ISO/IEC JTC1/SC22/WG5 N2145

The new features of Fortran 2018

John Reid

January 12, 2018

Abstract

The aim of this paper is to summarize the new features of the Fortran standard that is expected to be published in 2018. It was known informally as Fortran 2015 because the choice of features to include was made in 2015, but WG5 has decided to rename it Fortran 2018 to reflect the date of publication. We take as our starting point Fortran 2008 (ISO/IEC 2010).

Two official extensions have been published as Technical Specifications. ISO/IEC (2012) specifies further features for interoperability of Fortran with C and ISO/IEC (2015) specifies additional parallel features. WG5 committed itself to include both of these in Fortran 2018. The introduction of a new IEEE standard for floating-point arithmetic (ISO/IEC/IEEE 2011) necessitated changes to the IEEE modules. Beyond these features, WG5 decided to limit changes to the removal of deficiencies and discrepancies, which by definition are all small.

For an informal description of Fortran 2008, see Metcalf, Reid and Cohen (2011).

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 five times (1978, 1991, 1997, 2004, 2010). The revisions alternated between being minor (1978, 1997, and 2010) and major (1991 and 2004). Features for further interoperability have been defined in a Technical Specification (ISO/IEC TS 29113:2012) and for further coarray features in another Technical Specification (ISO/IEC TS 18508:2015).

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.

2 Further interoperability of Fortran with C

2.1 C descriptors

Fortran 2008 provides for interoperability of procedures with nonoptional dummy arguments that are scalars, explicit-shape arrays, or assumed-size arrays, but not with dummy arguments that are assumed shape, assumed character length, allocatable, or pointers. This deficiency is circumvented by the Fortran processor passing to the C function the address of a standardized descriptor, known as a ***C descriptor***, of the object instead of the object itself. This allows the C function to work with the object because it has access to all its properties. It is not permitted to use this knowledge to access memory outwith the object via a pointer based on the base address of the object. The source file `ISO_Fortran_binding.h`, whose contents will vary between vendors, provides definitions and prototypes that enable a C function to do this.

A C descriptor is a C structure of the type `CFI_cdesc_t`. This type has members that are required to include the following:

`void * base_addr` C address of the object if it is scalar or of the first array element if it is an array. If the object is an unallocated allocatable variable or a pointer that is disassociated, the value is a null pointer. If the object has zero size, the value is not a null pointer but is otherwise processor dependent.

`size_t elem_len` The storage size in bytes of the object if it is scalar or of an array element if it is an array.

`int version` The version number of the file `ISO_Fortran_binding.h` in use when the descriptor was established. It is equal to the value of `CFI_VERSION` in the file.

`CFI_rank_t rank` Rank of the object if it is an array or 0 if it is scalar. `CFI_rank_t` is a typedef name for a standard signed integer type capable of representing the largest supported rank.

`CFI_type_t type` Code for the type of the object. `CFI_type_t` is a typedef name for a standard signed integer type capable of representing the values for the supported type codes. For details, see Section 2.4.

`CFI_attribute_t attribute` Code to indicate whether the object is allocatable, a pointer, or neither. For details, see Section 2.2.

`CFI_dim_t dim` If the object is an array, this is an array of size its rank and holding its lower bounds, extents, and strides. For details, see Section 2.3.

The first three members are always at the beginning and ordered as in this list. The `dim` member is always at the end. The remainder, including any vendor-specific additional members, may be in any order.

2.2 Attribute codes

The macros in Table 1 provide attribute codes. They are nonnegative and have distinct integer values.

`CFI_attribute_t` is a typedef name for a standard signed integer type capable of representing the values for the supported attribute codes.

Table 1: Macros for attribute codes

Macro name	Attribute
`CFI_attribute_pointer`	data pointer
`CFI_attribute_allocatable`	allocatable
`CFI_attribute_other`	nonallocatable nonpointer

2.3 The type `CFI_dim_t`

The type `CFI_dim_t` has members that are required to include the following:

`CFI_index_t lower_bound` Lower bound of a dimension of an array. For an array pointer or allocatable array, the value is determined by argument association, allocation, or pointer association. For a nonallocatable nonpointer array, the value is zero.

`CFI_index_t extent` Number of elements in the dimension or -1 for the final dimension of an assumed-size array.

`CFI_index_t sm` Stride in memory, that is, the difference in bytes between the addresses of successive elements in the dimension.

where `CFI_index_t` is a typedef name for a standard signed integer type capable of representing a memory address difference in bytes.

2.4 Type codes

The macros in Table 2 provide integer type codes. The value for `CFI_type_other` is negative and distinct from the others. `CFI_type_struct` specifies a C structure that is interoperable with a Fortran derived type; its value is positive and distinct from all other type codes. The value for a C type that is not interoperable with a Fortran type and kind supported by the Fortran processor is negative. Otherwise, the value for an intrinsic type is positive.

2.5 Other constants

The macro `CFI_MAX_RANK` provides an integer constant whose value is the largest rank supported, which must be at least 15.

The macro `CFI_VERSION` provides an integer constant whose value encodes the version of the header `ISO_Fortran_binding.h` in use.

Table 2: Macros for type codes

Macro name	C type
`CFI_type_signed_char`	`signed char`
`CFI_type_short`	`short int`
`CFI_type_int`	`int`
`CFI_type_long`	`long int`
`CFI_type_long_long`	`long long int`
`CFI_type_size_t`	`size_t`
`CFI_type_int8_t`	`int8_t`
`CFI_type_int16_t`	`int16_t`
`CFI_type_int32_t`	`int32_t`
`CFI_type_int64_t`	`int64_t`
`CFI_type_int_least8_t`	`int_least8_t`
`CFI_type_int_least16_t`	`int_least16_t`
`CFI_type_int_least32_t`	`int_least32_t`
`CFI_type_int_least64_t`	`int_least64_t`
`CFI_type_int_fast8_t`	`int_fast8_t`
`CFI_type_int_fast16_t`	`int_fast16_t`
`CFI_type_int_fast32_t`	`int_fast32_t`
`CFI_type_int_fast64_t`	`int_fast64_t`
`CFI_type_intmax_t`	`intmax_t`
`CFI_type_intptr_t`	`intptr_t`
`CFI_type_ptrdiff_t`	`ptrdiff_t`
`CFI_type_float`	`float`
`CFI_type_double`	`double`
`CFI_type_long_double`	`long double`
`CFI_type_float_Complex`	`float _Complex`
`CFI_type_double_Complex`	`double _Complex`
`CFI_type_long_double_Complex`	`long double _Complex`
`CFI_type_Bool`	`_Bool`
`CFI_type_char`	`char`
`CFI_type_cptr`	`void *`
`CFI_type_struct`	interoperable C structure
`CFI_type_other`	Not otherwise specified

2.6 Memory for a C descriptor

`CFI_CDESC_T` is a function-like macro that takes one argument and evaluates to an unqualified type of suitable size and alignment for a variable holding a C descriptor. The argument is an integer constant expression specifying the rank of the object to be described. A pointer to a variable declared using `CFI_CDESC_T` can be cast to `CFI_cdesc_t *`, for example

```
    CFI_CDESC_T(5) object;
    CFI_cdesc_t * dv = (CFI_cdesc_t *)&object;
    int ind;
    ind = CFI_establish(dv, ...);
   /* CFI_establish explained in Section 2.7.2. */
```

2.7 C functions declared in ISO_Fortran_binding.h

2.7.1 Introduction

In this section, we give details of functions that are provided for the C programmer to

- Establish a C descriptor for a nonallocatable nonpointer data object of known shape, an unallocated allocatable object, or a data pointer.
- Allocate or deallocate an allocatable or pointer object using the Fortran mechanism.
- Update a C descriptor to describe an array section.
- Compute a C address as if by Fortran subscripting.
- Test an array for contiguity.

A C descriptor must not be initialized, updated, or copied except by calling one of these functions.

If the address of a C descriptor is a formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument,

- the C descriptor must not be modified if either the corresponding dummy argument in the Fortran interface has intent `in` or the C descriptor is for a nonallocatable nonpointer object, and
- the `base_addr` member of the C descriptor must not be accessed before it is given a value if the corresponding dummy argument in the Fortran interface is a pointer and has intent `out`.

If the address of a C descriptor is a C actual argument that corresponds to an assumed-shape Fortran dummy argument, that descriptor shall not be for an assumed-size array.

Most of the functions return an integer error code. The meanings of these codes are tabulated in Section 2.7.7.

2.7.2 Establishing a C descriptor

```
int CFI_establish(CFI_cdesc_t *dv, void *base_addr,
          CFI_attribute_t attribute, CFI_type_t type, size_t elem_len,
          CFI_rank_t rank, const CFI_index_t extents[]);
```

updates the object with the address `dv` to be

- an established C descriptor for a nonallocatable nonpointer data object that is a scalar or a contiguous array, an unallocated allocatable object, or a data pointer; or
- a C descriptor with the attribute `CFI_attribute_other` that is suitable for argument `result` of one of the functions of Section 2.7.4 when it is desired to construct a C descriptor for a nonallocatable nonpointer array section.

The function return value is an error indicator.

`dv` is the address of a data object that is large enough to hold a C descriptor of the rank specified by `rank`. It must not be the address of a C descriptor that is a formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument. It must not be the address of a C descriptor for an allocated allocatable object. If an error is detected, the data object is not modified.

`base_addr` is a null pointer or the address of a contiguous storage sequence that is appropriately aligned for an object of the type specified by argument `type`. If it is the address of a Fortran data object, the `type` and `elem_len` arguments must be consistent with the type and type parameters of the Fortran data object. If it is a null pointer, the aim is to establish a C descriptor for an unallocated allocatable object, a disassociated pointer, or a C descriptor that has the attribute `CFI_attribute_other` and can later be associated with a data object.

`attribute` must be one of the attribute codes in Table 1. If it is `CFI_attribute_allocatable`, `base_addr` must be a null pointer.

`type` must be one of the type codes in Table 2.

`element_len` is ignored unless `type` is equal to `CFI_type_struct`, `CFI_type_other`, or a Fortran character type code, in which case `element_len` must be greater than zero and equal to the storage size in bytes of an element of the object.

`rank` specifies the desired rank.

`extents` is ignored if `rank` is zero or `base_addr` is a null pointer. Otherwise, it is the address of an array specifying the desired extents. For a pointer, any lower bounds are set to zero.

```
int CFI_setpointer(CFI_cdesc_t *result, CFI_cdesc_t *source,
                   const CFI_index_t lower_bounds[]);
```

updates a C descriptor for a Fortran pointer to be associated with the whole of a given object or to be disassociated. The function return value is an error indicator.

`result` is the address of a C descriptor for a Fortran pointer. If `source` is a null pointer or the address of a C descriptor for a disassociated pointer, the C descriptor is updated to describe a disassociated pointer. Otherwise, the `base_addr` member is updated to that of the C descriptor whose address is `source` and, unless the `lower_bounds` argument is a null pointer, its lower bounds are replaced by the values of the first `source->rank` elements of `lower_bounds`.

`source` is a null pointer or the address of a C descriptor for an allocated allocatable object, a data pointer object, or a nonallocatable nonpointer data object that is not an assumed-size array. If `source` is not a null pointer, the values of its `elem_len`, `rank`, and `type` members must be the same as in the C descriptor with the address `result`.

`lower_bounds` Unless `source` is a null pointer or `source->rank` is zero, `lower_bounds` is the address of an array with at least `source->rank` elements.

2.7.3 Fortran allocation and deallocation

Within a C function, an allocatable object may be allocated or deallocated only by execution of the two functions of this section. A Fortran pointer can become associated with a target by execution of the `CFI_allocate` function.

```
int CFI_allocate(CFI_cdesc_t *dv, const CFI_index_t lower_bounds[],
                 const CFI_index_t upper_bounds[], size_t elem_len);
```

allocates memory for the object described by a C descriptor using the Fortran `allocate` mechanism. The function return value is an error indicator.

`dv` is the address of a C descriptor for the object. The `base_addr` member of the C descriptor must be a null pointer. If the type is not a character type, the `elem_len` member specifies the element length. The attribute member must have the value of `CFI_attribute_allocatable` or `CFI_attribute_pointer`. If an error is detected, the C descriptor is not modified.

`lower_bounds` is the address of an array holding the desired lower bounds.

`upper_bounds` is the address of an array holding the desired upper bounds.

`elem_len` is the desired storage size in bytes of an element of the object if the type specified in the C descriptor is a Fortran character type; otherwise, `elem_len` is ignored.

```
int CFI_deallocate(CFI_cdesc_t *dv);
```

deallocates memory for the object described by a C descriptor using the Fortran `deallocate` mechanism. The function return value is an error indicator.

`dv` is the address of a C descriptor for the object. It must have been allocated using the Fortran `allocate` mechanism. If the object is a pointer, it must be associated with a target satisfying the conditions for successful deallocation by the Fortran `deallocate` statement. If an error is detected, the C descriptor is not modified.

2.7.4 Array sections

```
int CFI_section(CFI_cdesc_t *result, const CFI_cdesc_t *source,
         const CFI_index_t lower_bounds[],
         const CFI_index_t upper_bounds[], const CFI_index_t strides[]);
```

constructs a C descriptor for an array section of a given array. The section is specified by the arguments `lower_bounds`, `upper_bounds`, and `strides` just as by the section notation in Fortran except that a stride value may be zero, in which case the lower bound (or its default) specifies a subscript rather than a section subscript and the upper bound (or its default) must have the same value. The function return value is an error indicator.

`result` is the address of a C descriptor with `attribute` member having value `CFI_attribute_other` or `CFI_attribute_pointer`. This must be `CFI_attribute_pointer` if the address is of a C descriptor that is a formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument. Successful execution updates the `base_addr` and `dim` members of the C descriptor to describe the array section of the array described by `source` that is determined by `lower_bounds`, `upper_bounds`, and `strides`. If an error is detected, the C descriptor is not modified.

`source` is the address of a C descriptor that describes a nonallocatable nonpointer array, an allocated allocatable array, or an associated array pointer. The `elem_len` and `type` members shall have the same values as the corresponding members of `result`.

`lower_bounds` is the address of an array with at least `source->rank` elements that specifies the lower bounds of the section or is a null pointer specifying default lower bounds (those of the `source` C descriptor).

`upper_bounds` is the address of an array with at least `source->rank` elements that specifies the upper bounds of the section or is a null pointer specifying default upper bounds (those of the `source` C descriptor).

`strides` is the address of an array with at least `source->rank` elements that specifies the strides of the section or is a null pointer specifying default strides of 1.

```
int CFI_select_part(CFI_cdesc_t *result, const CFI_cdesc_t *source,
                    size_t displacement, size_t elem_len);
```

constructs a C descriptor for an array section for which each element is a part of the corresponding element of a given array. The part has type `result->type` and is specified by `displacement`. It must be a component of a structure, a substring, or the real or imaginary part of a complex value. The function return value is an error indicator.

`result` is the address of a C descriptor with `attribute` member having value `CFI_attribute_other` or `CFI_attribute_pointer`. This must be `CFI_attribute_pointer` if the address is of a C descriptor that is a formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument. Successful execution updates the `base_addr`, `dim`, and `elem_len` members of the C descriptor to describe the array section of the array described by `source` that is determined by `result->type` and `displacement`. If an error is detected, the C descriptor is not modified.

`source` is the address of a C descriptor that describes an allocated allocatable array, an associated array pointer or a nonallocatable nonpointer array that is not an assumed-size array.

`displacement` specifies the displacement in bytes of an element of the section from the corresponding element of the array and must be appropriately aligned for an object of the type `result->type`.

`elem_len` has a value equal to the storage size in bytes of an element of the array section if `result->type` specifies a Fortran character type; otherwise, `elem_len` is ignored.

2.7.5 Fortran subscripting

```
void *CFI_address(const CFI_cdesc_t *dv, const CFI_index_t subscripts[]);
```

returns the C address of a scalar or of an element of an array using Fortran subscripting.

`dv` is the address of a C descriptor for the object. The object must not be an unallocated allocatable variable or a pointer that is not associated.

`subscripts` is a null pointer or the address of an array. If the object is an array, `subscripts` must be the address of an array with at least as many elements as the rank of the object. The value of `subscripts[i]` must be within the bounds of dimension `i` as specified by the `dim` member of the C descriptor, except for the final dimension of an assumed-size array. For an assumed-size array, the subscript order value specified by the subscripts must not exceed the size of the array.

2.7.6 Testing for contiguity

```
int CFI_is_contiguous(const CFI_cdesc_t * dv);
```

returns 1 if the array described by `dv` is contiguous, and 0 otherwise.

`dv` is the address of a C descriptor for an array. The `base_addr` member of the C descriptor must not be a null pointer.

2.7.7 Error codes

The macros in Table 3 are used as error codes. The macro `CFI_SUCCESS` is the integer constant 0. The value of each other macro is nonzero and different from the others in this table. Error conditions other than those listed are indicated by error codes different from the values in this table.

Table 3: Macros for error codes

Macro name	Meaning
`CFI_SUCCESS`	No error detected.
`CFI_ERROR_BASE_ADDR_NULL`	The base address member of a C descriptor is a null pointer in a context that requires a non-null pointer value.
`CFI_ERROR_BASE_ADDR_NOT_NULL`	The base address member of a C descriptor is not a null pointer in a context that requires a null pointer value.
`CFI_INVALID_ELEM_LEN`	The value supplied for the element length member of a C descriptor is not valid.
`CFI_INVALID_RANK`	The value supplied for the rank member of a C descriptor is not valid.
`CFI_INVALID_ATTRIBUTE`	The value supplied for the attribute member of a C descriptor is not valid.
`CFI_INVALID_EXTENT`	The value supplied for the extent member of a C descriptor is not valid.
`CFI_INVALID_DESCRIPTOR`	A C descriptor is invalid in some way.
`CFI_ERROR_MEM_ALLOCATION`	Memory allocation failed.
`CFI_ERROR_OUT_OF_BOUNDS`	A reference is out of bounds.

2.8 Interoperability of procedures

It would be a severe burden to implementors to provide `CFI_allocate`, see 2.7.3, for an object of a derived type with default initialization, so an allocatable or pointer dummy argument of an interoperable procedure are not permitted to be of such a type.

In Fortran 2018, a Fortran dummy argument without the `value` attribute may correspond to a formal parameter of the C prototype that is of a pointer type if

1. the dummy argument is a nonallocatable nonpointer character variable with assumed character length (specified by aa asterisk) and the formal parameter is a pointer to `CFI_cdesc_t`,
2. the dummy argument is allocatable, assumed-shape, assumed-rank (Section 2.12), or a pointer without the `contiguous` attribute, and the formal parameter is a pointer to `CFI_cdesc_t`, or
3. the dummy argument is assumed-type (Section 2.13) and not allocatable, assumed-shape, assumed-rank, or a pointer, and the formal parameter is a pointer to void,

Any allocatable or pointer dummy argument of type `character` must have deferred character length (specified by a colon).

In a reference from C to a Fortran procedure with an interoperable interface, a C actual argument must be the address of a C descriptor for the intended effective argument if the corresponding dummy argument interoperates with a C formal parameter that is a pointer to `CFI_cdesc_t`. In this C descriptor, the members other than `attribute` and `type` must describe an object with the characteristics of the intended effective argument. The value of the `attribute` member must be compatible with the characteristics of the dummy argument. The `type` member must have a value that depends on the intended effective argument as follows:

- if the dynamic type of the intended effective argument is an interoperable type listed in Table 2, the corresponding value for that type;
- if the dynamic type of the intended effective argument is an intrinsic type for which the processor defines a nonnegative type specifier value not listed in Table 2, that type specifier value;
- otherwise, `CFI_type_other`.

2.9 Lifetimes

When a Fortran object is deallocated, execution of its host instance is completed, or its association status becomes undefined, all C descriptors and C pointers to any part of it become undefined, and no further use of them may be made.

A C descriptor whose address is a formal parameter that corresponds to a Fortran dummy argument becomes undefined on return from a call to the function from Fortran. If the dummy argument does not have either the `target` or `asynchronous` attribute, all C pointers to any part of the object become undefined on return from the call, and no further use of them may be made.

2.10 Interoperability with the C type `ptrdiff_t`

The named integer constant `c_ptrdiff_t` has been added to the `iso_c_binding` module for use as a kind parameter to allow interoperability with the C type `ptrdiff_t`.

2.11 Changes to procedures in the `iso_c_binding` module

The argument to `c_loc` may be a noninteroperable array.

The `cptr` argument to `c_f_pointer` may be the C address of a storage sequence that is not in use by any other Fortran entity. In this case, the `fptr` argument becomes associated with that storage sequence.

The `fptr` argument to `c_f_pointer` may become pointer associated with a noninteroperable array.

The argument to `c_funloc` may be a noninteroperable procedure.

The `fptr` argument to `c_f_procpointer` may become pointer associated with a noninteroperable procedure pointer.

2.12 Assumed rank

2.12.1 Assumed-rank objects

The concept of ***assumed rank*** has been added to facilitate interoperating with C functions that have been written for arguments of any rank. An addition to the features of TS 29113 is the `select rank` construct, which allows a Fortran procedure to process arguments of variable rank.

A dummy argument that is not a coarray and does not have the `value` attribute may be declared of assumed rank with the syntax `(..)`. For example, the procedure

```
subroutine scale(a) bind(c)
  real a(..)
     :
end subroutine scale
```

may be provided with an array of any rank or even a scalar as an actual argument. For a call from Fortran to a C function with such a dummy argument in its interface, the Fortran processor will construct a C descriptor for the actual argument and this will allow the C function to discover the rank and act accordingly.

The interface of a procedure with an assumed-rank dummy argument is required to be explicit.

To allow a Fortran procedure to determine the rank, a new intrinsic inquiry function has been added:

`rank(a)`

> `a` is a scalar of array of any type.

The result is a default integer scalar whose value is the rank of `a`.

An assumed-rank dummy argument is allowed to be passed to another procedure as assumed rank or appear as the first argument of `c_loc`, `c_sizeof`, or an intrinsic inquiry function. For `c_sizeof`, it must not be associated with an assumed-size array (for the reason explained in Section 2.12.3).

An assumed-rank array that is associated with a scalar is regarded as having rank 0 and size 1. An assumed-rank array that is associated with an array has the rank and extents of the actual argument.

The concept of type-kind-rank (TKR) compatibility for dummy arguments has been extended to allow for assumed-rank dummy arguments. They are compatible with any rank including assumed-rank.

2.12.2 The `select rank` construct

To execute alternative code depending on the actual rank of an assumed-rank object, the `select rank` construct is provided. It takes the form

```
[name:] select rank ( [ associate-name => ] selector )
           [ select-rank-case-stmt [name]
                   block ]...
           end select [ name ]
```

where *selector* is the name of an assumed-rank object and each *select-rank-case-stmt* is one of

```
    rank (scalar-int-constant-expr)
    rank (*)
    rank default
```

A `select rank` construct selects at most one block to be executed. If an *associate-name* is provided, this is used for the assumed-rank object within the construct.

A `rank (`*scalar-int-constant-expr*`)` statement selects an array that is not assumed-size and has the given rank. Within the block, it is treated as if it had been declared with this rank. Its bounds are those that are obtained by the intrinsics `lbound` and `ubound` for the corresponding actual argument.

A `rank (*)` statement selects an assumed-size array. Within the block, it is treated as if it had been declared `dimension (*)`.

A `rank default` statement selects an object that is not otherwise matched. It is assumed-rank and has the same properties as the selector.

As with other constructs, the `select rank` construct can be named; either the same name appears on the `select rank` and `end rank` statements or no name appears on either; and if a name appears on a *select-rank-case-stmt*, the same name must appear on the `select rank` statement.

2.12.3 Assumed-size arrays

An assumed-size array (from Fortran 77) is a dummy argument array whose size is assumed from that of its effective argument. Unfortunately, this means that if the rank is greater than one, the final extent need not be well defined. For example, if the dummy array declared thus

```
real a(3,*)
```

is associated with an array of size 10, `a(1,4)` is valid but `a(2,4)` is not.

A further difficulty is that many processors pass only the address of the first element of the effective argument and do not pass its size and this convention has to be respected for communication with C. Therefore, an assumed-size array has to be treated as an unknown-size array. This lies behind many of the decisions made in connection with assumed-size arrays and interfacing with C.

Because an assumed-size array does not have a well-defined final extent, an assumed-rank array that is associated with an assumed-size array is regarded as having a final extent of -1. This is the value returned as `size(array,dim=rank(array))` and as the final element of `shape(array)`. The result of `size(array)` is the negative value `product(shape(array))`. The value for the final upper bound of an array returned by `ubound` is 2 less than the final lower bound for the array.

Because the size of an assumed-size array is likely to be unknown, if an assumed-size array is an actual argument that corresponds to a dummy argument that is an intent `out` assumed-rank array, it must not be polymorphic, finalizable, of a type with an allocatable ultimate component, or of a type for which default initialization is specified. Because a nonallocatable nonpointer assumed-rank array might be associated with an assumed-size array, the same restriction applies to this as an actual argument.

2.13 Assumed type

The concept of ***assumed type*** has been added to allow a C function to accept an argument of any type except a derived type that has type parameters, type-bound procedures, or final subroutines.

An assumed-type object is declared with the syntax `type(*)`[1]. It is unlimited polymorphic. It is required to be a dummy argument so that during execution it is always associated with an object that has a type. It is not itself considered to have the same declared type as another entity, including another unlimited polymorphic entity.

[1]or with an `implicit type(*)` statement.

The interface of a procedure with an assumed-type dummy argument is required to be explicit.

There are two possibilities for the type of the corresponding C formal parameter

1. pointer to `void`, i.e. `void *`
2. pointer to a C descriptor

The first case provides support for an actual argument that is of any type and is either a scalar or an assumed-size array. When a C function is invoked, the type is unknown, just as the size is for an assumed-size array (see Section 2.12.3). Usually, at least the size of a scalar of the type is needed; in this case, the programmer must make it available in some other way, for example, through another argument.

The second case provides support for an actual argument that is of any type and is an assumed-shape or assumed-rank array. When a C function is invoked, the type may be determined from the C descriptor, but may not be altered.

A less restricted facility would have been possible, but these were the main requests and it was decided that is was important to keep the feature simple.

These considerations led to the restrictions that an object of assumed type is permitted to be neither allocatable, a coarray, a pointer, nor an explicit-shape array. Furthermore, it is not permitted to have intent `out` or the `value` attribute. If it is not of assumed shape or assumed rank, it is passed to a C function as a pointer to `void`; otherwise, it is passed as a pointer to a C descriptor. To avoid an assumed-shape array ever being passed as a pointer to `void`, an assumed-type actual argument that corresponds to an assumed-rank dummy argument is required to be assumed-shape or assumed-rank.

An object of assumed type is severely limited within Fortran. It is not permitted to appear in a designator or expression except as an actual argument corresponding to a dummy argument that is of assumed-type, or as the first argument of one of the functions `is_contiguous`, `lbound`, `present`, `rank`, `shape`, `size`, `ubound`, and `c_loc`.

2.14 Allocatable dummy arguments of intent `out`

In Fortran 2008, an allocatable dummy argument of intent `out` is deallocated on entry to the procedure if the actual argument is allocated. This still applies if the procedure is called from C and the dummy argument is the address of a C descriptor for an allocated allocatable variable. If a C function with an interface including an allocatable dummy argument is called from Fortran with the variable allocated, it is deallocated by the Fortran processor before entry to the procedure. Thus, in both cases, the Fortran processor performs the deallocation when it is appropriate.

2.15 `Contiguous` attribute

An assumed-rank array may have the `contiguous` attribute. During execution, the corresponding effective argument must be contiguous.

When an interoperable Fortran procedure with a simply contiguous dummy argument is invoked from C and the actual argument is the address of a C descriptor for a discontiguous object, the Fortran processor is required handle the difference in contiguity.

When an interoperable C procedure whose Fortran interface has a simply contiguous dummy argument is invoked from Fortran and the effective argument is discontiguous, the Fortran processor ensures that the C procedure receives a descriptor for a contiguous object.

When an interoperable C procedure whose Fortran interface has a simply contiguous dummy argument is invoked from C, and the actual argument is the address of a C descriptor for a discontiguous object, the C code within the procedure must be prepared to handle the discontiguous argument.

2.16 Optional arguments

Fortran 2008 does not provide for interoperability of procedures with optional arguments. This is addressed in Fortran 2018 for optional arguments that do not have the `value` attribute.

If an interoperable procedure defined by means other than Fortran has an optional dummy argument, and the corresponding actual argument in a reference from Fortran is absent, the procedure is invoked with a null pointer for that argument.

If an interoperable procedure defined by means of Fortran is invoked by a C function, an optional dummy argument is absent if and only if the corresponding argument in the invocation is a null pointer.

2.17 Asynchronous communication

The `asynchronous` attribute has been extended from I/O to apply to communication performed by means other than Fortran. The main application is for nonblocking calls of `MPI_Irecv` and `MPI_Isend`. A call of `MPI_Irecv` is very like asynchronous input – data is put in the buffer array while execution continues. And a call of `MPI_Isend` is very like asynchronous output - data is copied from the buffer array while execution continues. For both, a call of `MPI_Wait` plays the role of `wait` for asynchronous I/O.

The standard does not limit asynchronous communication to these MPI functions. Instead, it talks in general of procedures that initiate input or output asynchronous communication or complete it. Whether a procedure has such a property is processor dependent.

The rules for input and output asynchronous communication are exactly the same as those for asynchronous input and output, respectively.

3 Additional parallel features in Fortran

3.1 Teams

Teams have been introduced to allow separate sets of images to execute independently. An important design objective was that, given a code that had been developed and tested on all images, it should be possible to run the code on a team without making changes. This requires that if a team has n images, the image indices within the team run from 1 to n.

It was decided that teams should always be formed by partitioning an existing team into parts, starting with the team of all the images, which is known as the ***initial team***. The team in which a statement is executed by an image is known as the ***current team***.

Information about a team is held collectively on all the images of the team in a scalar variable of type `team_type` from the intrinsic module `iso_fortran_env`. The components of this type are private and it is expected that an implementation will use them to hold information about the team that will enable efficient communication to take place. Because doing this efficiently in terms of both memory and execution time requires the data to differ from image to image, it is inappropriate to copy the data between images. Therefore, the following are not permitted: a coarray variable of type `team_type`, a coarray component of type `team_type`, a coindexed object of type `team_type`, and allocating a polymorphic coarray to be of type `team_type` or of a type with a subcomponent of type `team_type`. A variable of type `team_type` becomes undefined when an intrinsic assignment is executed if it would otherwise be given a value from another image.

A set of new teams is formed by executing a `form team` statement on all the images of the current team. To which new team an image of the current team belongs is determined by its ***team number***, which is a positive integer. All the images with the same team number belong to the same new team. The `form team` statement is supplied with an integer expression holding the value of the team number. For example, the code

```
use iso_fortran_env
type ( team_type ) new_team
:
form team ( 1+mod(this_image(),2), new_team )
```

forms two new teams consisting of the images of the current team that have odd or even image indices. We describe the `form team` statement in detail in Section 3.3.

Changes of team take place at the `change team` and `end team` statements, which mark the beginning and end of a new construct, the `change team` construct:

```
change team ( new_team )
    : ! Statements executed with new_team as the current team
end team
```

The values of `new_team` must have been established by the prior execution of a `form team` statement on all the images of the current team. Both the `change team` and the `end team` statement are image control statements. The executing image and the other images of the new team synchronize at these statements. These images must all execute the same `change team` statement. While we expect it to be usual for the images of the other new teams to execute the construct, this is not required – they might continue to execute in the previously current team. We describe the `change team` construct in detail in Section 3.4.

The team that is current during the execution of a `form team` statement is known as the ***parent*** of each new team formed. We find it convenient to refer to each new team as a ***child*** of the current team. If the current team is not the initial team, the children have as parent a team that itself has a parent. Parents of parents can occur at any depth and are known as ***ancestors***. Note that `change team` constructs can be nested to any depth and can be executed in a procedure called from within a `change team` construct.

3.2 Image failure

It is anticipated that coarray programs may execute on huge numbers of images. While the likelihood of a particular image failing during the execution of a program is small, the likelihood that one of them might fail is significant when there are a huge number of them. Therefore, the concept of continued execution in the presence of failed images has been introduced. It is not required that the system support this. The constant `stat_failed_image` has been added to the module `iso_fortran_env`. This is positive if this support is provided and negative otherwise. If it is positive, it is used for the value of a `stat=` specifier or `stat` argument if a failed image is involved in either an image control statement, a reference to a coindexed object (see Section 3.10), or an invocation of a collective (Section 3.19) or atomic (Section 3.20) subroutine, and no other error condition occurs.

It is not expected that the system will automatically produce correct results in the presence of failed images. Instead, there are features in the language to permit a programmer to design a recovery process. For example, it may be possible to go back to a previous state of the computation and repeat it with fewer images or with 'reserve' images brought in to replace failed ones. There are likely to be only a few key points in the computation from which recovery is practical, so it is important that the working images all reach such a point, albeit with incorrect results. For example, the `change team` construct does not fail in the presence of failed images – instead, it executes on all the remaining images of the team. In Fortran 2018, `stat=` specifiers (see Section 3.21) have been added to allow such continued execution. The term ***active*** has been introduced for an image that has neither failed nor stopped.

3.3 `Form team` statement

The `form team` statement takes the general form

`form team (` *team-number*`,` *team-variable*[`,` *form-team-list*] `)`

where *team-number* is a scalar integer expression whose value must be positive and *team-variable* is a scalar variable of type `team type` whose values on all the images of a child team will be defined with information about that child team. All the images of the current team that have the same *team-number* value will be in the same child team. This value is the ***team number*** for the child team and is used to identify it in statements executed in an image of another child of the same parent. The team numbers of all the images of the initial team are always -1.

By default, the processor chooses which image indices are assigned to which images of each child team, and the choice may vary from processor to processor. However, they may be specified by including

new_index=*scalar-int-expr*

in the *form-team-list* to give the image index for the executing image in the child team. If a child team has k images, the values on those images must be a permutation of $1, 2, \ldots k$. The *form-team-list* may also contain stat= and errmsg= specifiers as, for example, for allocate and deallocate.

The form team statement is an image control statement. The same statement must be executed by all active images of the current team and they synchronize.

3.4 Change team **construct**

The change team statement takes the general form

[construct-name:*]* change team (*team-value[*, *association-list][*, *stat-list]*)

where *team-value* is a scalar of type team_type. Its values on all active images of the current team must be as constructed by the execution of a form team statement on all those images (see Section 3.1). If it is a variable, its value must not be altered during the execution of the change team construct. Each *association* declares a new name and new cobounds for an ***associating*** coarray that is in scope at the change team statement. For example, if big has cobounds [1:k,1:k] and the current team is subdivided into k teams of k images, the association

```
part[*] => big
```

makes part an associating coarray with cobounds [1:k] on each new team. The appearance of an association does not prevent the coarray being referenced by its original name and with its original cosubscripts and cobounds. The mappping to an image index is unchanged but the image index now refers to an image of the new team. The intrinsic ucobound returns the same final upper cobound now as it would have in the original team, despite this probably now being out of range.

The *stat-list* returns information about the success of the execution. There may be at most one stat= specifier and at most one errmsg= specifier.

The end team statement takes the general form

end team *[*(*[stat-list]*)*] [construct-name]*

The reason for the appearance of *stat-list* here is to detect the possibility of image failure in the current team. The end team statement is an image control statement and the images of the team that was current inside the construct synchronize here.

3.5 Coarrays allocated in teams

In Fortran 2008, coarrays are always allocated and deallocated in synchrony across all images, which allows each image to calculate the address of a coarray element on another image. This is sometimes called ***symmetric memory***. In Fortran 2018, synchronization is now across the team, of course. Symmetric memory is maintained within teams by requiring that

1. any allocatable coarray that is allocated before entry to a `change team` construct remains allocated during the execution of the construct and

2. any allocatable coarray that becomes allocated within a `change team` construct and is still allocated when the construct is left is automatically deallocated, even if it has the `save` attribute.

This allows each image to hold its allocatable coarrays in a stack with those allocated in the initial team at the bottom, those allocated in the team that is a child of the initial team next, those allocated in the team that is a child of the child team next, etc. Of course, there is no requirement for exactly this form of memory management to be used.

3.6 `Critical` construct

The `critical` statement now takes the general form

[construct-name`:`*]* `critical` *[*`(` *[stat-list]* `)`*]*

with optional `stat=` and `errmsg=` specifiers in its *stat-list* to detect the case of an image failing while executing a `critical` construct. If `stat=` is present in this case, the construct is treated as having completed execution so that another image can commence executing it. When this other image commences execution of the construct, the `stat=` variable will have the value `stat_failed_image` to indicate the failure of the previous execution of the construct. The `errmsg=` specifier provides a message in the event of failure.

3.7 `Lock` and `unlock` statements

Failure of an image causes all lock variables that are locked by that image to become unlocked.

If a `stat=` specifier is present in a `lock` statement and the image of the lock variable has failed, the stat variable is given the value `stat_failed_image`. Otherwise, if the lock variable is unlocked because of the failure of the image that locked it, the stat variable is given the value `stat_unlocked_failed_image` from the module `iso_fortran_env`. If a `stat=` specifier is present in an `unlock` statement and the image of the lock variable has failed, the stat variable is given the value `stat_failed_image`.

3.8 Events

Events have been introduced to allow an action to be delayed until one or more actions have been performed on other images. An image records that it has performed an action by executing an `event post` statement. The record is held in a scalar coarray of type `event_type` from the intrinsic module `iso_fortran_env` and known as an ***event variable***. An image executes an `event wait` statement if it needs to delay its actions for those of other images. Each `event post` execution has a matching `event wait` execution that involves the same event variable. The segment that precedes the `event post` execution precedes the segment that succeeds the matching `event wait` execution.

The value of an event variable includes its event count, which is of type `integer (atomic_int_kind)` and intially has the value 0. It records the number of `event post` executions for the event variable that are currently unmatched. The count of the specified event variable is atomically incremented by 1 when an `event post` statement is executed and is decremented by a chosen threshold when an `event wait` statement is executed. This allows for the case where several actions are needed before another action can take place.

The type `event_type` is extensible and has no type parameters. All its components are private. An event variable may be defined only by appearance in an `event post` or `event wait` statement. It may be referenced or defined in a segment that is unordered with respect to another segment in which it is defined.

The `event post` statement is an image control statement that takes the general form

`event post (` *event-variable [, stat-list]* `)`

with optional `stat=`, and `errmsg=` specifiers in its *stat-list*. Successful completion of the statement atomically increases its event count by 1. If there is an error condition, the value of the event count is processor dependent.

The `event wait` statement is an image control statement that takes the general form

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

with optional `until_count=`, `stat=` and `errmsg=` specifiers in its *wait-list*. The event variable must not be coindexed. An `until_count=` specifier has the form

`until_count =` *scalar-integer-expression*

where the value of the expression provides the threshold. The threshold is 1 if there is no `until_count=` specifier. The executing image waits until the event count is at or above the threshold, then atomically decreases the count by the threshold and resumes execution. If the threshold is k, the first k unmatched `event post` executions for the event variable are matched with this `event wait` execution. After an error condition, the value of the event count is processor dependent.

The value of an event count may be determined by the intrinsic subroutine `call event_query( event, count[, stat] )`

`event` is an event variable that is not coindexed. It has intent `in`.

`count` is a scalar integer with decimal range at least that of default integer. It has intent `out`. It is atomically given the value of the count of `event`.

`stat` is scalar integer with decimal range at least 4. It may not be coindexed. If present and no error condition occurs, it is given the value 0. If an error condition occurs and `stat` is present, it is given a processor-dependent positive value; otherwise error termination is initiated.

3.9 `Sync team` statement

The `sync team` statement has been introduced to allow synchronization within an ancestor team without leaving a `change team` construct or within a child team to which the executing image belongs without entering a `change team` construct. It has the general form

`sync team (` *[team-value [, stat-list]* `)`

where *team-value* is of type `team_type` and identifies the child team, the current team, or an ancestor team. Successful execution synchronizes all the images of the specified team in the same way as `sync all` does for the current team.

The *stat-list* returns information about the success of the execution. There may be at most one `stat=` specifier and at most one `errmsg=` specifier.

3.10 Image selectors

An image selector may have a `stat=` specifier to permit the programmer to detect the case where the image selected has failed:

`a[` *cosubscript-list[*`, stat=`*stat-variable]* `]`

If the image has failed, the `stat=` specifier is given the value `stat_failed_image`. Otherwise, it is given the value zero. Execution always continues if the image has failed, whether or not there is a `stat=` specifier. The value obtained on a reference is processor dependent. For a definition, there is no effect except for defining the `stat=` specifier if it appears.

Consider a coarray `a` that is in scope at a `change team` statement. If this is accessed in the `change team` construct using its name, cosubscripts map to an image index just as they do outside the construct, but this refers to an image within the current team. However, it is likely that data from other teams will need to be accessed from time to time, subject to suitable synchronization. Significant overheads are likely to be associated with leaving the construct, performing the data exchange, and changing teams again. Instead, an image of a sibling team may be accessed thus

`a[`*cosubscript-list*`, team_number=`*scalar-integer-expression*`]` where the `team_number` value is one of those used to identify teams in the `form team` statement that was executed to create the currrent team. The coarray must be established (Section 3.11) in an ancestor of the current team. An image of an ancestor team (or the current team) may be accessed thus

`a[`*cosubscript-list*`, team=`*team-variable* `]`

where *team-variable* is a scalar variable of type `team_type`. The coarray must be established (Section 3.11) in the team or an ancestor of it.

A `stat=` specifier may appear as well as a `team_number=` or `team=` specifier and may be placed before or after it.

3.11 Procedure calls and teams

When a procedure with a coarray dummy argument is called, the current team does not change but its siblings and ancestors are not available to the dummy coarray in image selectors. Indeed, it is hard to see how code could be written in the procedure to cope with different sets of nested `change team` constructs at the point of invocation. Because of this, the concept of 'establishment' for a coarray in a team has been introduced.

A nonallocatable coarray with the `save` attribute is ***established*** in the initial team. An allocated allocatable coarray is ***established*** in the team in which it was allocated. An unallocated allocatable coarray is not established. An associating coarray in a `change team` construct (Section 3.4) is ***established*** in the team that is current in the `change team` construct. A nonallocatable coarray that is an associating entity in an `associate`, `select rank`, or `select type` construct is ***established*** in the team in which the `associate`, `select rank`, or `select type` statement is executed. A nonallocatable coarray that is a dummy argument or host associated with a dummy argument is ***established*** in the team in which the procedure was invoked. A coarray dummy argument is not established in any ancestor team.

3.12 Intrinsic functions `get_team` and `team_number`

The transformational intrinsic function

`get_team (` *[* `level` *]* `)`

returns a value of type `team_type`.

`level` is an optional integer scalar with value one of the constants `initial_team`, `parent_team`, and `current_team` in the intrinsic module `iso_fortran_env`.

The value returned is that of a team variable for the current team if `level` is not present or the team indicated by `level` if it is present.

This is the only way to obtain a team value for the initial team and will be needed to refer to it in a `sync_team` statement or image selector when executing in a child team.

The transformational intrinsic function

`team_number ( [ team ] )`

returns a value of type default integer. It is the team number (Section 3.3) of the executing image within the specified team.

`team` is an optional scalar value of type `team_type` that specifies the current or an ancestor team. Absence specifies the current team.

This allows the executing image to determine in which team it lies and execute appropriate code, as illustrated in Figure 1.

Figure 1: Use of `team_number`.

```
change team (odd_even)
   select case (team_number())
      case (1)
        :  ! Code for images in team 1.
      case (2)
        :  ! Code for images in team 2.
   end select
 :
end team
```

3.13 Intrinsic function `image_index`

The intrinsic function `image_index` now has three forms:

```
image_index ( coarray, sub )
image_index ( coarray, sub, team )
image_index ( coarray, sub, team_number )
```

`coarray` is a coarray of any type.

`sub` is a rank-one integer array whose size is the corank of `coarray`.

`team` is a scalar value of type `team_type` that specifies the current or an ancestor team.

`team_number` is an integer scalar value identifying a sibling team of the current team.

The relevant team for the three forms of invocation is the current team, the team specified by `team`, or the sibling team specified by `team_number`. If `sub` holds a valid sequence of cosubscripts for `coarray` in the relevant team, the result is the corresponding image index. Otherwise, the result is zero.

3.14 Intrinsic function `num_images`

The intrinsic function `num_images` now has three forms:

```
num_images ( )
num_images ( team )
num_images ( team_number )
```

`team` is a scalar value of type `team_type` that specifies the current or an ancestor team.

`team_number` is an integer scalar value identifying a sibling team of the current team.

The relevant team for the three forms of invocation is the current team, the team specified by `team`, or the sibling team specified by `team_number`. The result is the number of images in the relevant team.

3.15 Intrinsic function `this_image`

The intrinsic function `this_image` now has three forms:

```
this_image ( [team] )
this_image ( coarray[, team] )
this_image ( coarray, dim[, team] )
```

`team` is a scalar value of type `team_type` that specifies the current or an ancestor team.

`coarray` is a coarray of any type. If allocatable, it must be allocated. If it is of type `team_type`, the argument `team` must be present.

`dim` is an integer scalar value in the range $1 \leq$ `dim` $\leq n$ where n is the corank of `coarray`.

The result is of type default integer. The relevant team is the team specified by `team` or the current team if `team` is absent. If `coarray` is absent, the result is the image index of the executing image in the relevant team. If `coarray` is present and `dim` is not, the result is a rank-one array holding the sequence of cosubscript values for `coarray` that would specify the executing image in the relevant team. If `coarray` and `dim` are present, the result is the value of cosubscript `dim` in the sequence of cosubscript values for `coarray` that would specify the executing image in the relevant team.

3.16 Intrinsic function `move_alloc`

It has been realized that the intrinsic function `move_alloc` cannot be pure if its arguments are coarrays because it then involves synchronization. It has had additional optional arguments `stat` and `errmsg` added and now has the form:

```
call move_alloc( from, to[, stat, errmsg] )
```

`from` is allocatable and of any type, rank, and corank. It has intent `inout`.

`to` is allocatable and of the same rank and corank as `from`. It must be type compatible with `from` and polymorphic if `from` is polymorphic. It has intent `out`. Each nondeferred type parameter value of the declared type of `to` must be the same as the corresponding value for `from`.

`stat` is an optional integer scalar with a decimal exponent range of at least 4. It has intent `out`. It must not be coindexed. If it is not present and an error occurs, error termination is initiated. If the invocation is successful, it is given the value zero. If an error occurs, it is given a nonzero value. If there is a stopped image, it is given the value `stat_stopped_image`. Otherwise, if there is a failed image, it is given the value `stat_failed_image`. Otherwise, it is given a value other than `stat_stopped_image` or `stat_failed_image`.

`errmsg` is an optional intent `inout` scalar of type default character. It must not be coindexed. When present, it provides an explanatory message in the event of an error condition.

Successful execution is as in Fortran 2008 except that it applies to the current team. If `stat` is given the value `stat_failed_image`, execution takes place on the active images.

3.17 `Fail image` statement

The statement

```
fail image
```

causes the executing image to behave as if it has failed. No further statements are executed by the image. It allows testing of a code that has been designed to recover from an image failure.

3.18 Detecting failed and stopped images

Three new intrinsic procedures have been added to assist the detection of failed and stopped images.

```
failed_images ( [team, kind] )
stopped_images ( [team, kind] )
```

are transformational functions that return a rank-one integer array of size equal to the number of images in the team that are known to have failed or stopped.

`team` is an optional scalar value of type `team_type` that specifies the current or an ancestor team. Absence specifies the current team.

`kind` is an optional integer scalar constant value that specifies an integer kind for the result array. Absence specifies the kind of default integer.

The values of the elements of the result array are the image indices of the failed or stopped images, in increasing order.

```
image_status ( image[, team] )
```

is an elemental function whose result is of type default integer.

`image` is a integer. Its value must be the image index of an image in `team`.

`team` is an optional scalar value of type `team_type` that specifies the current or an ancestor team. Absence specifies the current team.

The value of the result is `stat_failed_image` if `image` has failed, `stat_stopped_image` if `image` has stopped, or zero otherwise.

3.19 Collective subroutines

Intrinsic subroutines have been added to perform collective operations on all the images of a team, such as summing the values of a variable across the images. It is to be expected that the execution will have been optimized by the system, for example, by associating the images of the team with the leaves of a binary tree and grouping the operations by tree level. The subroutines are invoked by one of these statements

```
call co_broadcast( a, source_image[, stat, errmsg] )
call co_max( a[, result_image, stat, errmsg] )
call co_min( a[, result_image, stat, errmsg] )
call co_sum( a[, result_image, stat, errmsg] )
call co_reduce( a, operation[, result_image, stat, errmsg] )
```

The same statement must be executed on all active images of the team and it must occur in a context that would allow an image control statement. There is no automatic synchronization at the statement, but it is to be expected that the system applies some form of synchronization while executing the subroutine. To avoid the possibility of these synchronizations causing deadlock, the sequence of invocations must be the same on all active images of the team from the beginning to the end of execution as a team.

The arguments are as follows

`a` is a scalar or an array that has the same shape on all the images of the current team. It has intent `inout`. It must not be coindexed. It may be a coarray but this is not required.

`source_image` is an intent `in` integer scalar that specifies the image from which values are broadcast. It must have the same value on all images of the team.

`result_image` is an optional intent `in` integer scalar. If present, it must have the same value on all images of the team. It specifies the image on which the result is placed and `a` becomes undefined on all other images of the team. If it is not present, the result is broadcast to all images of the team.

`operation` is a pure function with two scalar dummy arguments and a scalar result all of the type and type parameters of `a`. Neither argument may be polymorphic, allocatable, optional, or a pointer. If one has the `asynchronous`, `target`, or `value` attribute, the other must too. It must be the same function on all images of the team and must implement an operation that is associative apart from the effects of rounding.

`stat` is an optional intent `out` integer scalar with a decimal exponent range of at least four. If it is not present and an error occurs, error termination is initiated. If present on one image, it must be present on all images of the team. It must not be coindexed. If the invocation is successful, it is given the value zero. If an error occurs, it is given a nonzero value and the argument `a` becomes undefined. If there is a stopped image, it is given the value `stat_stopped_image`. Otherwise, if there is a failed image, it is given the value `stat_failed_image`. Otherwise, it is given a value other than `stat_stopped_image` or `stat_failed_image`.

`errmsg` is an optional intent `inout` scalar of type default character. It must not be coindexed. When present, it provides an explanatory message in the event of an error condition.

The actions of the collectives are as follows

`co_broadcast` copies the value of `a` on `source_image` to the corresponding argument on all the other images of the team as if by intrinsic assignment. The variable `a` may have any type, but its dynamic type and type parameter values must be the same on all images of the team.

`co_max` and `co_min` compute the maximum or minimum value of `a` on all images of the team. The result is placed in `a` on `result_image` if this is present and otherwise in `a` on all images of the team. The variable `a` may be of type integer, real, or character, but must have the same type and type parameters on all images of the team. If it is an array, the result is computed element by element.

`co_sum` computes the sum of the values of `a` on all images of the team. The variable `a` may be of any numeric type, but must have the same type and type parameters on all images of the team. If it is an array, the result is computed element by element. The order of summation is not specified so the result may be affected by rounding errors that can vary even between different runs on the same computer. The result is placed in `a` on `result_image` if this is present and otherwise exactly the same result is placed in `a` on all images of the team.

`co_reduce` behaves like `co_sum` but is for an operation that has been written by the programmer as a function. The variable `a` may be of any type but must not be polymorphic.

3.20 New and enhanced atomic subroutines

Nine atomic subroutines have been added and the two old ones have an added argument. The new ones are invoked by one of these statements

```
call atomic_add ( atom, value[, stat] )
call atomic_and ( atom, value[, stat] )
call atomic_or ( atom, value[, stat] )
call atomic_xor ( atom, value[, stat] )
call atomic_fetch_add ( atom, value, old[, stat] )
call atomic_fetch_and ( atom, value, old[, stat] )
call atomic_fetch_or ( atom, value, old[, stat] )
call atomic_fetch_xor ( atom, value, old[, stat] )
call atomic_cas ( atom, old, compare, new[, stat] )
```

The arguments are as follows

`atom` is a scalar coarray or coindexed object of type `integer(atomic_int_kind)` or, alternatively, for `atomic_cas` only, `logical(atomic_logical_kind)`. It has intent `inout`.

`value` is an integer scalar. It has intent `in`.

`old` is a scalar of the same type and kind as `atom`. It has intent `out`.

`compare` is a scalar of the same type and kind as `atom`. It has intent `in`.

`new` is a scalar of the same type and kind as `atom`. It has intent `in`.

`stat` is an optional integer scalar with a decimal exponent range of at least 4. It must not be coindexed. It has intent `out`. If is not present and an error occurs, error termination is initiated. If the invocation is successful, it is given the value zero. If an error occurs, it is given a nonzero value and any `atom` or `old` arguments become undefined. If `atom` is on a failed image and there is no other cause for the error, the value given is `stat_failed_image`.

The actions of the subroutines are as follows

`atomic_add` causes `atom` to be given the value `atom+int(value,atomic_int_kind)`.

`atomic_and` causes `atom` to be given the value `iand(atom,int(value,atomic_int_kind))`.

`atomic_or` causes `atom` to be given the value `ior(atom,int(value,atomic_int_kind))`.

`atomic_xor` causes `atom` to be given the value `ieor(atom,int(value,atomic_int_kind))`.

`atomic_fetch_add` causes `atom` to be given the value `atom+int(value,atomic_int_kind)` and `old` to be given the value `atom` had on entry.

`atomic_fetch_and` causes `atom` to be given the value `iand(atom,int(value,atomic_int_kind))` and `old` to be given the value `atom` had on entry.

`atomic_fetch_or` causes `atom` to be given the value `ior(atom,int(value,atomic_int_kind))` and `old` to be given the value `atom` had on entry.

`atomic_fetch_xor` causes `atom` to be given the value `ieor(atom,int(value,atomic_int_kind))` and `old` to be given the value `atom` had on entry.

`atomic_cas` tests whether the value of `atom` is equal to that of `compare`. If they are equal, it causes `atom` to be given the value of `int(new,atomic_int_kind)` if it is of type integer and `new` if it is of type logical. Otherwise, the value of `atom` is not changed. In either case, `old` is given the value `atom` had on entry.

If the `stat` argument is present and no error condition occurs, it is given the value 0. If an error condition occurs, any `atom` or `old` argument becomes undefined. This argument has also been added as an optional final argument with the same effect to the intrinsic subroutines `atomic_define` and `atomic_ref`.

3.21 Failed images and `stat=` specifiers

If a `stat=` specifier is present in a `change team`, `end team`, `event post`, `form team`, `sync all`, `sync images`, or `sync team` statement and one of the images involved has failed but none has stopped and no other error condition occurs, the `stat=` specifier is given the value `stat_failed_image` and the intended action takes place on the active images involved.

4 Conformance with ISO/IEC/IEEE 60559:2011

A large number of changes to the intrinsic modules `ieee_arithmetic`, `ieee_exceptions`, and `ieee_features` have been made for conformance with the new IEEE standard for floating-point arithmetic.

4.1 Subnormal values

The new IEEE standard uses the term ‘subnormal’ instead of ‘denormal’ for a value with magnitude less than any normal value and less precision.

The named constants `ieee_negative_subnormal` and `ieee_positive_subnormal` of type `ieee_class_type` have been added to the module `ieee_arithmetic` with the same values as `ieee_negative_denormal` and `ieee_positive_denormal`, respectively.

The named constant `ieee_subnormal` of type `ieee_features_type` has been added to the module `ieee_features` and has the same value as `ieee_denormal`.

The inquiry function `ieee_support_subnormal` has been added to `ieee_arithmetic` and is exactly the same as `ieee_support_denormal` except for its name.

4.2 Type for floating-point modes

The type `ieee_modes_type` has been added to `ieee_exceptions` for storing all the floating-point modes, that is, the rounding modes, underflow mode, and halting mode. Also added to `ieee_exceptions` are subroutines for getting and setting the modes.

`call ieee_get_modes(modes)` where `modes` is a scalar of type `ieee_modes_type` that has intent `out` and is assigned the value of the floating-point modes.

`call ieee_set_modes(modes)` where `modes` is a scalar of type `ieee_modes_type` that has intent `in` and has a value obtained by a previous call of `ieee_get_modes`. The floating-point modes are restored to their values at the time of the previous call.

4.3 Rounding modes

The new IEEE standard has two modes for rounding to the nearest representable number, which differ in the way they handle ties: in the case of a tie, *roundTiesToEven* uses the value with an even least significant digit and *roundTiesToAway* uses the value further from zero. The value `ieee_nearest` of the type `ieee_round_type` corresponds to *roundTiesToEven* and the value `ieee_away` has been added to correspond to *roundTiesToAway*.

The optional argument `radix` has been added to the subroutines `ieee_get_rounding_mode` and `ieee_set_rounding_mode` to allow the decimal rounding mode to be inquired about and set independently of the binary rounding mode. The argument `radix` has type integer and must have the value 2 or 10. `ieee_set_rounding_mode (round_value, radix)` must not be invoked unless there is some support for this radix.

4.4 Rounded conversions

The elemental function `ieee_real` has been added to `ieee_arithmetic` for rounded conversion to real type, as specified in the IEEE standard by the convertFromInt and convertFormat operations.

`ieee_real(a[, kind])` where `a` is of type real or integer and `kind` is a scalar integer constant expression. The result is of type default real if kind is absent and of type `real(kind)` otherwise. The result has the same value as `a` if that value is representable in the representation method of the result and is rounded according to the rounding mode otherwise.

The optional argument `round` of type `ieee_round_type` has been added to `ieee_rint`. The result for `ieee_rint(x,round)` is the value of `x` rounded to an integer according to the mode specified by `round`.

4.5 Fused multiply-add

The elemental function `ieee_fma` has been added to `ieee_arithmetic` for the fused multiply-add operation.

`ieee_fma(a,b,c)` where `a`, `b`, and `c` are real with the same kind type parameter returns the mathematical value of a×b+c rounded according to the rounding mode. `ieee_overflow`, `ieee_underflow`, or `ieee_inexact` are signaled according to the final step in the calculation and not by any intermediate calculation.

4.6 Test sign

The elemental function `ieee_signbit` has been added to `ieee_arithmetic` to test the sign of a real as specified in the IEEE standard by the isSignMinus operation.

`ieee_signbit(x)` where `x` is real returns a value of type default logical that is true if and only if the sign of `x` is minus.

4.7 Conversion to integer type

The elemental function `ieee_int` has been added to `ieee_arithmetic` for conversion to integer type.

`ieee_int(a, round[, kind])` where `a` is of type real, `round` is of type `ieee_round_type`, and `kind` is an integer constant expression returns a value of type integer. If `kind` is present, it gives the kind type parameter of the result; otherwise, the result is of type default integer. The value of the result is that of `a` converted to an integer according to the rounding mode specified by `round` provided this value is representable in the result; otherwise, `ieee_invalid` is signaled and the result is processor dependent. The processor is required to consistently signal `ieee_inexact` if the result is not exact or not do so.

4.8 Remainder function

The arguments of the elemental function `ieee_rem` are now required to have the same radix.

4.9 Maximum and minimum values

The elemental functions `ieee_max_num`, `ieee_max_num_mag`, `ieee_min_num`, and `ieee_min_num_mag` have been added for the IEEE operations of maxNum, maxNumMag, minNum, and minNumMag.

```
ieee_max_num(x,y)
ieee_max_num_mag(x,y)
ieee_min_num(x,y)
ieee_min_num_mag(x,y)
```

where `x` and `y` are real with the same kind type parameter, returns 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 one of `x` and `y` is a quiet NaN, the result is the other. If one or both of `x` and `y` are signaling NaNs, `ieee_invalid` signals and the result is a NaN. Otherwise, which is returned is processor dependent for `ieee_max_num(x,y)` and `ieee_min_num(x,y)`; it is `ieee_max_num(x,y)` for `ieee_max_num_mag(x,y)`; and it is `ieee_min_num(x,y)` for `ieee_min_num_mag(x,y)`.

4.10 Adjacent machine numbers

The elemental functions `ieee_next_down` and `ieee_next_up` have been added for for the IEEE operations of nextDown and nextUp.

`ieee_next_down(x)` where `x` is real returns the greatest value in the representation method of `x` that compares less than `x`, except that when `x` is equal to $+\infty$ the result has the value $+\infty$, and when `x` is a NaN the result is a NaN. If `x` is a signaling NaN, `ieee_invalid` signals; otherwise, no exception is signaled.

`ieee_next_up(x)` where `x` is real returns the least value in the representation method of `x` that compares greater than `x`, except that when `x` is equal to $-\infty$ the result has the value $-\infty$, and when `x` is a NaN the result is a NaN. If `x` is a signaling NaN, `ieee_invalid` signals; otherwise, no exception is signaled.

If `ieee_support_datatype(x)` has the value false, these functions must not be invoked. If `ieee_support_inf(x)` has the value false, `ieee_next_down(x)` must not be invoked when `x` has the value `-huge(x)` and `ieee_next_up(x)` must not be invoked when `x` has the value `huge(x)`.

4.11 Comparisons

The elemental functions `ieee_quiet_eq`, `ieee_quiet_ge`, `ieee_quiet_gt`, `ieee_quiet_le`, `ieee_quiet_lt`, and `ieee_quiet_ne` have been added for the IEEE operations of compareQuietEqual, compareQuietGreaterEqual, compareQuietGreater, compareQuietLess-Equal, compareQuietLess, and compareQuietNotEqual.

```
ieee_quiet_eq(a,b)
ieee_quiet_ge(a,b)
ieee_quiet_gt(a,b)
ieee_quiet_le(a,b)
ieee_quiet_lt(a,b)
ieee_quiet_ne(a,b)
```

where `a` and `b` are real with the same kind type parameter returns true if `a` compares with `b` as equal, greater than or equal, greater than, less than or equal, less than, or not equal, respectively, and false otherwise. If `a` or `b` is a NaN, the result will be false. If `a` or `b` is a signaling NaN, `ieee_invalid` signals; otherwise, no exception is signaled.

The elemental functions `ieee_signaling_eq`, `ieee_signaling_ge`, `ieee_signaling_gt`, `ieee_signaling_le`, `ieee_signaling_lt`, and `ieee_signaling_ne` have been added for the IEEE operations of compareSignalingEqual, compareSignalingGreaterEqual, compareSignaling-Greater, compareSignalingLessEqual, compareSignalingLess, and compareSignalingNotEqual.

```
ieee_signaling_eq(a,b)
ieee_signaling_ge(a,b)
ieee_signaling_gt(a,b)
ieee_signaling_le(a,b)
ieee_signaling_lt(a,b)
ieee_signaling_ne(a,b)
```

where `a` and `b` are real with the same kind type parameter returns true if `a` compares with `b` as equal, greater than or equal, greater than, less than or equal, less than, or not equal, respectively, and false otherwise. If `a` or `b` is a NaN, the result will be false and `ieee_invalid` signals; otherwise, no exception is signaled.

If x_1 and x_2 are of numeric types and the type of x_1+x_2 is real, comparisons are made as follows

x_1`==`x_2 or x_1`.eq.`x_2	compareQuietEqual
x_1`>=`x_2 or x_1`.ge.`x_2	compareSignalingGreaterEqual
x_1`>`x_2 or x_1`.gt.`x_2	compareSignalingGreater
x_1`<=`x_2 or x_1`.le.`x_2	compareSignalingLessEqual
x_1`<`x_2 or x_1`.lt.`x_2	compareSignalingLess
x_1`/=`x_2 or x_1`.ne.`x_2	compareQuietNotEqual

If x_1 and x_2 are of numeric types and the type of x_1+x_2 is complex, comparisons are made as follows

x_1`==`x_2 or x_1`.eq.`x_2	compareQuietEqual
x_1`/=`x_2 or x_1`.ne.`x_2	compareQuietNotEqual

5 Features that address deficiencies and discrepancies

5.1 Default accessibility for entities accessed from a module

If a module `a` uses module `b`, the default accessibility for entities it accesses from `b` is `public`. Specifying another accessibility for each entity is awkward and error prone. It is now possible for the name of a module to be included in the list of names of entities made public or private on a `public` or `private` statement. This sets the default for all entities accessed from that module. The name must appear at most once in all the `public` and `private` statements in the module.

5.2 `Implicit none` enhancement

The syntax of the `implicit none` statement has been extended to

> `implicit none` *[* `(` *[implicit-none-spec-list]* `)` *]*

where each *implicit-none-spec* is `external` or `type` and appears at most once. The appearance of `external` requires that the names of external and dummy procedures with implicit interfaces in the scoping unit and any contained scoping units be explicitly declared to have the `external` attribute. The appearance of `type` requires the types of all data entities in the scoping unit and any contained scoping units to be explicitly declared. The unqualified `implicit none` statement has the same effect as `implicit none (type)`.

5.3 Referencing a property of an object in a constant expression

The rules on what may appear in a constant expression have been relaxed to allow such things as

```
integer :: b = bit_size(b)
real :: e = sqrt(sqrt(epsilon(e)))
integer :: iota(10) = [ ( i, i = 1, size(iota,1) ) ]
```

In Fortran 2018, a constant expression may depend on a type parameter or an array bound of an entity specified to the left in the same statement, but not within the same expression. For example, the following remain prohibited:

```
integer :: x(10,size(x,1))
character(*),parameter :: c = repeat('c',len(c))
```

5.4 Enhancements to `inquire`

If a `recl=` inquiry is made in an `inquire` statement in Fortran 2008 and there is no connection or the connection is for stream access, the variable becomes undefined. In Fortran 2018, it is assigned the value -1 if there is no connection and the value -2 if the connection is for stream access.

If a `pos=` or `size=` inquiry is made in an `inquire` statement and there are pending data transfer operations for the specified unit, the value assigned is computed as if all the pending data transfers had already completed.

5.5 `d0.d, e0.d, es0.d, en0.d, g0.d` and `e`*w.d*`e0` edit descriptors

It was anomalous that the field width *w* was allowed to be given as zero for some output `g` editing, but not for `d`, `e`, `es`, `en`, and all `g` output editing. This has been corrected. If the value is zero, the processor selects the smallest positive actual field width that does not result in a field filled with asterisks. A zero value for the field width is not permitted for input.

The `g0.d` edit descriptor can be used to specify the output of integer, logical, and character data. It follows the rules for the `i0`, `l1`, and `a` edit descriptors, respectively.

The value 0 may be given to an exponent digits parameter in an `e`*w*`.`*d*`e`*e* edit descriptor. The effect of this is that the processor uses the minimum number of digits required to represent the exponent value. This avoids leading zeros, which are unsightly and can cause errors if the value is read in another language.

5.6 Formatted input error conditions

An error condition occurs if the input field presented for logical or numeric editing during execution of a formatted input statement does not have one of the standard forms and is not acceptable to the processor.

5.7 Rules for generic procedures

In Fortran 2018, if one procedure has more nonoptional dummy procedures than the other has nonoptional dummy procedures, they may be identified by the same generic name. It was anomalous that this means of disambiguation was absent from Fortran 2008.

5.8 Enhancements to `stop` and `error stop`

The stop code in a `stop` or `error stop` statement is no longer restricted to a constant expression. It can be any scalar expression of type integer or default character.

Output of the stop code and exception summary from a `stop` or `error stop` statement can be controlled with a `quiet=` specifier, for example,

```
      stop failure_message, quiet=no_messages
```

If the `quiet=` specifier is true, no stop code or exception summary is output. Any scalar logical expression may be used for the `quiet=` specifier.

5.9 Intrinsics that access the computing environment

An extra optional argument `errmsg` has been added to each of the intrinsic subroutines `get_command`, `get_command_argument`, and `get_environment_variable`. It is a scalar intent `inout` argument of type default character and returns an error message if an error occurs. In the case of a warning situation that would assign -1 to the argument `status`, `errmsg` is left unchanged. This change brings these procedures into line with the ability to retrieve error messages for I/O statements via the `iomsg=` specifier, image control statements via the `errmsg=`

specifier and invocations of `execute_command_line` via the `cmdmsg` argument.

The effects of invoking the intrinsic procedures `command_argument_count`, `get_command`, `get_command_argument`, and `get_environment_variable`, on images other than image one of the initial team are now as on image one. This is important with the addition of teams, because a team need not include image one. In the case of `get_environment_variable`, it is processor dependent whether an environment variable that exists on an image also exists on another image, and if it does exist on both images, whether the values are the same.

5.10 New elemental intrinsic function `out_of_range`

The new elemental intrinsic function `out_of_range` has been added to test whether a real or integer value can be safely converted to a different real or integer type and kind.

`out_of_range(x,mold`*[*`,round`*]*`)`

`x` is of type real or integer.

`mold` is of type real or integer.

`round` is of type logical and may be present only if `x` is of type real and `mold` is of type integer.

The arguments `mold` and `round` are required to be scalars. The result is of type default logical. It has the value true if and only if

- the value of `x` is an IEEE infinity or NaN, and `mold` is of type integer or is of type real and of a kind that that does not support such a value;
- `mold` is of type integer, `round` is absent or present with the value false, and the integer with largest magnitude that lies between zero and `x` inclusive is not representable by objects with the type and kind of `mold`;
- `mold` is of type integer, `round` is present with the value true, and the integer nearest `x`, or the integer of greater magnitude if two integers are equally near to `x`, is not representable by objects with the type and kind of `mold`; or
- `mold` is of type real and the result of rounding the value of `x` to the extended model for the kind of `mold` has magnitude larger than that of the largest finite number with the same sign as `x` that is representable by objects with the type and kind of `mold`.

5.11 New reduction intrinsic `reduce`

The new transformational intrinsic function `reduce` has been added to match the collective subroutine `co_reduce` that was included in the Technical Specification for additional parallel features.

`reduce(array,operation`*[*`,mask,identity,ordered`*]*`)` or

`reduce(array,operation,dim`*[*`,mask,identity,ordered`*]*`)`

> `array` is an array of any type.
>
> `operation` is a pure function with two arguments and result of the type and type parameters of `array`. It implements a mathematically associative operation that need not be commutative. The arguments and result must not be polymorphic.
>
> `dim` is an integer scalar with value 1 <= `dim` <= n, where n is the rank of `array`.
>
> `mask` is logical and conformable with `array`.
>
> `identity` is a scalar of the type and type parameters of `array`.
>
> `ordered` is a logical scalar.

This function returns a scalar of the type and type parameters of `array`. Starting with the sequence of elements of `array` in array element order, an adjacent pair of elements are replaced by the result of applying `operation` to the pair; this is repeated until there is only one element, which is the result. If `ordered` is present with the value true, the chosen pair is always the first two elements. If `mask` is present, the starting sequence consists of those elements of `array` for which `mask` is true, in array element order. If the sequence is empty, the result is `identity` if it is present; otherwise, error termination is initiated. If `dim` is present, the operation is applied to all rank-one sections that span through dimension `dim` to produce an array of rank reduced by one and extents equal to the extents in the other dimensions, or a scalar if the original rank is one.

5.12 Intrinsic functions `image_index`, `lcobound`, `ucobound`, and `this_image`

If a coarray is of a derived type in Fortran 2008, a subobject that is formed by taking a component is also a coarray but it cannot have cosubscripts. For example, given the declarations

```
type t
   real r
end type
type(t) x[10,*]
```

`x%r` is a coarray that cannot have cosubscripts. However, it may be passed as an actual argument to a procedure where the corresponding dummy argument is a named coarray that can have cosubscripts.

The intrinsic functions `image_index`, `lcobound`, `ucobound`, and `this_image` all have an argument `coarray` and are defined in terms of the possible cosubscript values that `coarray` may have. They were not intended to be called for a coarray that cannot have cosubscripts. This is now disallowed.

5.13 Intrinsic function `coshape`

The intrinsic function `coshape` has been added for consistency with `lcobound` and `ucobound`.

`coshape(coarray` *[*`,kind`*]* `)`

> `coarray` is a coarray that is permitted to have subscripts. It may be of any type. If it is allocatable, it must be allocated.
>
> `kind` is a scalar integer constant expression.

The result is a rank-one integer array whose size is the corank of `coarray`. It is of default kind if `kind` is not present and of kind `kind` if it is present. The value of element i is 1+`ucobound(coarray,`i`)-lcobound(coarray,`i`)`.

5.14 Intrinsic subroutine `random_init`

Two problems have been found in Fortran 2008 with the random number generator `random_number`:

1. Some processors always initialize the pseudo-random sequence in the same way. Others purposely initialize it randomly.
2. A processor that generates exactly the same pseudo-random sequence on all the images conforms to the standard.

The intrinsic subroutine `random_init` has been added to control of the initiation of the sequence and address these problems.

`random_init (repeatable, image_distinct)`

> `repeatable` is a logical scalar of intent `in`. If it has the value true, the seed accessed by the pseudorandom number generator is set to a processor-dependent value that is the same each time `random_init` is called from the same image. If it has the value false, the seed is set to a processor-dependent, unpredictably different value on each call.
>
> `image_distinct` is a logical scalar of intent `in`. If it has the value true, the seed accessed by the pseudorandom number generator is set to a processor-dependent value that is distinct from the value that would be set by a call to `random_init` by another image. If it has the value false, the value to which the seed is set does not depend on which image calls `random_init`.

When executing on more than one image, there clearly has to be some ordering of calls of `random_init`, `random_seed`, and `random_number`. The requirement is that there must be an ordering between a segment in which `random_init` or `random_seed` is called and another segment in which `random_init`, `random_seed`, or `random_number` is called. It is processor dependent whether each image uses a separate random number generator, or if some or all images use common random number generators.

5.15 Intrinsic function `sign`

The arguments to the intrinsic function `sign` can be of different kind.

5.16 Intrinsic functions `extends_type_of` and `same_type_as`

Because the dynamic type of a nonpolymorphic pointer is always well defined, nonpolymorphic pointer arguments to the intrinsic functions `extends_type_of` and `same_type_as` need not have defined pointer association status.

5.17 Detecting nonstandard intrinsics

In Fortran 2018, the processor is required to have the capability to detect and report the use of a nonstandard intrinsic procedure, a nonstandard intrinsic module, and a nonstandard procedure of a standard intrinsic module.

5.18 Kind of the do variable in implied do

In Fortran 2008, for a `do concurrent` construct, `forall` construct, or `forall` statement, the implied do index or indices have the scope of the construct or statement and can be declared there, for example,

```
do concurrent (integer(long)::i=1:n, j=1:m, i/=j)
forall (integer(long)::idx=100:200) a(idx, idx) = idx**2
```

but this was not the case for implied do indices in array constructors and data statements. These are now allowed, for example,

```
[ (a(i,i), integer(long) :: i=1,n) ]
data ((a(i,j), integer(long) :: i=1,5,2), j=1,5) /15*0./
```

In these cases, the type must be `integer` and applies only to the do indices of the implied do in which it occurs.

5.19 Locality clauses in `do concurrent`

The rules on the use of a variable within a `do concurrent` construct allow the implementation to execute it concurrently while working with each variable involved being either shared between all iterations or held in separate local memory for each iteration. Sometimes the implementation has to allow for situations that the user knows will not occur and performance suffers.

To provide guidance to the implementation, a `do concurrent` statement may include specifications that make a variable that appears in the construct have `local`, `local_init`, or `shared` locality in the construct.

A variable with `local` or `local_init` locality is treated within the construct as being separate from the variable with the same name outside the construct. In the `local` case, the initial value of the variable within the construct is undefined, whereas in the `local_init` case, it takes the value or pointer association status of the outside variable. In neither case does the value of the outside variable change at the end of execution of the construct.

A variable with `shared` locality is treated within the construct as being the same as the variable with the same name outside the construct.

A variable without `local`, `local_init`, or `shared` locality is said to have unspecified locality and the rules of Fortran 2008 apply to it.

Localities are specified by one or more of the forms

> `local` (*variable-name-list*)
> `local_init` (*variable-name-list*)
> `shared` (*variable-name-list*)
> `default (none)`

appearing at the end of a `do concurrent` statement. Each variable named must be in the scope of the `do concurrent` statement, must not be named more than once in the statement, and must not be the variable that controls the loop execution. The appearance of `default (none)` indicates that no variable of the loop has unspecified locality.

A variable with `local` or `local_init` locality must not be allocatable, a coarray of intent `(in)`, an assumed-size array, of finalizable type, a nonpointer polymorphic dummy argument, an optional argument, or a variable that is not permitted to appear in a variable definition context. The construct variable has all the properties of the outside variable except that it does not have the `bind`, `intent`, `protected`, `save`, or `value` attribute, even if the outside variable does.

If a variable with `shared` locality is defined or becomes undefined during any iteration, it must not be referenced, defined, or become undefined during any other iteration. If it is allocated, deallocated, nullified, or pointer assigned during an iteration, it must not have its allocation or association status, dynamic type, array bounds, shape, or a deferred type parameter value inquired about in any other iteration. A noncontiguous array with `shared` locality must not be supplied as an actual argument corresponding to a contiguous intent `inout` dummy argument.

5.20 Control of host association

The `import` statement can be used in a contained subprogram or block construct to control host association. In addition to the old form

> `import` *[[::] import-name-list]*

it has the new forms

```
import, only:   import-name-list
import, none
import, all
```

If one `import` statement in a scoping unit is an `import,only` statement, they must all be, and only the entities listed become accessible by host association.

If an `import,none` statement appears in a scoping unit, no entities are accessible by host association and it must be the only `import` statement in the scoping unit. `import,none` is not permitted in the specification part of a submodule except within an interface body.

If an `import,all` statement appears in a scoping unit, all entities of the host are accessible by host association and it must be the only `import` statement in the scoping unit.

Each *import-name* must be the name of a host entity and must not appear in a context that makes the host entity inaccessible, such as being declared as a local entity. For `import,all` no entity of the host may be inaccessible for this reason. This restriction does not apply to the simple form of `import`.

5.21 Connect a file to more than one unit

It can be convenient to connect a file to more than one unit at a time. For example, the two units might be reading different parts of the same sequential file. While the prohibition against connecting a file to more than one unit at a time has been removed, there is no requirement on the processor to provide this functionality. For a given file and action choice, whether or not it is available is processor dependent.

5.22 Advancing input with `size=`

The prohibition against `size=` appearing with advancing input has been removed.

5.23 Extension to the `generic` statement

The `generic` statement can be used to declare generic interfaces outside a type definition. The syntax is similar to that of a `generic` statement in a type definition. It takes the form:

```
generic[[,access-spec]::]generic-spec => specific-procedure-list
```

where *access-spec* is `public` or `private`. It provides a compact way to associate a generic identifier with a set of procedures and declare whether it is `public` or `private`. The functionality is as for an interface block that does not contain interface bodies, perhaps with a `public` or `private` statement.

5.24 The `value` attribute for an argument of a defined operation or assignment

In Fortran 2008, a pure procedure may have an argument with the `value` attribute but a procedure with such an argument is not permitted for a defined operation or assignment. This is now permitted.

5.25 Removal of anomalies regarding pure procedures

Execution of an `error stop` statement causes the execution to cease as soon as possible on all images, so there is no need to disallow it in a `pure` procedure. It is now allowed and gives the programmer the opportunity to provide an explanation in the *stop-code*.

The standard procedures in the intrinsic module `iso_c_binding`, other than `c_f_pointer`, are now pure.

A dummy argument of a pure function is permitted in a variable definition context if it has the `value` attribute.

5.26 Recursive and non-recursive procedures

Procedures, apart from functions whose result is of type `character` with an asterisk character length[2], are now recursive by default and the keyword `non_recursive` has been added to allow a procedure to be specified as non-recursive. It can be used wherever `recursive` can be used. The restriction against elemental recursion was intended to make elemental procedures easier to implement and optimise, but recursion has become normal so it is not needed. Using a non-recursive procedure recursively is likely to produce invalid results on a system that does not detect such use, so it is safer to require the keyword `non_recursive` for the case where the user is sure that this is all that is wanted.

5.27 Simplification of calls of the intrinsic `cmplx`

In a call of the intrinsic function `cmplx` with an argument of type complex, the keyword was needed for the `kind` argument. This requirement has been removed by regarding it as having two overloaded forms

```
cmplx(x[,kind])
cmplx(x[,y,kind])
```

5.28 Removal of the restriction on argument `dim` of many intrinsic functions

The function `all` had the form

[2] An obsolescent feature – such functions cannot be recursive.

```
    all(mask[,dim])
```

and a prohibition against the actual argument corresponding to `dim` being an optional dummy argument. This restriction was needed because whether `dim` is present may affect the rank of the result. It has been made unnecessary by regarding the function as having two overloaded forms

```
  all(mask)
  all(mask,dim)
```

The same change has been made to `any`, `norm2`, `parity`, and `this_image`. The functions `findloc`, `iall`, `iany`, `iparity`, `maxloc`, `maxval`, `minloc`, `minval`, `product` and `sum` already have this form. In all these cases, the restriction on `dim` has been removed. Unfortunately, it remains for the function `count` because an ambiguity would be introduced by the overloaded form.

5.29 Kinds of arguments of intrinsic and IEEE procedures

Several of the intrinsic procedures and the IEEE module procedures require that their numerical or logical arguments be of default kind. This is an unnecessary restriction and has been changed. The change will make for better consistency. There is no change for character arguments. The arguments affected are

- Restriction changed to integer with a decimal exponent range of at least four:

```
date_and_time (., ., ., values)
execute_command_line (., ., ., cmdstat, .)
get_command (., length, status)
get_command_argument (., ., length, status)
get_environment_variable (., ., length, status, .)
```

- Restriction changed to integer with a decimal exponent range of at least nine:

```
execute_command_line (., ., exitstat, ., .)
```

- Restriction removed:

```
execute_command_line (., wait, ., ., .)
get_command_argument (number, ., ., .)
this_image (., dim)
ieee_get_flag (., flag_value)
ieee_get_halting mode (., halting)
ieee_get_underflow_mode (gradual)
ieee_set_flag (., flag_value)
ieee_set_halting_mode (., halting)
ieee_set_underflow_mode (gradual)
```

5.30 Hexadecimal input/output

A new edit descriptor has been introduced for hexadecimal output of real values. It has the forms `ex`*w.d* and `ex`*w.d*`e`*e*. Here, *w* gives the field width or is zero to specify that the processor should choose it. *d* gives the number of hexadecimal digits required or may be zero if all digits are wanted. The exponent is a power of 2, expressed as a decimal integer. Unless the value is zero, the exponent is such that the leading hexadecimal digit is 1. The hexadecimal point appears after the leading digit. For `ex`*w.d* and `ex`*w.d*`e0`, the exponent part contains the minimum number of digits needed to represent the exponent; otherwise, the exponent contains *e* digits. Table 4 shows some examples.

Table 4: Examples of hexadecimal output.

Internal value	Edit descriptor	Possible output
1.375	`ex0.1`	`0X1.6P+0`
-15.625	`ex14.4e3`	`-0X1.F400P+003`
1048580.0	`ex0.0`	`0X1.00004P+20`

For input and when the internal value is an IEEE infinity or NaN, the effect is as for `f`*w.d*.

5.31 Precision of `stat=` variables

It is recommended that any `stat=` variable should have a decimal exponent range of at least four to ensure that the error code is representable in the variable.

5.32 Deletions

5.32.1 Arithmetic `if` statement

The arithmetic `if` statement has been classed as obsolescent since Fortran 90. It is incompatible with the IEEE floating-point standard ISO/IEC/IEEE 60559:2011 because it does not distinguish between the signs of zero. It involves the use of labels, which can hinder optimization and make code hard to read and maintain. Similar logic can be more clearly encoded using other conditional statements. It is now deleted.

5.32.2 Nonblock `do construct`

Shared do termination and do termination by a statement other than `end do` or `continue` have been classed as obsolescent since Fortran 90. They offer considerable scope for confusion and unexpected errors and are now deleted. Note that labelled `do` statements are still included as an obsolescent part of the language.

5.33 New obsolescences

5.33.1 `common` and `equivalence`

The `common` and `equivalence` statements and the `block data` program unit are now all obsolescent. Common blocks are error-prone and have largely been superseded by modules. The `equivalence` statement is similarly error-prone. Whilst use of these statements was invaluable prior to Fortran 90, they are now redundant and can inhibit performance. The `block data` program unit exists only to initialize data in common blocks and hence is also redundant.

5.33.2 Labelled `do` statements

Labelled `do` statements are now an obsolescent part of the language. They are redundant with the use of a name for the construct and the use of the `cycle` statement.

5.33.3 Specific names for standard intrinsic functions

Specific names for intrinsic functions have been redundant since Fortran 90. They are now obsolescent. All the intrinsics have generic names.

5.33.4 The `forall` construct and statement

The `forall` construct and statement were added in the hope of efficient execution, but this has not happened. They are redundant with the `do concurrent` construct and the use of pointer rank remapping. They are now obsolescent.

6 Acknowledgements

I would like to express my thanks to Reinhold Bader, Erik Kruyt, Steve Lionel, Bill Long, David Muxworthy, Anton Shterenlikht, and Van Snyder for suggesting many improvements.

References

ISO/IEC (2010), 'International Standard ISO/IEC 1539-1:2010(E) Information technology - Programming languages - Fortran - Part 1: Base language', *ISO/IEC*, Geneva.

ISO/IEC (2012), 'ISO/IEC TS 29113:2012 Information technology – Further interoperability of Fortran with C', *ISO/IEC*, Geneva.

ISO/IEC (2015), 'ISO/IEC TS 18508:2015 Information technology – Additional parallel features in Fortran', *ISO/IEC*, Geneva.

ISO/IEC/IEEE (2011), ‘ISO/IEC/IEEE 60559:2011(E) Information technology – Microprocessor Systems – Floating-Point arithmetic’, *ISO/IEC*, Geneva.

Metcalf, M., Reid, J. and Cohen, M. (2011), *Modern Fortran Explained*, Oxford University Press.