Report from WG5 convener

Content of Fortran 2008

Framework was decided at last years' WG5 meeting and was not substantially changed at this year's WG5 meeting. Two large items – *bits* and *intelligent macros* – were left as 'do if time', which made the May J3 meeting important.

In the event both were deemed to be ready, but *interoperability of pointers, allocatables, assumed-shape arrays, and optional arguments* was not ready and has been deferred to a TR.

Interpretations

Corrigendum 1 has been published but is unsatisfactory because of typographical changes and will be republished.

Corrigendum 2 is ready and is going through a final WG5 vote.

Co-array discussion at the May J3 meeting

There was a J3 discussion over moving co-arrays to a TR or another part of the standard, despite the WG5 decision at the Feb. meeting that it be a ‘first priority’ item.

However, the major vendors reported pressure from users to provide co-arrays and it was decided (straw vote 6-3-2) to keep them.

A significant argument was that it would be unsatisfactory to renege on the decision of last year that co-arrays would be there.

The language features that have been chosen for Fortran 2008: co-arrays, bits, intelligent macros, ...

John Reid,

JKR Associates, UK

Abstract

Following the WG5 meeting in Fairfax in February and the J3 meeting in May, the new features for Fortran 2008 have been chosen.

This talks aims to give an overview of the new features.

BCS Fortran AGM
8 June 2006

Items seen to need over a year to develop

- UK-01 Co-arrays
- J3-47 BITS (originally TYPELESS objects)
- J3-14 Intelligent macros (supersedes parameterized modules)

Items seen to need over 6 months to develop

- J3-43 Contiguous attribute
- J3-46 `DO CONCURRENT`

Item to be developed as a TR

- J3-41/2 Interoperability of pointers, allocatables, assumed-shape arrays, and optional arguments

Minor technical changes, 1

- J3-03 Execute external program
- J3-12 Allocatable/pointer in generic resolution
- J3-13 Internal procedure as actual argument
- J3-15 Updating complex parts
- J3-16 Disassociated or deallocated actual argument associated with nonpointer nonallocatable optional dummy argument is considered not to be present
- J3-18 Non-null initial targets for pointers
- J3-19 Extend intrinsics such as `ASIN` to complex arguments
- J3-22 Allow a polymorphic allocatable variable in intrinsic assignment

Minor technical changes, 2

- J3-38 Libm: Bessel, erf, gamma, hypot
- J3-39 Rank plus co-rank limited to 15.
- J3-48 Writing Comma Separated Value CSV files
- RU-03 Obsolescent: ENTRY
- UK-05 Guarantee support of `selected_int_kind(18)`
- UK-07 Pointer function references as actual arguments
- UK-08 Pointer function references assignment contexts
- UK-11 Elemental procedures that are not pure
- UK-12 Recursive I/O to different unit

Co-arrays – basics

- SPMD – Single Program, Multiple Data
- Replicated to a number of **images**
- Number of images fixed during execution
- Each image has its own set of **local** variables
- All images start by executing main program and mostly execute asynchronously
- Variables declared as co-arrays are accessible on another image through second set of array subscripts, delimited by [ ]
- Statements: `sync_all`, `sync_team`, `sync_images`, `notify`, `query`, `sync_memory`
- Critical construct
- Collectives: `co_all`, `co_any`, etc., `form_team`
- Intrinsics: `this_image`, `num_images`, `image_index`, `co_lbound`, `co_ubound`

Examples of co-array syntax

```
real :: r[*], s[0:*], x(n)[*]
type(u) :: u2(m,n)[np,*]
! Co-arrays always have assumed
! co-size (equal to number of images)

real :: t
integer p, q, index(n)
!  Local variables
     :
t = s[p]
x(:) = x(:)[p]
! Reference without [] is to local part
x(:)[p] = x(:)
u2(i,j)%b(:) = u2(i,j)[p,q]%b(:)
```

Images have indices 1, 2, ..., `num_images()` and co-subscript lists are mapped to image indices by the usual rule.

Example: redistribution

Consider redistributing the array
`a(1:kx,1:ky)[1:kz]`
from
`b(1:ky,1:kz)[1:kx]`,
where `max(kx,kz)` ≤ `num_images()`.

```
iz = this_image(a)
if (iz<=kz) then
   do ix = 1, kx
      a(ix,:) = b(:,iz)[ix]
   end do
end if
```

The `if` construct is needed so that no action is taken on the images on which we have no data.

Implementation model

The compiler may arrange that a co-array, when originally declared, occupies the same set of addresses within each image:

- A co-array must have the same set of bounds on all images
- There is an implicit synchronization of all images at an allocate or deallocate statement so that they all perform their allocations and deallocations in the same order.

On a shared-memory machine, a co-array may be implemented as a single large array.

On any machine, a co-array may be implemented so that each image can calculate the memory address of an element on any image.

Synchronization

The images normally execute asynchronously. If one image relies on another image having taken an action, explicit synchronization is needed.

For example, to read data on image 1 and get it to other images:

```
if(this_image()==1) read(*,*)p
sync_all
p = p[1]
```

Critical sections

Exceptionally, it may be necessary to limit execution to one image at a time:

```
critical
   p[6] = p[6] + 1
   :
end critical
```

Dynamic co-arrays

Only dynamic form: the allocatable co-array.

Automatic arrays or array-valued functions would require automatic synchronization, which would be awkward.

Co-Arrays and SAVE

Unless allocatable or a dummy argument, a co-array must be given the SAVE attribute.

This is to avoid the need for synchronization when co-arrays go out of scope on return from a procedure.

Optimization

Most of the time, the compiler can optimize as if the image is on its own, using its temporary storage such as cache, registers, etc.

Structure components

Types may have allocatable co-array components, but structures of this type must be scalar.

However, a co-array may be of a derived type with allocatable or pointer components, which allows the size to vary from image to image.

Pointers must have targets in their own image:

```
q => z[i]%p       ! Not allowed
allocate(z[i]%p) ! Not allowed
```

BITS

There will be a new intrinsic type, BITS. The number of bits is specified by the kind type parameter with default `NUMERIC_STORAGE_SIZE`.

Up to `4*NUMERIC_STORAGE_SIZE` bits must be supported. Processor may support more.

Concatenation operator `//` available.

`==`, `/=` available for bits with bits, real, integer, or complex.

`>`, `>=`, `<`, `<=`, available for bits with bits, real, or integer.

`.AND.`, `.OR.`, `.XOR.`, `.EQV.`, `.NEQV.` available for bits with bits or integer.

`.NOT.` available for bits.

If the kinds differ, the shorter is padded on the left with zeros.

Assignment to bits

Assignment to bits available from bits, real, integer, or complex. If the kinds differ, digits on the left are discarded or padded with zeros. For types other than bits, the internal representation is used.

Interoperability

There are 26 C types that are interoperable with bits.

New intrinsics

BITS_KIND(X[,KIND])	Bits kind type parameter value compatible with the argument
SELECTED_BITS_KIND(N)	Bits kind type parameter value, given number of bits
BITS(A [,KIND])	Conversion to bits type
DSHIFTL (I, J, SHIFT)	Double left shift
DSHIFTR (I, J, SHIFT)	Double right shift
MERGE_BITS (I,J,MASK)	Merge bits under mask
SHIFTA (I, SHIFT)	Arithmetic right shift
SHIFTL (I, SHIFT)	Left shift
SHIFTR (I, SHIFT)	Right shift
LEADZ (I [,KIND])	Number of leading zero bits
POPCNT (I [,KIND])	Number of one bits
POPPAR (I [,KIND])	Parity of one bits
TRAILZ (I [,KIND])	Number of trailing zero bits
MASKL (I [,KIND])	Left justified bit mask
MASKR (I [,KIND])	Right justified bit mask
IALL(ARRAY,DIM[,MASK]) or IALL(ARRAY[,MASK])	Bitwise AND of array elements
IANY(ARRAY,DIM[,MASK]) or IANY(ARRAY[,MASK])	Bitwise OR of array elements
IPARITY(ARRAY,DIM,[,MASK]) or IPARITY(ARRAY[,MASK])	Bitwise exclusive OR of array elements
PARITY (MASK [,DIM])	True if an odd number of values is true

Intelligent macros

Parameterized modules were proposed to provide a facility whereby a module or subprogram can be developed in a generic form, and then applied to any appropriate type.

Malcolm pointed out that this is roughly equivalent to a built-in macro facility, but it misses out on useful things one can do with macros.

By ‘intelligent’, he means that they know about Fortran and are scoped. He wants them to be able to create modules, types, procedures, and sections of code.

The elements are the macro definition, e.g.

```
DEFINE MACRO :: single_linked_list(type)
    TYPE type%%_list
      type :: value
      TYPE(type%%_list),POINTER :: next
    END TYPE
END MACRO
```

and the later macro expansion:

```
EXPAND single_linked_list(real)
```

where the EXPAND statement is replaced by the sequence of statements

```
    TYPE real_list
      real :: value
      TYPE(real_list),POINTER :: next
    END TYPE
```

Note that type is a macro dummy argument whose scope is the macro definition. On expansion it is replaced by a token or a sequence of tokens and %% is used for concatenation during expansion.

Macro IF and DO constructs

Macros may contain IF and DO constructs that are processed during macro expansion. Here it is likely that several macro body statements may be needed to build a single statement, e.g.

```
      CALL impure_scalar_procedure &&
      (array(index%%1 &&
MACRO DO i=2,rank
      ,index%i &&
MACRO END DO
      ),traceinfo)
```

The double amphersands indicate that a single statement is being created.