronises all images, which must execute the same `CALL` statement.

Note that image control statements have side-effects, and therefore are not permitted in pure procedures or within `DO CONCURRENT` constructs.

3.7 Allocating and deallocating coarrays

When you allocate an `ALLOCATABLE` coarray, you must give the desired cobounds in the `ALLOCATE` statement. For example,

```
REAL,ALLOCATABLE :: x(:,:,:)[:,:]
...
ALLOCATE(x(100,100,3)[1:10,*])
```

Note that the last upper cobound must be an asterisk, the same as when declaring an explicit-coshape coarray.

When allocating a coarray there is a synchronisation: all images must execute the same `ALLOCATE` statement, and all the bounds, type parameters, and cobounds of the coarray must be the same on all images.

Similarly, there is a synchronisation when a coarray is deallocated, whether by a `DEALLOCATE` statement or automatic deallocation by an `END` statement; every image must execute the same statement.

Note that the usual automatic reallocation of allocatable variables in an intrinsic assignment statement, e.g. when the expression is an array of a different shape, is not available for coarrays. An allocatable coarray variable being assigned to must already be allocated and be conformable with the expression; furthermore, if it has deferred type parameters they must have the same values, and if it is polymorphic it must have the same dynamic type.

3.8 Critical constructs

The `CRITICAL` construct provides a mechanism for ensuring that only one image at a time executes a code segment. For example,

```
CRITICAL
  ...do something
END CRITICAL
```

If an image I arrives at the CRITICAL statement while another image J is executing the block of the construct, it will wait until image J has executed the `END CRITICAL` statement before continuing. Thus the `CRITICAL` — `END CRITICAL` segment on image I follows the equivalent segment on image J.

As a construct, this may have a name, e.g.

```
critsec: CRITICAL
  ...
END CRITICAL critsec
```

The name has no effect on the operation of the construct. Each `CRITICAL` construct is separate from all others, and has no effect on their execution.

3.9 Lock variables

A "lock variable" is a variable of the type `LOCK_TYPE`, defined in the intrinsic module `ISO_FORTRAN_ENV`. A lock variable must be a coarray, or a component of a coarray. It is initially "unlocked"; it is locked by execution of a `LOCK` statement, and unlocked by execution of an `UNLOCK` statement. Apart from those statements, it cannot appear in any variable definition context, other than as the actual argument for an `INTENT(INOUT)` dummy argument.

Execution of the segment after a `LOCK` statement successfully locks the variable follows execution of the segment before the `UNLOCK` statement on the image that unlocked it. For example,

```
INTEGER FUNCTION get_sequence_number()
  USE iso_fortran_env
  INTEGER :: number = 0
  TYPE(lock_type) lock[*]
  LOCK(lock[1])
  number = number + 1
  get_sequence_number = number
  UNLOCK(lock[1])
END FUNCTION
```

If the variable `lock` on image 1 is locked when the `LOCK` statement is executed, it will wait for it to become unlocked before continuing. Thus the function `get_sequence_number()` provides an one-sided ordering relation: the segment following a call that returned the value N will follow every segment that preceded a call that returned a value less than N.

Conditional locking is provided with the ACQUIRED_LOCK= specifier; if this specifier is present, the executing image only acquires the lock if it was previously unlocked. For example,

```
LOGICAL gotit
LOCK(lock[1],ACQUIRED_LOCK=gotit)
IF (gotit) THEN
  ! We have the lock.
ELSE
  ! We do not have the lock - some other image does.
END IF
```

It is an error for an image to try to LOCK a variable that is already locked to that image, or to UNLOCK a variable that is already unlocked, or that is locked to another image. If the STAT= specifier is used, these errors will return the values STAT_LOCKED, STAT_UNLOCKED, or STAT_LOCKED_OTHER_IMAGE respectively (these named constants are provided by the intrinsic module ISO_FORTRAN_ENV).

3.10 Atomic coarray accessing

As an exception to the segment ordering rules, a coarray that is an integer of kind ATOMIC_INT_KIND or a logical of kind ATOMIC_LOGICAL_KIND (these named constants are provided by the intrinsic module ISO_FORTRAN_ENV), can be defined with the intrinsic subroutine ATOMIC_DEFINE, or referenced by the intrinsic subroutine ATOMIC_REF. For example,

```
MODULE stopping
  USE iso_fortran_env
  LOGICAL(atomic_logical_kind),PRIVATE :: stop_flag[*] = .FALSE.
CONTAINS
  SUBROUTINE make_it_stop
    CALL atomic_define(stop_flag[1],.TRUE._atomic_logical_kind)
  END SUBROUTINE
  LOGICAL FUNCTION please_stop()
    CALL atomic_ref(please_stop,stop_flag[1])
  END FUNCTION
END MODULE
```

In this example, it is perfectly valid for any image to call make_it_stop, and for any other image to invoke the function please_stop(), without any regard for segments. (On a distributed memory machine it might take some time for changes to the atomic variable to be visible on other images, but they should eventually get the message.)

Note that ordinary assignment and referencing should not be mixed with calls to the atomic subroutines, as ordinary assignment and referencing are always subject to the segment ordering rules.

3.11 Normal termination of execution

If an image executes a STOP statement, or the END PROGRAM statement, normal termination is initiated. The other images continue execution, and all data on the "stopped" image remains; other images can continue to reference and define coarrays on the stopped image.

When normal termination has been initiated on all images, the program terminates.

3.12 Error termination

If any image terminates due to an error, for example an input/output error in an input/output statement that does not have any IOSTAT= or ERR= specifier, the entire program is error terminated. On a distributed memory machine it may take some time for the error termination messages to reach every image, so the termination might not be immediate.

The ERROR STOP statement initiates error termination.

3.13 Fault tolerance

The Fortran 2018 standard adds many features for detecting, simulating, and recovering from image failure. For example, the `FAIL IMAGE` statement causes the executing image to fail (stop responding to accesses from other images). These extensions are listed in the detailed syntax below, even though they are not part of the Fortran 2008 standard.

The `FAIL IMAGE` statement itself is not very useful when the number of images is equal to one, as it inevitably causes complete program failure.

3.14 Detailed syntax of coarray features

Coindexed object (data object designator):

> In a data object designator, a part (component or base object) that is a coarray can include an image selector: *part-name [* `(` *section-subscript-list* `)` *] [image-selector]*

where *part-name* identifies a coarray, and *image-selector* is

> *left-bracket cosubscript-list [* `,` *image-selector-spec] right-bracket*

The number of *cosubscript*s must be equal to the corank of *part-name*. If *image-selector* appears and *part-name* is an array, *section-subscript-list* must also appear. The optional *image-selector-spec* is Fortran 2018 (part of the fault tolerance feature), and is a comma-separated list of one or more of the following specifiers:

> `STAT =` *scalar-int-variable*
> `TEAM =` *team-value*
> `TEAM_NUMBER =` *scalar-int-expression*

A *team-value* must be a scalar expression of type `TEAM_TYPE` from the intrinsic module `ISO_FORTRAN_ENV`. The `STAT=` variable is assigned zero if the reference or definition was successful, and the value `STAT_FAILED` if the image referenced has failed.

`CRITICAL` construct:

> *[construct-name* `:` *] CRITICAL [* `(` *[sync-stat-list]* `)` *]*
> *block*
> `END CRITICAL` *[construct-name]*

where the optional *sync-stat-list* is a `STAT=` specifier, an `ERRMSG=` specifier, or both (separated by a comma). Note: The optional parentheses and *sync-stat-list* are Fortran 2018.

The *block* is not permitted to contain:

- a `RETURN` or `STOP` statement;
- an image control statement;
- a branch whose target is outside the construct.

`FAIL IMAGE` statement:

> `FAIL IMAGE`

Note: This statement is Fortran 2018.

`LOCK` statement:

> `LOCK (` *lock-variable [* `,` *lock-stat-list]* `)`

where *lock-stat-list* is a comma-separated list of one or more of the following:

```
ACQUIRED_LOCK = scalar-logical-variable
ERRMSG = scalar-default-character-variable
STAT = scalar-int-variable
```

and *lock-variable* is a scalar variable of type `LOCK_TYPE` from the intrinsic module `ISO_FORTRAN_ENV`.

`SYNC ALL` statement:

```
SYNC ALL [ ( [ sync-stat-list ] ) ]
```

`SYNC IMAGES` statement:

```
SYNC IMAGES ( image-set [ , sync-stat-list ] )
```

where *image-set* is an asterisk, or an integer expression that is scalar or of rank one.

`SYNC MEMORY` statement:

```
SYNC MEMORY [ ( [ sync-stat-list ] ) ]
```

`UNLOCK` statement:

```
UNLOCK ( lock-variable [ , sync-stat-list ] )
```

Note:

- The variables in *sync-stat-list* or *lock-stat-list* are not permitted to be coindexed objects, nor may they depend on anything else in the statement.

3.15 Intrinsic procedures and coarrays

```
SUBROUTINE ATOMIC_DEFINE(ATOM, VALUE, STAT)
```

`ATOM` is `INTENT(OUT)` scalar `INTEGER(ATOMIC_INT_KIND)` or `LOGICAL(ATOMIC_LOGICAL_KIND)`, and must be a coarray or a coindexed object.

`VALUE` is scalar with the same type as `ATOM`.

`STAT` *(Optional)* is scalar Integer and must have a decimal exponent range of at least four. It must not be coindexed.

The variable `ATOM` is atomically assigned the value of `VALUE`, without regard to the segment rules. If `STAT` is present, it is assigned a positive value if an error occurs, and zero otherwise. Note: `STAT` is part of Fortran 2018.

```
SUBROUTINE ATOMIC_REF(VALUE, ATOM, STAT)
```

`VALUE` is `INTENT(OUT)` scalar with the same type as `ATOM`.

`ATOM` is scalar `INTEGER(ATOMIC_INT_KIND)` or `LOGICAL(ATOMIC_LOGICAL_KIND)`, and must be a coarray or a coindexed object.

`STAT` *(Optional)* is scalar Integer and must have a decimal exponent range of at least four. It must not be coindexed.

The value of ATOM is atomically read, without regard to the segment rules, and then assigned to the variable VALUE. If STAT is present, it is assigned a positive value if an error occurs, and zero otherwise. Note: STAT is part of Fortran 2018.

```
INTEGER FUNCTION IMAGE_INDEX(COARRAY, SUB)
```

COARRAY a coarray of any type.

SUB an integer vector whose size is equal to the corank of COARRAY.

If the value of SUB is a valid set of cosubscripts for COARRAY, the value of the result is the image index of the image they will reference, otherwise the result has the value zero. For example, if X is declared with cobounds [11:20,13:*], the result of IMAGE_INDEX(X,[11,13]) will be equal to one, and the result of IMAGE_INDEX(x,[1,1]) will be equal to zero.

```
FUNCTION LCOBOUND(COARRAY, DIM , KIND)
```

COARRAY coarray of any type and corank N;

DIM *(Optional)* scalar Integer in the range 1 to N;

KIND *(Optional)* scalar Integer constant expression;

Result Integer or Integer(Kind=KIND).

If DIM appears, the result is scalar, being the value of the lower cobound of that codimension of COARRAY. If DIM does not appear, the result is a vector of length N containing all the lower cobounds of COARRAY. The actual argument for DIM must not itself be an optional dummy argument.

```
SUBROUTINE MOVE_ALLOC(FROM, TO, STAT, ERRMSG)  ! Revised
```

FROM an allocatable variable of any type.

TO an allocatable with the same declared type, type parameters, rank and corank, as FROM.

STAT INTENT(OUT) scalar Integer with a decimal exponent range of at least four.

ERRMSG INTENT(INOUT) scalar default character variable.

If FROM and TO are coarrays, the CALL statement is an image control statement that synchronises all images. If STAT is present, it is assigned a positive value if any error occurs, otherwise it is assigned the value zero. If ERRMSG is present and an error occurs, it is assigned an explanatory message. Note: The STAT and ERRMSG arguments are Fortran 2018.

```
INTEGER FUNCTION NUM_IMAGES()
```

This intrinsic function returns the number of images. In this release of the NAG Fortran Compiler, the value will always be equal to one.

```
INTEGER FUNCTION THIS_IMAGE()
```

Returns the image index of the executing image.

```
FUNCTION THIS_IMAGE(COARRAY)
```

Returns an array of type Integer with default kind, with the size equal to the corank of COARRAY, which may be a coarray of any type. The values returned are the cosubscripts for COARRAY that correspond to the executing image.

```
INTEGER FUNCTION THIS_IMAGE(COARRAY, DIM)
```

`COARRAY` is a coarray of any type.

`DIM` is scalar Integer.

Returns the cosubscript for the codimension `DIM` that corresponds to the executing image. Note: In Fortran 2008 `DIM` was not permitted to be an optional dummy argument; Fortran 2018 permits that.

```
FUNCTION UCOBOUND(COARRAY, DIM, KIND)
```

`COARRAY` coarray of any type and corank N;

`DIM` *(Optional)* scalar Integer in the range 1 to N;

`KIND` *(Optional)* scalar Integer constant expression;

Result Integer or Integer(Kind=KIND).

If `DIM` appears, the result is scalar, being the value of the upper cobound of that codimension of `COARRAY`. If `DIM` does not appear, the result is a vector of length N containing all the upper cobounds of `COARRAY`. The actual argument for `DIM` must not itself be an optional dummy argument.

Note that if `COARRAY` has corank $N>1$, and the number of images in the current execution is not an integer multiple of the coextents up to codimension $N-1$, the images do not make a full rectangular pattern. In this case, the value of the last upper cobound is the maximum value that a cosubscript can take for that codimension; e.g. with a coarray-spec of `[1:3,1:*]` and four images in the execution, the last upper cobound will be equal to 2 because the cosubscripts `[1,2]` are valid even though `[2,2]` and `[2,3]` are not.

4 Data declaration [mostly 6.0]

- The maximum rank of an array has been increased from 7 to 15. For example,

  ```
  REAL array(2,2,2,2,2,2,2,2,2,2,2,2,2,2,2)
  ```

  declares a 15-dimensional array.

- [3.0] 64-bit integer support is required, that is, the result of `SELECTED_INT_KIND(18)` is a valid integer kind number.

- A named constant (`PARAMETER`) that is an array can assume its shape from its defining expression; this is called an implied-shape array. The syntax is that the upper bound of every dimension must be an asterisk, for example

  ```
  REAL,PARAMETER :: idmat3(*,*) = Reshape( [ 1,0,0,0,1,0,0,0,1 ], [ 3,3 ] )
  REAL,PARAMETER :: yeardata(2000:*) = [ 1,2,3,4,5,6,7,8,9 ]
  ```

  declares `idmat3` to have the bounds `(1:3,1:3)`, and `yeardata` to have the bounds `(2000:2008)`.

- The `TYPE` keyword can be used to declare entities of intrinsic type, simply by putting the intrinsic *type-spec* within the parentheses. For example,

  ```
  TYPE(REAL) x
  TYPE(COMPLEX(KIND(0d0))) y
  TYPE(CHARACTER(LEN=80)) z
  ```

  is completely equivalent, apart from being more confusing, to

  ```
  REAL x
  COMPLEX(KIND(0d0)) y
  CHARACTER(LEN=80) z
  ```

- As a consequence of the preceding extension, it is no longer permitted to define a derived type that has the name `DOUBLEPRECISION`.
- [5.3] A type-bound procedure declaration statement may now declare multiple type-bound procedures. For example, instead of

```
PROCEDURE,NOPASS :: a
PROCEDURE,NOPASS :: b=>x
PROCEDURE,NOPASS :: c
```

  the single statement

```
PROCEDURE,NOPASS :: a, b=>x, c
```

  will suffice.
- [5.3 for `C_ASSOCIATED`, 7.0 for `C_LOC` and `C_FUNLOC`] A specification expression may now use the `C_ASSOCIATED`, `C_LOC` and `C_FUNLOC` functions from the `ISO_C_BINDING` module. For example, given a `TYPE(C_PTR)` variable X and another interoperable variable `Y` with the `TARGET` attribute,

```
INTEGER workspace(MERGE(10,20,C_ASSOCIATED(X,C_LOC(Y))))
```

  is allowed, and will give `workspace` a size of 10 elements if the C pointer `X` is associated with `Y`, and 20 elements otherwise.
- [7.0] A specification expression may now use a user-defined operation, provided that operation is provided by a specification function. (A specification function must be a pure function that is not a statement function or internal function, and that does not have a dummy procedure argument.) For example, given the interface block

```
INTERFACE OPERATOR(.user.)
   PURE INTEGER FUNCTION userfun(x)
      REAL,INTENT(IN) :: x
   END FUNCTION
END INTERFACE
```

  the user-defined operator `.user.` may be used in a specification expression as follows:

```
LOGICAL mask(.user.(3.145))
```

  Note that this applies to overloaded intrinsic operators as well as user-defined operators.
- [7.1] An allocatable component can forward-reference a type, for example:

```
Type t2
  Type(t),Pointer :: p
  Type(t),Allocatable :: a
End Type
Type t
  Integer c
End Type
```

  An allocatable component can also be of recursive type, or two types can be mutually recursive. For example,

```
Type t
  Integer v
  Type(t),Allocatable :: a
End Type
```

  This allows lists and trees to be built using allocatable components. Building or traversing such data structures will usually require recursive procedure calls, as there is no allocatable analogue of pointer assignment.

  No matter how deeply nested such recursive data structures become, they can never be circular (again, because there is no pointer assignment). As usual, deallocating the top object of such a structure will recursively deallocate all its allocatable components.

- [7.1] A dummy argument can be used in a specification expression in an elemental subprogram, as long as it is not used to specify a type parameter (such as character length) of a function result. For type parameters of function results, they remain limited to appearing in specification enquiries (such as `LEN`) when the enquiry is not about a deferred characteristic.

  For example, in

  ```
  Elemental Subroutine s(x,n,y)
    Real,Intent(In) :: x
    Integer,Intent(In) :: n
    Real,Intent(Out) :: y
    Real temp(n)
    ...
  ```

  the dummy argument `N` can be used to declare the local array `TEMP`.

- [7.1] Pointers and pointer components can be initialised to point to a target. The target must be valid for that pointer (e.g. same type, rank, etc.). The main cases are:

  Named pointer initialisation

  For data pointers, the target must have the `SAVE` attribute (variables in modules and the main program have this attribute implicitly). For procedure pointers, the target must be a module procedure or external procedure, not a dummy procedure, internal procedure, or statement function.

  For example,

  ```
  Module m
    Real,Target :: x
    Real,Pointer :: p => x
  End Module
  Program test
    Use m
    p = 3
    Print *,x ! Will print the value 3.0
  End Program
  ```

  Component default initialisation

  Pointer components can be default-initialised to point to a target. The requirements on the target are the same as for named pointer initialisation.

  For example,

  ```
  Module m
    Real,Target :: x
    Type t
      Real,Pointer :: p => x
    End Type
  End Module
  Program test
    Use m
    Type(t) y
    y%p = 3
    Print *,x ! Will print the value 3.0
  End Program
  ```

  Component initialisation with structure constructors

  A structure constructor in a constant expression can specify a target for any pointer component. The requirements on the target are the same as for named pointer initialisation.

  For example,

  ```
  Module m
    Real,Target :: x
    Type t
      Real,Pointer :: p
    End Type
  ```

```
End Module
Program test
  Use m
  Type(t) :: y = t(x)
  y%p = 3
  Print *,x ! Will print the value 3.0
End Program
```

- [7.1] A reference to a function that returns a pointer can be used as a variable in many contexts. In particular, it can be used as the variable in an assignment statement, as the actual argument corresponding to an `INTENT(OUT)` or `INTENT(INOUT)` dummy argument, and as the selector in an `ASSOCIATE` or `SELECT TYPE` construct that modifies the associate-name.

  For example, with this module,

```
Module m
  Real,Target,Save :: table(100) = 0
Contains
  Function f(n)
    Integer,Intent(In) :: n
    Real,Pointer :: f
    f => table(Min(Max(1,n),Size(table)))
  End Function
End Module
```

  the program below will print "`-1.23E+02`".

```
Program example
  Use m
  f(13) = -123
  Print 1,f(13)
1 Format(ES10.3)
End Program
```

  It should be noted that the syntax of a statement function definition is identical to part of the syntax of a pointer function reference as a variable; the existence of a pointer-valued function that is accessible in the scope determines which of these it is. This may lead to confusing error messages in some situations.

  With the above module, this program demonstrates the use of the feature with an `ASSOCIATE` construct.

```
Program assoc_eg
  Use m
  Associate(x=>f(3), y=>f(4))
    x = 0.5
    y = 3/x
  End Associate
  Print 1,table(3:4) ! Will print "  5.00E-01  6.00E+00"
1 Format(2ES10.2)
End Program
```

  Finally, here is an example using argument passing.

```
Program argument_eg
  Use m
  Call set(f(7))
  Print 1,table(7) ! Will print "1.41421"
1 Format(F7.5)
Contains
  Subroutine set(x)
    Real,Intent(Out) :: x
    x = Sqrt(2.0)
  End Subroutine
End Program
```

Other contexts where a reference to a pointer-valued function may be used instead of a variable designator include:

  - as an internal file specifier in a `WRITE` statement (the function must return a pointer to a character string or array for this);
  - as an input-item in a `READ` statement;
  - as a `STAT=` or `ERRMSG=` variable in an `ALLOCATE` or `DEALLOCATE` statement, or in an image control statement such as `EVENT WAIT`;
  - as the team variable in a `FORM TEAM` statement.

- [7.1] The result of a function can be a procedure pointer. For example,

```
Module ppfun
  Private
  Abstract Interface
    Subroutine charsub(string)
      Character(*),Intent(In) :: string
    End Subroutine
  End Interface
  Public charsub,hello_goodbye
Contains
  Subroutine hello(string)
    Character(*),Intent(In) :: string
    Print *,'Hello: ',string
  End Subroutine
  Subroutine bye(string)
    Character(*),Intent(In) :: string
    Print *,'Goodbye: ',string
    Stop
  End Subroutine
  Function hello_goodbye(flag)
    Logical,Intent(In) :: flag
    Procedure(hello),Pointer :: hello_goodbye
    If (flag) Then
      hello_goodbye => hello
    Else
      hello_goodbye => bye
    End If
  End Function
End Module
Program example
  Use ppfun
  Procedure(charsub),Pointer :: pp
  pp => hello_goodbye(.True.)
  Call pp('One')
  pp => hello_goodbye(.False.)
  Call pp('Two')
End Program
```

The function `hello_goodbye` in module `ppfun` returns a pointer to a procedure, which needs to be pointer-assigned to a procedure pointer to be invoked. When executed, this example will print

```
Hello: One
Goodbye: Two
```

Use of this feature is not recommended, as it blurs the lines between data objects and procedures; this may lead to confusion or misunderstandings during code maintenance. The feature provides no functionality that was not already provided by procedure pointer components.

5 Data usage and computation [mostly 5.3]

- In a structure constructor, the value for an allocatable component may be omitted: this has the same effect as specifying `NULL()`.
- [6.0] When allocating an array with the `ALLOCATE` statement, if `SOURCE=` or `MOLD=` is present and its expression is an array, the array can take its shape directly from the expression. This is a lot more concise than using `SIZE` or `UBOUND`, especially for a multi-dimensional array.

  For example,

```
SUBROUTINE s(x,mask)
   REAL x(:,:,:)
   LOGICAL mask(:,:,:)
   REAL,ALLOCATABLE :: y(:,:,:)
   ALLOCATE(y,MOLD=x)
   WHERE (mask)
     y = 1/x
   ELSEWHERE
     y = HUGE(x)
   END WHERE
   ! ...
END SUBROUTINE
```

- [6.2] An `ALLOCATE` statement with the `SOURCE=` clause is permitted to have more than one *allocation*. The *source-expr* is assigned to every variable allocated in the statement. For example,

```
PROGRAM multi_alloc
   INTEGER,ALLOCATABLE :: x(:),y(:,:)
   ALLOCATE(x(3),y(2,4),SOURCE=42)
   PRINT *,x,y
END PROGRAM
```

  will print the value "42" eleven times (the three elements of `x` and the eight elements of `y`). If the *source-expr* is an array, every *allocation* needs to have the same shape.

- [6.1] The real and imaginary parts of a `COMPLEX` object can be accessed using the complex part designators '`%RE`' and '`%IM`'. For example, given

```
COMPLEX,PARAMETER :: c = (1,2), ca(2) = [ (3,4),(5,6) ]
```

  the designators `c%re` and `c%im` have the values 1 and 2 respectively, and `ca%re` and `ca%im` are arrays with the values `[ 3,5 ]` and `[ 4,6 ]` respectively. In the case of variables, for example

```
COMPLEX :: v, va(10)
```

  the real and imaginary parts can also be assigned to directly; the statement

```
va%im = 0
```

  will set the imaginary part of each element of `va` to zero without affecting the real part.

- In an `ALLOCATE` statement for one or more variables, the `MOLD=` clause can be used to give the variable(s) the dynamic type and type parameters (and optionally shape) of an expression. The expression in `MOLD=` must be type-compatible with each allocate-object, and if the expression is a variable (e.g. `MOLD=X`), the variable need not be defined. Note that the `MOLD=` clause may appear even if the type, type parameters and shape of the variable(s) being allocated are not mutable. For example,

```
CLASS(*),POINTER :: a,b,c
ALLOCATE(a,b,c,MOLD=125)
```

will allocate the unlimited polymorphic pointers `A`, `B` and `C` to be of type Integer (with default kind); unlike `SOURCE=`, the values of `A`, `B` and `C` will be undefined.

- [5.3.1] Assignment to a polymorphic allocatable variable is permitted. If the variable has different dynamic type or type parameters, or if an array, a different shape, it is first deallocated. If it is unallocated (or is deallocated by step 1), it is then allocated to have the correct type and shape. It is then assigned the value of the expression. Note that the operaton of this feature is similar to the way that `ALLOCATE(`*variable*`,SOURCE=`*expr*`)` works. For example, given

  ```
  CLASS(*),ALLOCATABLE :: x
  ```

  execution of the assignment statement

  ```
  x = 43
  ```

  will result in `X` having dynamic type Integer (with default kind) and value 43, regardless of whether `X` was previously unallocated or allocated with any other type (or kind).

- [6.1] Rank-remapping pointer assignment is now permitted when the target has rank greater than one, provided it is "simply contiguous" (a term which means that it must be easily seen at compile-time to be contiguous). For example, the pointer assignment in

  ```
  REAL,TARGET :: x(100,100)
  REAL,POINTER :: x1(:)
  x1(1:Size(x)) => x
  ```

  establishes `X1` as a single-dimensional alias for the whole of `X`.

6 Execution control [mostly 6.0]

- [5.3] The `BLOCK` construct allows declarations of entities within executable code. For example,

  ```
  Do i=1,n
    Block
      Real tmp
      tmp = a(i)**3
      If (tmp>b(i)) b(i) = tmp
    End Block
  End Do
  ```

  Here the variable `tmp` has its scope limited to the `BLOCK` construct, so will not affect anything outside it. This is particularly useful when including code by `INCLUDE` or by macro preprocessing.

  All declarations are allowed within a `BLOCK` construct except for `COMMON`, `EQUIVALENCE`, `IMPLICIT`, `INTENT`, `NAMELIST`, `OPTIONAL` and `VALUE`; also, statement function definitions are not permitted.

  `BLOCK` constructs may be nested; like other constructs, branches into a `BLOCK` construct from outside are not permitted. A branch out of a `BLOCK` construct "completes" execution of the construct.

  Entities within a `BLOCK` construct that do not have the `SAVE` attribute (including implicitly via initialisation), will cease to exist when execution of the construct is completed. For example, an allocated `ALLOCATABLE` variable will be automatically deallocated, and a variable with a `FINAL` procedure will be finalised.

- The `EXIT` statement is no longer restricted to exiting from a `DO` construct; it can now be used to jump to the end of a named `ASSOCIATE`, `BLOCK`, `IF`, `SELECT CASE` or `SELECT TYPE` construct (i.e. any named construct except `FORALL` and `WHERE`). Note that an `EXIT` statement with no *construct-name* still exits from the innermost `DO` construct, disregarding any other named constructs it might be within.

- In a `STOP` statement, the *stop-code* may be any scalar constant expression of type integer or default character. (In the NAG Fortran Compiler this also applies to the `PAUSE` statement, but that statement is no longer standard Fortran.) Additionally, the `STOP` statement with an integer *stop-code* now returns that value as the process exit status (on most operating systems there are limits on the value that can be returned, so for the NAG Fortran Compiler this returns only the lower eight bits of the value).

- The `ERROR STOP` statement has been added. This is similar to the `STOP` statement, but causes error termination rather than normal termination. The syntax is identical to that of the `STOP` statement apart from the extra keyword '`ERROR`' at the beginning. Also, the default process exit status is zero for normal termination, and non-zero for error termination.

  For example,

  ```
  IF (x<=0) ERROR STOP 'x must be positive'
  ```

- [6.1] The `FORALL` construct now has an optional type specifier in the initial statement of the construct, which can be used to specify the type (which must be `INTEGER`) and kind of the index variables. When this is specified, the existence or otherwise of any entity in the outer scope that has the same name as an index variable does not affect the index variable in any way. For example,

  ```
  Complex i(100)
  Real x(200)
  ...
  Forall (Integer :: i=1:Size(x)) x(i) = i
  ```

  Note that the `FORALL` construct is still not recommended for high performance, as the semantics imply evaluating the right-hand sides into array temps the size of the iteration space, and then assigning to the variables; this usually performs worse than ordinary `DO` loops.

- [6.1] The `DO CONCURRENT` construct is a `DO` loop with restrictions and semantics intended to allow efficient execution. The iterations of a `DO CONCURRENT` construct may be executed in any order, and possibly even in parallel. The loop index variables are local to the construct.

  The `DO CONCURRENT` header has similar syntax to the `FORALL` header, including the ability to explicitly specify the type and kind of the loop index variables, and including the scalar mask.

  The restrictions on the `DO CONCURRENT` construct are:

  - no branch is allowed from within the construct to outside of it (this includes the `RETURN` and `STOP` statements, but `ERROR STOP` is allowed);
  - the `EXIT` statement cannot be used to terminate the loop;
  - the `CYCLE` statement cannot refer to an outer loop;
  - there must be no dependencies between loop iterations, and if a variable is assigned to by any iteration, it is not allowed to be referenced by another iteration unless that iteration assigns it a value first;
  - all procedures referenced within the construct must be pure;
  - no image control statements can appear within the loop;
  - no reference to `IEEE_GET_FLAG` or `IEEE_SET_HALTING_MODE` is allowed.

  For example,

  ```
  Integer vsub(n)
  ...
  Do Concurrent (i=1:n)
    ! Safe because vsub has no duplicate values.
    x(vsub(i)) = i
  End Do
  ```

  The full syntax of the `DO CONCURRENT` statement is:

  [do-construct-name `:` *]* `DO` *[label] [* `,` *]* `CONCURRENT` *forall-header*

  where *forall-header* is

  `(` *[integer-type-spec* `::` *] triplet-spec [* `,` *triplet-spec]... [* `,` *mask-expr]* `)`

  where *mask-expr* is a scalar logical expression, and *triplet-spec* is

  name = *expr* : *expr [* : *expr]*

7 Intrinsic procedures and modules

7.1 Additional mathematical intrinsic functions [mostly 5.3.1]

- The elemental intrinsic functions `ACOSH`, `ASINH` and `ATANH` compute the inverse hyperbolic cosine, sine or tangent respectively. There is a single argument `X`, which may be of type Real or Complex; the result of the function has the same type and kind. When the argument is Complex, the imaginary part is expressed in radians and lies in the range $0 \leq im \leq \pi$ for the `ACOSH` function, and $-\pi/2 \leq im \leq \pi/2$ for the `ASINH` and `ATANH` functions.

  For example, `ACOSH(1.543081)`, `ASINH(1.175201)` and `ATANH(0.7615942)` are all approximately equal to `1.0`.

- [6.1] The new elemental intrinsic functions `BESSEL_J0`, `BESSEL_Y0`, `BESSEL_J1` and `BESSEL_Y1` compute the Bessel functions J_0, Y_0, J_1 and Y_1 respectively. These functions are solutions to Bessel's differential equation. The J functions are of the 1^{st} kind and the Y functions are of the 2^{nd} kind; the following subscript indicates the order (0 or 1). There is a single argument `X`, which must be of type Real; the result of the function has the same type and kind. For functions of the 2^{nd} kind (`BESSEL_Y0` and `BESSEL_Y1`), the argument `X` must be positive.

  For example, `BESSEL_J0(1.5)` is approximately `0.5118276`, `BESSEL_Y0(1.5)` is approximately `0.3824489`, `BESSEL_J1(1.5)` is approximately `0.5579365` and `BESSEL_Y1(1.5)` is approximately `-0.4123086`.

- [6.1] The new intrinsic functions `BESSEL_JN` and `BESSEL_YN` compute the Bessel functions J_n and Y_n respectively. These functions come in two forms: an elemental form and a transformational form.

  The elemental form has two arguments: `N`, the order of the function to compute, and `X`, the argument of the Bessel function. `BESSEL_JN(0,X)` is identical to `BESSEL_J0(X)`, etc..

  The transformational form has three scalar arguments: `N1`, `N2` and `X`. The result is a vector of size `MAX(N2-N1+1,0)`, containing approximations to the Bessel functions of orders `N1` to `N2` applied to `X`.

  For example, `BESSEL_JN(5,7.5)` is approximately `0.283474`, `BESSEL_YN(5,7.5)` is approximately `0.175418`, `BESSEL_JN(3,5,7.5)` is approximately `[ -0.258061, 0.023825, 0.283474 ]` and `BESSEL_YN(3,5,7.5)` is approximately `[ 0.159708, 0.314180, 0.175418 ]`.

- [6.0] The elemental intrinsic functions `ERF`, `ERFC` and `ERFC_SCALED` compute the error function, the complementary error function and the scaled complementary error function, respectively. The single argument `X` must be of type real.

  The error function is the integral of $-t^2$ from 0 to `X`, times $2/$`SQRT`(π); this rapidly converges to 1. The complementary error function is 1 minus the error function, and fairly quickly converges to zero. The scaled complementary error function scales the value (of 1 minus the error function) by `EXP(X**2)`; this also converges to zero but only very slowly.

- [6.0] The elemental intrinsic functions `GAMMA` and `LOG_GAMMA` compute the gamma function and the natural logarithm of the absolute value of the gamma function respectively. The single argument `X` must be of type real, and must not be zero or a negative integer.

  The gamma function is the extension of factorial from the integers to the reals; for positive integers, `GAMMA(X)` is equal to `(X-1)`!, i.e. factorial of `X-1`. This grows very rapidly and thus overflows for quite small `X`; `LOG_GAMMA` also diverges but much more slowly.

- The elemental intrinsic function `HYPOT` computes the "Euclidean distance function" (square root of the sum of squares) of its arguments `X` and `Y` without overflow or underflow for very large or small `X` or `Y` (unless the result itself overflows or underflows). The arguments must be of type Real with the same kind, and the result is of type Real with that kind. Note that `HYPOT(X,Y)` is semantically and numerically equal to `ABS(CMPLX(X,Y,KIND(X)))`.

  For example, `HYPOT(3e30,4e30)` is approximately equal to `5e30`.

- The array reduction intrinsic function `NORM2(X,DIM)` reduces Real arrays using the L_2-norm operation. This operates exactly the same as `SUM` and `PRODUCT`, except for the operation involved. The L_2 norm of an array is the square root of the sum of the squares of the elements. Note that unlike most of the other reduction functions, `NORM2` does not have a `MASK` argument. The `DIM` argument is optional; an actual argument for `DIM` is not itself permitted to be an optional dummy argument.

  The calculation of the result value is done in such a way as to avoid intermediate overflow and underflow, except when the result itself is outside the maximum range. For example, `NORM2([X,Y])` is approximately the same as `HYPOT(X,Y)`.

7.2 Additional intrinsic functions for bit manipulation [mostly 5.3]

- The elemental intrinsic functions BGE, BGT, BLE and BLT perform bitwise (i.e. unsigned) comparisons. They each have two arguments, I and J, which must be of type Integer but may be of different kind. The result is default Logical.

  For example, BGE(INT(Z'FF',INT8),128) is true, while INT(Z'FF',INT8)>=128 is false.

- [5.3.1] The elemental intrinsic functions DSHIFTL and DSHIFTR perform double-width shifting. They each have three arguments, I, J and SHIFT which must be of type Integer, except that one of I or J may be a BOZ literal constant – it will be converted to the type and kind of the other I or J argument. I and J must have the same kind if they are both of type Integer. The result is of type Integer, with the same kind as I and J. The I and J arguments are effectively concatenated to form a single double-width value, which is shifted left or right by SHIFT positions; for DSHIFTL the result is the top half of the combined shift, and for DSHIFTR the result is the bottom half of the combined shift.

  For example, DSHIFTL(INT(B'11000101',1),B'11001001',2) has the value INT(B'00010111',1) (decimal value 23), whereas DSHIFTR(INT(B'11000101',1),B'11001001',2) has the value INT(B'01110010',1) (decimal value 114).

- The array reduction intrinsic functions IALL, IANY and IPARITY reduce arrays using bitwise operations. These are exactly the same as SUM and PRODUCT, except that instead of reducing the array by the + or * operation, they reduce it by the IAND, IOR and IEOR intrinsic functions respectively. That it, each element of the result is the bitwise-and, bitwise-or, or bitwise-exclusive-or of the reduced elements. If the number of reduced elements is zero, the result is zero for IANY and IPARITY, and NOT(*zero*) for IALL.

- The elemental intrinsic functions LEADZ and TRAILZ return the number of leading (most significant) and trailing (least significant) zero bits in the argument I, which must be of type Integer (of any kind). The result is default Integer.

- The elemental intrinsic functions MASKL and MASKR generate simple left-justified and right-justified bitmasks. The value of MASKL(I,KIND) is an integer with the specified kind that has its leftmost I bits set to one and the rest set to zero; I must be non-negative and less than or equal to the bitsize of the result. If KIND is omitted, the result is default integer. The value of MASKR is similar, but has its rightmost I bits set to one instead.

- [5.3.1] The elemental intrinsic function MERGE_BITS(I,J,MASK) merges the bits from Integer values I and J, taking the bit from I when the corresponding bit in MASK is 1, and taking the bit from J when it is zero. All arguments must be BOZ literal constants or of type Integer, and all the Integer arguments must have the same kind; at least one of I and J must be of type Integer, and the result has the same type and kind.

  Note that MERGE_BITS(I,J,MASK) is identical to IOR(IAND(I,MASK),IAND(J,NOT(MASK))).

  For example, MERGE_BITS(INT(B'00110011',1),B'11110000',B'10101010') is equal to INT(B'01110010') (decimal value 114).

- The array reduction intrinsic function PARITY reduces Logical arrays. It is exactly the same as ALL and ANY, except that instead of reducing the array by the .AND. or .OR. operation, it reduces it by the .NEQV. operation. That is, each element of the result is .TRUE. if an odd number of reduced elements is .TRUE..

- The elemental intrinsic function POPCNT(I) returns the number of bits in the Integer argument I that are set to 1. The elemental intrinsic function POPPAR(I) returns zero if the number of bits in I that are set to 1 are even, and one if it is odd. The result is default Integer.

7.3 Other new intrinsic procedures [mostly 5.3.1]

- The intrinsic subroutine EXECUTE_COMMAND_LINE passes a command line to the operating system's command processor for execution. It has five arguments, in order these are:
  CHARACTER(*),INTENT(IN) :: COMMAND — the command to be executed;
  LOGICAL,INTENT(IN),OPTIONAL :: WAIT — whether to wait for command completion (default true);
  INTEGER,INTENT(INOUT),OPTIONAL :: EXITSTAT — the result value of the command;
  INTEGER,INTENT(OUT),OPTIONAL :: CMDSTAT — see below;
  CHARACTER(*),INTENT(INOUT),OPTIONAL :: CMDMSG — the error message if CMDSTAT is non-zero.

  CMDSTAT values are zero for success, −1 if command line execution is not supported, −2 if WAIT is present and false but asynchronous execution is not supported, and a positive value to indicate some other error. If CMDSTAT

is not present but would have been set non-zero, the program will be terminated. Note that Release 5.3.1 supports command line execution on all systems, and does not support asynchronous execution on any system.

For example, `CALL EXECUTE_COMMAND_LINE('echo Hello')` will probably display '`Hello`' in the console window.

- The intrinsic function `STORAGE_SIZE(A,KIND)` returns the size in bits of a scalar object with the same dynamic type and type parameters as `A`, when it is stored as an array element (i.e. including any padding). The `KIND` argument is optional; the result is type Integer with kind `KIND` if it is present, and default kind otherwise.

  If `A` is allocatable or a pointer, it does not have to be allocated unless it has a deferred type parameter (e.g. `CHARACTER(:)`) or is `CLASS(*)`. If it is a polymorphic pointer, it must not have an undefined status.

  For example, `STORAGE_SIZE(13_1)` is equal to 8 (bits).

- [6.0] The intrinsic inquiry function `IS_CONTIGUOUS` has a single argument `ARRAY`, which can be an array of any type. The function returns true if `ARRAY` is stored contiguously, and false otherwise. Note that this question has no meaning for an array with no elements, or for an array expression since that is a value and not a variable.

- [7.0] The intrinsic function `FINDLOC` is similar to `MAXLOC` and `MINLOC`, but instead of finding the location of the maximum or minimum value of an array, it finds a location that is equal to a specified value; thus it is available for all intrinsic types including `COMPLEX` and `LOGICAL`. It has one of the following two forms:

  ```
  FINDLOC (ARRAY, VALUE, DIM, MASK , KIND , BACK )
  FINDLOC (ARRAY, VALUE, MASK , KIND , BACK )
  ```

  where

`ARRAY`	is an array of intrinsic type, with rank N;
`VALUE`	is a scalar of the same type (if `LOGICAL`) or which may be compared with `ARRAY` using the intrinsic operator `==` (or `.EQ.`);
`DIM`	is a scalar `INTEGER` in the range 1 to N;
`MASK`	(optional) is an array of type `LOGICAL` with the same shape as `ARRAY`
`KIND`	(optional) is a scalar `INTEGER` constant expression that is a valid Integer kind number;
`BACK`	(optional) is a scalar `LOGICAL` value.

  The result of the function is type `INTEGER`, or `INTEGER(KIND)` if `KIND` is present.

  In the form without `DIM`, the result is a vector of length N, and is the location of the element of `ARRAY` that is equal to `VALUE`; if `MASK` is present, only elements for which the corresponding element of `MASK` are `.TRUE.` are considered. As in `MAXLOC` and `MINLOC`, the location is reported with 1 for the first element in each dimension; if no element equal to `VALUE` is found, the result is zero. If `BACK` is present with the value `.TRUE.`, the element found is the last one (in array element order); otherwise, it is the first one.

  In the form with `DIM`, the result has rank $N-1$ (thus scalar if `ARRAY` is a vector), the shape being that of `ARRAY` with dimension `DIM` removed, and each element of the result is the location of the (masked) element in the dimension `DIM` vector that is equal to `VALUE`.

  For example, if `ARRAY` is an Integer vector with value `[ 10,20,30,40,50 ]`, `FINDLOC(ARRAY,30)` will return the vector `[ 3 ]` and `FINDLOC(ARRAY,7)` will return the vector `[ 0 ]`.

7.4 Changes to existing intrinsic procedures [mostly 5.3.1]

- The intrinsic functions `ACOS`, `ASIN`, `ATAN`, `COSH`, `SINH`, `TAN` and `TANH` now accept arguments of type Complex. Note that the hyperbolic and non-hyperbolic versions of these functions and the new `ACOSH`, `ASINH` and `ATANH` functions are all related by simple algebraic identities, for example the new `COSH(X)` is identical to the old `COS((0,1)*X)` and the new `SINH(X)` is identical to the old `(0,-1)*SIN((0,1)*X)`.

- The intrinsic function `ATAN` now has an extra form `ATAN(Y,X)`, with exactly the same semantics as `ATAN2(Y,X)`.

- [6.2] The intrinsic functions `MAXLOC` and `MINLOC` now have an additional optional argument `BACK` following the `KIND` argument. It is scalar and of type Logical; if present with the value `.True.`, if there is more than one element that has the maximum value (for `MAXLOC`) or minimum value (for `MINLOC`), the array element index returned is for the last element with that value rather than the first.

  For example, the value of

  ```
  MAXLOC( [ 5,1,5 ], BACK=.TRUE.)
  ```

is the array `[ 3 ]`, rather than `[ 1 ]`.

- The intrinsic function `SELECTED_REAL_KIND` now has a third argument `RADIX`; this specifies the desired radix of the Real kind requested. Note that the function `IEEE_SELECTED_REAL_KIND` in the intrinsic module `IEEE_ARITHMETIC` also has this new third argument, and will allow requesting IEEE decimal floating-point kinds if they become available in the future.

7.5 `ISO_C_BINDING` additions [6.2]

The standard intrinsic module `ISO_C_BINDING` contains an additional procedure as follows.

```
INTERFACE c_sizeof
  PURE INTEGER(c_size_t) FUNCTION c_sizeof...(x) ! Specific name not visible
    TYPE(*) :: x(..)
  END FUNCTION
END INTERFACE
```

The actual argument `x` must be interoperable. The result is the same as the C `sizeof` operator applied to the conceptually corresponding C entity; that is, the size of `x` in bytes. If `x` is an array, it is the size of the whole array, not just one element. Note that `x` cannot be an assumed-size array.

7.6 `ISO_FORTRAN_ENV` additions

[5.3] The standard intrinsic module `ISO_FORTRAN_ENV` contains additional named constants as follows.

- The additional scalar integer constants `INT8`, `INT16`, `INT32`, `INT64`, `REAL32`, `REAL64` and `REAL128` supply the kind type parameter values for integer and real kinds with the indicated bit sizes.
- The additional named array constants `CHARACTER_KINDS`, `INTEGER_KINDS`, `LOGICAL_KINDS` and `REAL_KINDS` list the available kind type parameter values for each type (in no particular order).

[6.1] The standard intrinsic module `ISO_FORTRAN_ENV` contains two new functions as follows.

- `COMPILER_VERSION`. This function is pure, has no arguments, and returns a scalar default character string that identifies the version of the compiler that was used to compile the source file. This function may be used in a constant expression, e.g. to initialise a variable or named constant with this information. For example,

  ```
  Module version_info
    Use Iso_Fortran_Env
    Character(Len(Compiler_Version())) :: compiler = Compiler_Version()
  End Module
  Program show_version_info
    Use version_info
    Print *,compiler
  End Program
  ```

  With release 6.1 of the NAG Fortran Compiler, this program will print something like

  ```
  NAG Fortran Compiler Release 6.1(Tozai) Build 6105
  ```

- `COMPILER_OPTIONS`. This function is pure, has no arguments, and returns a scalar default character string that identifies the options supplied to the compiler when the source file was compiled. This function may be used in a constant expression, e.g. to initialise a variable or named constant with this information. For example,

  ```
  Module options_info
    Use Iso_Fortran_Env
    Character(Len(Compiler_Options())) :: compiler = Compiler_Options()
  ```

```
End Module
Program show_options_info
  Use options_info
  Print *,compiler
End Program
```

If compiled with the options *−C=array −C=pointer −O*, this program will print something like

```
-C=array -C=pointer -O
```

8 Input/output extensions [mostly 5.3]

- The `NEWUNIT=` specifier has been added to the `OPEN` statement; this allocates a new unit number that cannot clash with any other logical unit (the unit number will be a special negative value). For example,

```
INTEGER unit
OPEN(FILE='output.log',FORM='FORMATTED',NEWUNIT=unit)
WRITE(unit,*) 'Logfile opened.'
```

  The `NEWUNIT=` specifier can only be used if either the `FILE=` specifier is also used, or if the `STATUS=` specifier is used with the value `'SCRATCH'`.

- Recursive input/output is allowed on separate units. For example, in

```
Write (*,Output_Unit) f(100)
```

  the function `f` is permitted to perform i/o on any unit except `Output_Unit`; for example, if the value 100 is out of range, it would be allowed to produce an error message with

```
Write (*,Error_Unit) 'Error in F:',n,'is out of range'
```

- [6.0] A sub-format can be repeated an indefinite number of times by using an asterisk (`*`) as its repeat count. For example,

```
SUBROUTINE s(x)
  LOGICAL x(:)
  PRINT 1,x
1   FORMAT('x =',*(:,' ',L1))
END SUBROUTINE
```

  will display the entire array `x` on a single line, no matter how many elements `x` has. An indefinite repeat count is only allowed at the top level of the format specification, and must be the last format item.

- [6.0] The `G0` and `G0.`*d* edit descriptors perform generalised editing with all leading and trailing blanks (except those within a character value itself) omitted. For example,

```
PRINT 1,1.25,.True.,"Hi !",123456789
1  FORMAT(*(G0,','))
```

  produces the output

```
1.250000,T,Hi !,123456789,
```

  [7.2] Note that `G0.`*d* was not permitted by Fortran 2008 to be used for output of types Integer, Logical, or Character data, but this limitation has been removed by Fortran 2018.

9 Programs and procedures [mostly 5.3]

- An empty internal subprogram part, module subprogram part or type-bound procedure part is now permitted following a `CONTAINS` statement. In the case of the type-bound procedure part, an ineffectual `PRIVATE` statement may appear following the unnecessary `CONTAINS` statement.
- [6.0] An internal procedure can be passed as an actual argument or assigned to a procedure pointer. When the internal procedure is invoked via the dummy argument or procedure pointer, it can access the local variables of its host procedure. In the case of procedure pointer assignment, the pointer is only valid until the host procedure returns (since the local variables cease to exist at that point).

  For example,

```
SUBROUTINE mysub(coeffs)
  REAL,INTENT(IN) :: coeffs(0:) ! Coefficients of polynomial.
  REAL integral
  integral = integrate(myfunc,0.0,1.0) ! Integrate from 0.0 to 1.0.
  PRINT *,'Integral =',integral
CONTAINS
  REAL FUNCTION myfunc(x) RESULT(y)
    REAL,INTENT(IN) :: x
    INTEGER i
    y = coeffs(UBOUND(coeffs,1))
    DO i=UBOUND(coeffs,1)-1,0,-1
      y = y*x + coeffs(i)
    END DO
  END FUNCTION
END SUBROUTINE
```

- The rules used for generic resolution and for checking that procedures in a generic are unambiguous have been extended. The extra rules are that
  - a dummy procedure is distinguishable from a dummy variable;
  - an `ALLOCATABLE` dummy variable is distinguishable from a `POINTER` dummy variable that does not have `INTENT(IN)`.
- [6.0] A disassociated pointer, or an unallocated allocatable variable, may be passed as an actual argument to an optional nonallocatable nonpointer dummy argument. This is treated as if the actual argument were not present.
- [5.3.1] Impure elemental procedures can be defined using the `IMPURE` keyword. An impure elemental procedure has the restrictions that apply to elementality (e.g. all arguments must be scalar) but does not have any of the "pure" restrictions. This means that an impure elemental procedure may have side effects and can contain input/output and `STOP` statements. For example,

```
Impure Elemental Integer Function checked_addition(a,b) Result(c)
  Integer,Intent(In) :: a,b
  If (a>0 .And. b>0) Then
    If (b>Huge(c)-a) Stop 'Positive Integer Overflow'
  Else If (a<0 .And. b<0) Then
    If ((a+Huge(c))+b<0) Stop 'Negative Integer Overflow'
  End If
  c = a + b
End Function
```

  When an argument is an array, an impure elemental procedure is applied to each element in array element order (unlike a pure elemental procedure, which has no specified order). An impure elemental procedure cannot be referenced in a context that requires a procedure to be pure, e.g. within a `FORALL` construct.

  Impure elemental procedures are probably most useful for debugging (because i/o is allowed) and as final procedures.

- [6.0] If an argument of a pure procedure has the `VALUE` attribute it does not need any `INTENT` attribute. For example,

```
PURE SUBROUTINE s(a,b)
  REAL,INTENT(OUT) :: a
  REAL,VALUE :: b
  a = b
END SUBROUTINE
```

  Note however that the second argument of a defined assignment subroutine, and all arguments of a defined operator function, are still required to have the `INTENT(IN)` attribute even if they have the `VALUE` attribute.

- [5.3.1] The `FUNCTION` or `SUBROUTINE` keyword on the END statement for an internal or module subprogram is now optional (when the subprogram name does not appear). Previously these keywords were only optional for external subprograms.

- `ENTRY` statements are regarded as obsolescent.

- [1.0] A line in the program is no longer prohibited from beginning with a semi-colon.

- [6.2] The name of an external procedure with a binding label is now considered to be a local identifier only, and not a global identifier. That means that code like the following is now standard-conforming:

```
SUBROUTINE sub() BIND(C,NAME='one')
  PRINT *,'one'
END SUBROUTINE
SUBROUTINE sub() BIND(C,NAME='two')
  PRINT *,'two'
END SUBROUTINE
PROGRAM test
  INTERFACE
    SUBROUTINE one() BIND(C)
    END SUBROUTINE
    SUBROUTINE two() BIND(C)
    END SUBROUTINE
  END INTERFACE
  CALL one
  CALL two
END PROGRAM
```

- [6.2] An internal procedure is permitted to have the `BIND(C)` attribute, as long as it does not have a `NAME=` specifier. Such a procedure is interoperable with C, but does not have a binding label (as if it were specified with `NAME=''`).

- [6.2] A dummy argument with the `VALUE` attribute is permitted to be an array, and is permitted to be of type `CHARACTER` with length non-constant and/or not equal to one. (It is still not permitted to have the `ALLOCATABLE` or `POINTER` attributes, and is not permitted to be a coarray.)

  The effect is that a copy is made of the actual argument, and the dummy argument is associated with the copy; any changes to the dummy argument do not affect the actual argument. For example,

```
PROGRAM value_example_2008
  INTEGER :: a(3) = [ 1,2,3 ]
  CALL s('Hello?',a)
  PRINT '(7X,3I6)',a
CONTAINS
  SUBROUTINE s(string,j)
    CHARACTER(*),VALUE :: string
    INTEGER,VALUE :: j(:)
    string(LEN(string):) = '!'
    j = j + 1
    PRINT '(7X,A,3I6)',string,j
  END SUBROUTINE
END PROGRAM
```

will produce the output

```
Hello!     2     3     4
     1     2     3
```

- [7.0] Submodules, together with separate module procedures, provide an additional method of structuring a Fortran program.

  A "separate module procedure" is a procedure whose interface is declared in the module specification part, but whose definition may provided either in the module itself, or in a submodule of that module. The interface of a separate module procedure is declared by using the `MODULE` keyword in the prefix of the interface body. For example,

```
INTERFACE
  MODULE RECURSIVE SUBROUTINE sub(x,y)
    REAL,INTENT(INOUT) :: x,y
  END SUBROUTINE
END INTERFACE
```

  An important aspect of the interface for a separate module procedure is that, unlike any other interface body, it accesses the module by host association without the need for an `IMPORT` statement. For example,

```
INTEGER,PARAMETER :: wp = SELECTED_REAL_KIND(15)
INTERFACE
  MODULE REAL(wp) FUNCTION f(a,b)
    REAL(wp) a,b
  END FUNCTION
END INTERFACE
```

  The eventual definition of the separate module procedure, whether in the module itself or in a submodule, must have exactly the same characteristics, the same names for the dummy arguments, the same name for the result variable (if a function), the same *binding-name* (if it uses `BIND(C)`), and be `RECURSIVE` if and only if the interface is declared so. There are two ways to achieve this:

  1. Define the procedure in the normal way, and get all the characteristics right; the compiler will check that you have done so. Note that the definition must also include the `MODULE` keyword in the prefix, just like the definition. For example,

```
...
CONTAINS
  MODULE REAL(wp) FUNCTION f(a,b)
    REAL(wp)a,b
    f = a**2 - b**3
  END FUNCTION
```

  2. Alternatively, the entire interface may be accessed in the definition without redeclaring everything by using the `MODULE PROCEDURE` statement in this context. For example,

```
...
CONTAINS
  MODULE PROCEDURE sub
    ! Arguments A and B, their characteristics, and that this is a recursive
    ! subroutine, are all taken from the interface declaration.
    IF (a>b) THEN
      CALL sub(b,-ABS(a))
    ELSE
      a = b**2 - a
    END IF
  END PROCEDURE
```

  A submodule has the form (italic square brackets indicate optionality):

```
submodule-stmt
  declaration-part
[ CONTAINS
   module-subprogram-part ]
END [ SUBMODULE [ submodule-name ] ]
```

The initial *submodule-stmt* has the form

```
SUBMODULE ( module-name [ : parent-submodule-name ] ) submodule-name
```

where *module-name* is the name of a module with one or more separate module procedures, *parent-submodule-name* (if present) is the name of another submodule of that module, and *submodule-name* is the name of the submodule being defined. The submodules of a module thus form a tree structure, with successive submodules being able to extend others; however, the name of a submodule is unique within that module. This structure is to facilitate creation of internal infrastructure (types, constants, and procedures) that can be used by multiple submodules, without having to put all the infrastructure inside the module itself.

The submodule being defined accesses its parent module or submodule by host association; for entities from the module, this includes access to `PRIVATE` entities. Any local entity it declares in the *declaration-part* will therefore block access to an entity in the host that has the same name.

The entities (variables, types, procedures) declared by the submodule are local to that submodule, with the sole exception of separate module procedures that are declared in the ancestor module and defined in the submodule. No procedure is allowed to have a binding name, again, except in the case of a separate module procedure, where the binding name must be the same as in the interface.

For example,

```
MODULE mymod
  INTERFACE
    MODULE INTEGER FUNCTION next_number() RESULT(r)
    END FUNCTION
    MODULE SUBROUTINE reset()
    END SUBROUTINE
  END INTERFACE
END MODULE
SUBMODULE (mymod) variables
  INTEGER :: next = 1
END SUBMODULE
SUBMODULE (mymod:variables) functions
CONTAINS
  MODULE PROCEDURE next_number
    r = next
    next = next + 1
  END PROCEDURE
END SUBMODULE
SUBMODULE (mymod:variables) subroutines
CONTAINS
  MODULE SUBROUTINE reset()
    PRINT *,'Resetting'
    next = 1
  END SUBROUTINE
END SUBMODULE
PROGRAM demo
  USE mymod
  PRINT *,'Hello',next_number()
  PRINT *,'Hello again',next_number()
  CALL reset
  PRINT *,'Hello last',next_number()
END PROGRAM
```

Submodule information for use by other submodules is stored by the NAG Fortran Compiler in files named *module*`.`*submodule*`.sub`, in a format similar to that of `.mod` files. The *−nomod* option, which suppresses creation of `.mod` files, also suppresses creation of `.sub` files.

10 References

The Fortran 2008 standard, IS 1539-1:2010(E), is available from ISO as well as from many national standards bodies. A number of books describing the new standard are available; the recommended reference book is "Modern Fortran Explained (Incorporating Fortran 2018)" by Metcalf, Reid & Cohen, Oxford University Press, 2018 (ISBN 978-0-19-881188-6).