NAG Fortran Compiler Release 7.1 Release Note

December 21, 2022

1 Introduction

Release 7.1 of the NAG Fortran Compiler is a major update.

Customers upgrading from a previous release of the NAG Fortran Compiler will need a new licence key for this release.

See `KLICENCE.txt` for more information about Kusari Licence Management.

2 Release Overview

Release 7.1 of the NAG Fortran Compiler supports all of Fortran 2008, in particular:

- A reference to a function that returns a data pointer can be used as a variable in many contexts, including on the left-hand side of an assignment statement.
- Other minor features, such as pointer target initialisation, allowing allocatable components to have recursive or mutually recursive type, and functions returning procedure pointers, are now supported.

Substantial additional support for Fortran 2018 has been added, in particular:

- With the addition of intrinsic "collective" subroutines (`CO_SUM` et al), all of the coarray parallel programming features of Fortran 2018 are now supported.
- Partial support for the advanced C interoperability has been added. Specifically, assumed-rank and assumed-type arguments, along with C descriptors, are supported.
- Use of assumed-rank arguments directly in Fortran (rather than C) is provided by the `RANK` intrinsic function and the `SELECT RANK` construct.
- A number of minor features, such as allowing an optional dummy argument to be used as the `DIM` argument of an intrinsic array reduction function (such as `SUM`), have been added.

Support for OpenMP programming has been improved by enabling undefined variable checking in OpenMP programs. Additionally, some minor features from newer OpenMP specifications have also been added.

Finally, there is also additional error checking available, and the bundled tools have some additional capabilities.

3 Compatibility

3.1 Compatibility with Release 7.0

Release 7.1 is compatible with Release 7.0, except that files compiled with the *−C=calls* option will need to be recompiled if they contain a procedure with a procedure pointer argument, or a reference to such a procedure.

3.2 Compatibility with Release 6.2

On MacOS the 32-bit ABI mode accessible via *−abi=32* has been removed; consequently only 64-bit compilation is supported and the *−abi=* switch has been removed entirely.

Other than this, Release 7.1 is fully compatible with Release 6.2 except when coarrays are used, or when the *−C=calls* option is used for a subroutine that has an alternate return. Any program that uses these features will need to be recompiled.

3.3 Compatibility with Release 6.1

Programs which use features from HPF (High Performance Fortran), for example the `ILEN` intrinsic function or the `HPF_LIBRARY` module, are no longer supported.

The previously deprecated *−abi=64* option on Linux x86-64 has been withdrawn. This option provided an ABI with 64-bit pointers but 32-bit object sizes and subscript arithmetic, and was only present for compatibility with Release 5.1 and earlier.

With the exception of HPF support and the deprecated option removal, Release 7.1 of the NAG Fortran Compiler is fully compatible with Release 6.1.

3.4 Compatibility with Release 6.0

With the exception of HPF support and the deprecated option removal, Release 7.1 of the NAG Fortran Compiler is compatible with Release 6.0 except that programs that use allocatable arrays of "Parameterised Derived Type" will need to be recompiled (this only affects module variables and dummy arguments).

3.5 Compatibility with Releases 5.3.1, 5.3 and 5.2

With the exception of HPF support and the deprecated option removal, Release 7.1 of the NAG Fortran Compiler is fully compatible with Release 5.3.1. It is also fully compatible with Releases 5.3 and 5.2, except that on Windows, modules or procedures whose names begin with a dollar sign ($) need to be recompiled.

For a program that uses the new "Parameterised Derived Types" feature, it is strongly recommended that all parts of the program that may allocate, deallocate, initialise or copy a polymorphic variable whose dynamic type might be a parameterised derived type, should be compiled with Release 7.1.

3.6 Compatibility with Release 5.1

Release 7.1 of the NAG Fortran Compiler is compatible with NAGWare f95 Release 5.1 except that:

- programs that use features from HPF are not supported;
- programs or libraries that use the `CLASS` keyword, or which contain types that will be extended, need to be recompiled;
- 64-bit programs and libraries compiled with Release 5.1 on Linux x86-64 (product NPL6A51NA) are binary incompatible, and need to be recompiled.

4 New Fortran 2008 Features

- An allocatable component can forward-reference a type, for example:

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

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

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

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

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

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

  For example, in

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

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

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

  Named pointer initialisation

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

  For example,

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

  Component default initialisation

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

  For example,

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

  Component initialisation with structure constructors

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

  For example,

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

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

  For example, with this module,

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

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

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

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

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

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

  Finally, here is an example using argument passing.

```
Program argument_eg
  Use m
  Call set(f(7))
  Print 1,table(7) ! Will print "1.41421"
```

```
1 Format(F7.5)
Contains
  Subroutine set(x)
    Real,Intent(Out) :: x
    x = Sqrt(2.0)
  End Subroutine
End Program
```

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

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

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

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

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

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

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

5 New Fortran 2018 Features

- The intrinsic subroutines CO_BROADCAST, CO_MAX, CO_MIN, CO_REDUCE and CO_SUM perform **collective** operations. These are for coarray parallelism: they compute values across all images in the current team, without explicit synchronisation.

  All of these subroutines have optional STAT and ERRMSG arguments. On successful execution, the STAT argument is assigned the value zero and the ERRMSG argument is left unchanged. If an error occurs, a positive value is assigned to STAT and an explanatory message is assigned to ERRMSG. Only the errors STAT_FAILED_IMAGE and STAT_STOPPED_IMAGE are likely to be able to be caught in this way. Because there is not full synchronisation (see below), different images may receive different errors, or none at all. If an error occurs and STAT is not present, execution is terminated. Note that if the actual arguments for STAT or ERRMSG are optional dummy arguments, they must be present on all images or absent on all images.

  A reference (CALL) to one of these subroutines is **not** an image control statement, does not end the current segment, and does not imply synchronisation (though some partial synchronisation will occur during the computation). However, such calls are only permitted where an image control statement is permitted.

  Each image in a team must execute the same sequence of CALL statements to collective subroutines as the other images in the team. There must be no synchronisation between the images at the time of the call; the invocations must come from unordered segments.

  All collective subroutines have the first argument "A", which is INTENT(INOUT), and must not be a coindexed object. This argument contains the data for the calculation, and must have the same type, type parameters, and shape on all images in the current team. If it is a coarray that is a dummy argument, it must have the same ultimate argument on all images.

  SUBROUTINE CO_BROADCAST (A, SOURCE_IMAGE *[*, STAT, ERRMSG *]*)

  > A : variable of any type; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;
  >
  > SOURCE_IMAGE : integer scalar, in the range one to NUM_IMAGES(), this argument must have the same value on all images in the current team;
  >
  > STAT (optional) : integer scalar variable, not coindexed;
  >
  > ERRMSG (optional) : character scalar variable of default kind, not coindexed.
  >
  > The value of argument A on image SOURCE_IMAGE is assigned to the argument A on all the other images.

  SUBROUTINE CO_MAX (A *[*, RESULT_IMAGE, STAT, ERRMSG *]*)

  > A: variable of type Integer, Real, or Character; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if A is a coarray that is a dummy argument, it must have the same ultimate argument on each image;
  >
  > RESULT_IMAGE (optional) : integer scalar, in the range one to NUM_IMAGES(), this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;
  >
  > STAT (optional) : integer scalar variable, not coindexed;
  >
  > ERRMSG (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes the maximum value of `A` across all images; if `A` is an array, the value is computed elementally. If `RESULT_IMAGE` is present, the result is assigned to argument `A` on that image, otherwise it is assigned to argument `A` on all images.

`SUBROUTINE CO_MIN ( A [, RESULT_IMAGE, STAT, ERRMSG ] )`

`A`: variable of type Integer, Real, or Character; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if `A` is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

`RESULT_IMAGE` (optional) : integer scalar, in the range one to `NUM_IMAGES()`, this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;

`STAT` (optional) : integer scalar variable, not coindexed;

`ERRMSG` (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes the minimum value of `A` across all images; if `A` is an array, the value is computed elementally. If `RESULT_IMAGE` is present, the result is assigned to argument `A` on that image, otherwise it is assigned to argument `A` on all images.

`SUBROUTINE CO_REDUCE ( A, OPERATION [, RESULT_IMAGE, STAT, ERRMSG ] )`

`A`: non-polymorphic variable of any type; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if `A` is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

`OPERATION` : pure function with exactly two arguments; the dummy arguments of `OPERATION` must be non-allocatable, non-optional, non-pointer, non-polymorphic dummy variables, and each argument and the result of the function must be scalar with the same type and type parameters as `A`;

`RESULT_IMAGE` (optional) : integer scalar, in the range one to `NUM_IMAGES()`, this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;

`STAT` (optional) : integer scalar variable, not coindexed;

`ERRMSG` (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes an arbitrary reduction of `A` across all images; if `A` is an array, the value is computed elementally. The reduction is computed starting with the set of corresponding values of `A` on all images; this is an iterative process, taking two values from the set and converting them to a single value by applying the `OPERATION` function; the process continues until the set contains only a single value — that value is the result. If `RESULT_IMAGE` is present, the result is assigned to argument `A` on that image, otherwise it is assigned to argument `A` on all images.

`SUBROUTINE CO_SUM ( A [, RESULT_IMAGE, STAT, ERRMSG ] )`

`A`: variable of type Integer, Real, or Complex; it must not be a coindexed object, and must have the same type, type parameters and shape on all images in the current team; if `A` is a coarray that is a dummy argument, it must have the same ultimate argument on each image;

`RESULT_IMAGE` (optional) : integer scalar, in the range one to `NUM_IMAGES()`, this argument must be either present on all images or absent on all images; if present, it must have the same value on all images in the current team;

`STAT` (optional) : integer scalar variable, not coindexed;

`ERRMSG` (optional) : character scalar variable of default kind, not coindexed.

This subroutine computes the sum of `A` across all images; if `A` is an array, the value is computed elementally. If `RESULT_IMAGE` is present, the result is assigned to argument `A` on that image, otherwise it is assigned to argument `A` on all images.

- The DIM argument to the intrinsic functions ALL, ANY, FINDLOC, IALL, IANY, IPARITY, MAXLOC, MAXVAL, MINLOC, MINVAL, NORM2, PARITY, PRODUCT and SUM can be an optional dummy argument, as long as it is present at execution time. For example,

```
Subroutine sub(x,n)
  Real,Intent(In) :: x(:,:,:)
  Integer,Intent(In),Optional :: n
  If (Present(n)) Then
    Print *,Norm2(x,n) ! Rank two array result.
  Else
    Print *,Norm2(x)   ! Scalar result.
  End If
End Subroutine
```

- The SIZE= specifier can be used in a READ statement without ADVANCE='NO', that is, in a READ statement with no ADVANCE= specifier, or one with an explicit ADVANCE='YES'. For example,

```
Character(65536) buf
Integer nc
Read(*,'(A)',Size=nc) buf
Print *,'The number of characters on that line was',nc
```

  Note that SIZE= is not permitted with list-directed or namelist formatting; that would be pointless, as there are no edit descriptors with such formatting and thus no characters to be counted by SIZE=.

- Specific intrinsic functions are reported as obsolescent with the *–f2018* option. In the case of a function that is both specific and generic, e.g. SQRT, only obsolescent usage is reported, e.g. passing as an actual argument, use as a procedure interface, or being the target of a procedure pointer assignment.

  For example, in

```
Program obsolete_cos_usage
  Real x
  Intrinsic cos
  Procedure(cos),Pointer :: pp    ! Obsolescent
  x = 1.5
  pp => cos                       ! Obsolescent
  Call other_procedure(cos)       ! Obsolescent
  Print *,cos(x),pp(x)
End Program
Subroutine other_procedure(f)
  Real,External :: f
  Print *,f(1.5)
End Subroutine
```

  only the three lines marked with the comment will produce warning messages.

- Assumed-rank dummy arguments accept actual arguments of any rank; they **assume** the rank from the actual argument. This rank may be zero; that is, the actual argument may be scalar. Furthermore, assumed-rank dummy arguments may have the ALLOCATABLE or POINTER attribute, and thus accept allocatable/pointer variables of any rank.

  The syntax is as follows:

```
Real,Dimension(..) :: a, b
Integer :: c(..)
```

  That declares three variables (which must be dummy arguments) to be assumed-rank.

  The use of assumed-rank dummy arguments within Fortran is extremely limited; basically, the intrinsic inquiry functions can be used, and there is a SELECT RANK construct, but other than that they may only appear as actual arguments to other procedures where they correspond to another assumed-rank argument.

The main use of assumed rank is for advanced C interoperability. An assumed-rank argument is passed by reference as a "C descriptor"; it is then up to the C routine to decode what that means. The C descriptor, along with several utility functions for manipulating it, is defined by the source file `ISO_Fortran_binding.h`; this can be found in the compiler's library directory (on Linux this is usually `/usr/local/lib/NAG_Fortran`, but that can be changed at installation time).

This topic is highly complex, and beyond the scope of a Release Note. The reader should direct their attention to the Fortran 2018 standard, or to a good textbook such as "Modern Fortran Explained" by Metcalf, Reid and Cohen.

Here is an extremely simple example:

```
Program assumed_rank_example
  Real x(1,2),y(3,4,5,6,7)
  Call showrank(1.5)
  Call showrank(x)
  Call showrank(y)
Contains
  Subroutine showrank(a)
    Real,Intent(In) :: a(..)
    Print *,'Rank is',Rank(a)
  End Subroutine
End Program
```

That will produce the output

```
 Rank is 0
 Rank is 2
 Rank is 5
```

- The `SELECT RANK` construct facilitates use of assumed rank objects in Fortran. It has the syntax

```
[ construct-name ] SELECT RANK ( [ assoc_name => ] assumed-rank-variable-name )
      [ rank-stmt
          block ]...
END SELECT [ construct-name ]
```

where *rank-stmt* is one of:

```
RANK ( scalar-int-constant-expression ) [ construct-name ]
RANK ( * ) [ construct-name ]
RANK DEFAULT [ construct-name ]
```

In any particular `SELECT RANK` construct, there must not be more than one `RANK DEFAULT` statement, or more than one `RANK (*)` statement, or more than `RANK (`*integer*`)` with the same value integer expression. If the assumed-rank variable has the `ALLOCATABLE` or `POINTER` attribute, the `RANK (*)` statement is not permitted.

The *block* following a `RANK` statement with an integer constant expression is executed if the assumed-rank variable is associated with a non-assumed-rank actual argument that has that rank, and is not an assumed-size array. Within the *block* it acts as if it were an assumed-shape array with that rank.

The *block* following a `RANK (*)` is executed if the ultimate argument is an assumed-size array. Within the *block* it acts as if it were declared with bounds '`(1:*)`'; if different bounds or rank are desired, this can be passed to another procedure using sequence association.

The *block* following a `RANK DEFAULT` statement is executed if no other block is selected. Within its *block*, it is still an assumed-rank variable, i.e. there is no change.

Here is a simple example of the `SELECT RANK` construct.

```
Program select_rank_example
  Integer :: a = 123, b(1,2) = Reshape( [ 10,20 ], [ 1,2 ] ), c(1,3,1) = 777, d(1,1,1,1,1)
  Call show(a)
  Call show(b)
```

```
  Call show(c)
  Call show(d)
Contains
  Subroutine show(x)
    Integer x(..)
    Select Rank(x)
    Rank (0)
      Print 1,'scalar',x
    Rank (1)
      Print 1,'vector',x
    Rank (2)
      Print 1,'matrix',x
    Rank (3)
      Print 1,'3D array',x
    Rank Default
      Print *,'Rank',Rank(x),'not supported'
    End Select
  1 Format(1x,a,*(1x,i0,:))
  End Subroutine
End Program
```

This will produce the output

```
scalar 123
matrix 10 20
3D array 777 777 777
Rank 5 not supported
```

- The `TYPE(*)` type specifier can be used to declare scalar, assumed-size, and assumed-rank dummy arguments. Such an argument is called **assumed-type**; the corresponding actual argument may be of any type. It must not have the `ALLOCATABLE`, `CODIMENSION`, `INTENT (OUT)`, `POINTER`, or `VALUE` attribute.

  An assumed-type variable is extremely limited in the ways it can be used directly in Fortran:

  - it may be passed as an actual argument to another assumed-type dummy argument;
  - it may appear as the first argument to the intrinsic functions `IS_CONTIGUOUS`, `LBOUND`, `PRESENT`, `SHAPE`, `SIZE`, or `UBOUND`;
  - it may be used as the argument of the function `C_LOC` (in the `ISO_C_BINDING` intrinsic module}.

  Other than these contexts, it cannot be used in any other way at all. Note that if it is an array, you cannot subscript it or create an array section from it.

  This is mostly useful for interoperating with C programs. A `TYPE(*)` dummy argument interoperates with a C argument declared as "`void *`". There is no difference between scalar and assumed-size on the C side, but on the Fortran side, if the dummy argument is scalar the actual argument must also be scalar, and if the dummy argument is an array, the actual argument must also be an array.

  Because an actual argument can be passed directly to a `TYPE(*)` dummy, the `C_LOC` function is not required, and so there is no need for the `TARGET` attribute on the actual argument.

  For example,

```
Program type_star_example
  Interface
    Function checksum(scalar,size) Bind(C)
      Use Iso_C_Binding
      Type(*) scalar
      Integer(C_int),Value :: size
      Integer(C_int) checksum
    End Function
  End Interface
  Type myvec3
```

```
    Double Precision v(3)
  End Type
  Type(myvec3) x
  Call Random_Number(x%v)
  Print *,checksum(x,Storage_Size(x)/8)
End Program
int checksum(void *a,int n)
{
  int i;
  int res = 0;
  unsigned char *p = a;
  for (i=0; i<n; i++) res = 0x3fffffff&((res<<1) + p[i]);
  return res;
}
```

- A `BIND(C)` procedure can have optional arguments. Such arguments cannot also have the `VALUE` attribute.
  An absent optional argument of a `BIND(C)` procedure is indicated by passing a null pointer argument.
  For example,

```
Program optional_example
  Use Iso_C_Binding
  Interface
    Function f(a,b) Bind(C)
      Import
      Integer(C_int),Intent(In) :: a
      Integer(C_int),Intent(In),Optional :: b
      Integer(C_int) f
    End Function
  End Interface
  Integer(C_int) x,y
  x = f(3,14)
  y = f(23)
  Print *,x,y
End Program

int f(int *arg1,int *arg2)
{
  int res = *arg1;
  if (arg2) res += *arg2;
  return res;
}
```

  The second reference to `f` is missing the optional argument `b`, so a null pointer will be passed for it. This will result in the output:

```
 17 23
```

- The intrinsic inquiry function `RANK` returns the dimensionality of its argument. It has the following syntax:

  `RANK ( A )`

  `A` : data object of any type:

  Result : scalar Integer of default kind.

  The result is the rank of `A`, that is, zero for scalar `A`, one if `A` is a one-dimensional array, and so on.
  This function can be used in a constant expression except when `A` is an assumed-rank variable.

- The intrinsic function REDUCE performs user-defined array reductions. It has the following syntax:

```
REDUCE ( ARRAY, OPERATION [, MASK, IDENTITY, ORDERED ] ) or
REDUCE ( ARRAY, OPERATION DIM [, MASK, IDENTITY, ORDERED ] )
```

  ARRAY : array of any type;

  OPERATION : pure function with two arguments, each argument being scalar, non-allocatable, non-pointer, non-polymorphic non-optional variables with the same declared type and type parameters as ARRAY; if one argument has the ASYNCHRONOUS, TARGET or VALUE attribute, the other must also have that attribute; the result must be a non-polymorphic scalar variable with the same type and type parameters as ARRAY;

  DIM : scalar Integer in the range 1 to N, where N is the rank of ARRAY;

  MASK : type Logical, and either scalar or an array with the same shape as ARRAY;

  IDENTITY : scalar with the same declared type and type parameters as ARRAY;

  ORDERED : scalar of type Logical;

  Result : Same type and type parameters as ARRAY.

  The result is ARRAY reduced by the user-supplied OPERATION. If DIM is absent, the whole (masked) ARRAY is reduced to a scalar result. If DIM is present, the result has rank N-1 and the shape of ARRAY with dimension DIM removed; each element of the result is the reduction of the masked elements in that dimension.

  If exactly one element contributes to a result value, that value is equal to the element; that is, OPERATION is only invoked when more that one element appears.

  If no elements contribute to a result value, the IDENTITY argument must be present, and that value is equal to IDENTITY.

  For example,

```
Module triplet_m
  Type triplet
    Integer i,j,k
  End Type
Contains
  Pure Type(triplet) Function tadd(a,b)
    Type(triplet),Intent(In) :: a,b
    tadd%i = a%i + b%i
    tadd%j = a%j + b%j
    tadd%k = a%k + b%k
  End Function
End Module
Program reduce_example
  Use triplet_m
  Type(triplet) a(2,3)
  a = Reshape( [ triplet(1,2,3),triplet(1,2,4), &
                 triplet(2,2,5),triplet(2,2,6), &
                 triplet(3,2,7),triplet(3,2,8) ], [ 2,3 ] )
  Print 1, Reduce(a,tadd)
  Print 1, Reduce(a,tadd,1)
  Print 1, Reduce(a,tadd,a%i/=2)
  Print 1, Reduce(Array=a,Dim=2,Operation=tadd)
  Print 1, Reduce(a, Mask=a%i/=2, Dim=1, Operation=tadd, Identity=triplet(0,0,0))
1 Format(1x,6('triplet(',I0,',',I0,',',I0,')',:,'; '))
End Program
```

This will produce the output:

```
 triplet(12,12,33)
 triplet(2,4,7); triplet(4,4,11); triplet(6,4,15)
 triplet(8,8,22)
 triplet(6,6,15); triplet(6,6,18)
 triplet(2,4,7); triplet(0,0,0); triplet(6,4,15)
```

- Generic resolution can use the number of procedure arguments; that is, if one procedure has more non-optional procedure arguments than the other has optional plus non-optional procedure arguments, the procedures are considered to be unambiguous.

  For example,

```
Module npa_example
  Interface g
    Module Procedure s1,s2
  End Interface
Contains
  Subroutine s1(a)
    External a
    Call a
  End Subroutine
  Subroutine s2(b,a)
    External b,a
    Call b
    Call a
  End Subroutine
End Module
```

  This example does not conform to the Fortran 2008 rules for unambiguous generic procedures, because the argument `A` distinguishes by position but not by keyword, the argument `B` distinguish by keyword but not by position, and the positional disambiguator (`A`) does not appear earlier in the list than the keyword disambiguator (`B`).

6 Additional OpenMP support

- Undefined variable detection, with the *−C=undefined* option, is supported. For example, executing the program:

```
Program bad
  Use omp_lib
  Real x,y(10)
  x = 3
  !$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(y)
    Do i=1,Size(y)
      y(omp_get_thread_num() + 1) = x * omp_get_thread_num()
    End Do
  !$OMP END PARALLEL DO
  Print *,y
End Program
```

  produces the output

```
Runtime Error: bad.f90, line 7: Reference to undefined variable X
Program terminated by fatal error
```

- The OpenMP 4.0 `CANCEL` and `CANCELLATION POINT` directives are supported, along with the `OMP_CANCELLATION` environment variable and the `OMP_GET_CANCELLATION` function. Also, the OpenMP 5.0 `CANCEL:` keyword on the `IF` clause of `CANCEL` is supported.

  The syntax of the `CANCEL` directive is:

```
!$OMP CANCEL construct-type [ [,] IF ( [ CANCEL: ] scalar-logical-expr ) ]
```

where *construct-type* is one of the keywords PARALLEL, DO, SECTIONS or TASKGROUP. The directive must be closely nested in an OpenMP construct of the type specified, except for TASKGROUP where it must be closely nested in an OpenMP TASK construct. In the case of SECTIONS, the construct must not have a NOWAIT clause. In the case of DO, the construct must not have a NOWAIT clause or an ORDERED clause.

Executing a CANCEL directive evaluates the IF clause if present, and if the IF clause is not present, or evaluates to true, and cancellation is enabled, the construct is **cancelled**. If cancelled, the executing thread jumps immediately to the end of the construct.

The syntax of the CANCELLATION POINT directive is:

```
!$OMP CANCELLATION POINT construct-type
```

where *construct-type* is the same as for the CANCEL directive. As for CANCEL, the CANCELLATION POINT directive must be closely nested in the appropriate OpenMP construct. That construct should also have a CANCEL directive, otherwise there can never be any effect (other than waste CPU time) from the CANCELLATION POINT directive.

Executing a CANCELLATION POINT directive will transfer control to the end of the construct if the construct has been cancelled; otherwise, there is no effect.

Apart from the CANCELLATION POINT directive, there is also a cancellation point at a CANCEL directive (even if it has an IF clause that evaluates to false), a BARRIER directive, or an implied barrier (e.g. at the end of a contained construct such as END SINGLE without NOWAIT).

After a construct has been cancelled, any barrier that is executed within the region must be closely nested inside the cancelled directive (i.e. not in a procedure called from the construct), otherwise there will either be a runtime error (with the NAG compiler) or an infinite hang (as the cancelling thread will never reach the barrier).

Cancellation is enabled if the OMP_CANCELLATION environment variable has the value 'TRUE' (not case-sensitive), and is disabled if the variable has the value 'FALSE'. If the variable does not exist, the setting is compiler-dependent; with the NAG Fortran Compiler, the default is FALSE.

The OMP_GET_CANCELLATION function has the interface

```
LOGICAL FUNCTION OMP_GET_CANCELLATION()
END FUNCTION
```

It returns .TRUE. if cancellation is enabled, and .FALSE. if disabled.

Here is a simple example of cancellation.

```
  Program example
    Real a(20000,4),b(4)
    Logical ok
    Call Random_Number(a)
    a = a*10
    Call normalise(a,b,ok)
    Print 1,ok,b
    If (Any(a>2)) Print *,'Cancellation occurred'
    Call Random_Number(a)
    a = a*10
    a(:,2) = -a(:,2)
    Call normalise(a,b,ok)
    Print 1,ok,b
1   Format(1X,'Succeeded = ',L1,', vmax =',4F8.4)
    If (Any(a>2)) Print *,'Cancellation occurred'
  Contains
    Subroutine normalise(x,y,succeeded)
      Use omp_lib
      Real,Intent(InOut) :: x(:,:)
      Real,Intent(Out) :: y(Size(x,2))
      Logical,Intent(Out) :: succeeded
      Real z
```

```
    Integer me,i
    succeeded = .True.
    !$OMP PARALLEL PRIVATE(me,i,z) SHARED(x,y) NUM_THREADS(Size(x,2))
      z = 0
      me = omp_get_thread_num() + 1
      Do i=1,Size(x,1)
        If (x(i,me)>z) z = x(i,me)
      End Do
      y(me) = z
      !$OMP CANCEL PARALLEL, IF (z==0)
      z = 2.0/z
      Do i=1,Size(x,1)
        x(i,me) = x(i,me)*z
        If (Iand(i,4095)==0) Then
          ! Every 4096 elements, check to see if the whole thing was cancelled already.
          !$OMP CANCELLATION POINT PARALLEL
        End If
      End Do
    !$OMP END PARALLEL
    succeeded = All(y/=0)
  End Subroutine
End Program
```

The output from this example could look something like

```
Succeeded = T, vmax =  9.9993  9.9997  9.9998  9.9974
Succeeded = F, vmax =  9.9999  0.0000  9.9996  9.9998
Cancellation occurred
```

- The OpenMP 5.1 `MASKED` construct is supported. This has the form:

  ```
  !$OMP MASKED [ FILTER( scalar-integer-expression ) ]
    structured-block
  !$OMP END MASKED
  ```

  A thread executes the *structured-block* if and only if the *scalar-integer-expression* evaluates equal to the thread number. Thread numbers start at zero, which is the number of the primary thread of a `PARALLEL` region. If the `FILTER` clause is not present, it acts as if it were present as `FILTER(0)`; that is, only the primary thread will execute the structured block.

  This is exactly equivalent to

  ```
  IF (omp_get_thread_num()==scalar-integer-expression) THEN
    structured-block
  END IF
  ```

  Note that the integer expression need not have the same value on each thread, and if that is the case, more than one thread may execute the masked region concurrently — there is no locking or synchronisation implied by this construct. That means that if any variables that are not local to the thread are updated, the necessary locking will need to be explicitly inserted by the programmer.

7 Additional error checking

- Multiple occurrences of the same variable in an `ALLOCATE` or `DEALLOCATE` statement is now detected as an error.
- Checking of type-bound procedure overriding and global consistency now reports differences in dummy argument shape.
- Integer arguments to `SYSTEM_CLOCK` that are 8-bit or 16-bit are now detected as errors, as they are too small to receive the values.

- Warnings are produced for argument inconsistencies for `SYSTEM_CLOCK` when
  - there are no arguments;
  - integer arguments have different kind (this will be invalid in Fortran 202x);
  - mixed integer and real arguments;
  - integer arguments are not 64-bit integer as recommended by Fortran 202x.
- An error is produced for use of a `NAMELIST` with a variable that has an allocatable, pointer, or inaccessible component, if the variable will not be processed by defined input/output.
- A warning is produced if a format specification that is a constant character string has non-blank characters after the closing parenthesis, for example,

  ```
  Print '(1X,I0) whatever',13
  ```

  will produce a warning like

  ```
  Questionable: file.f90, line 2: Extraneous nonblank characters "whatever" after right
                                  parenthesis in character string format specification
  ```

- A `CONTINUE` statement does not have any effect, but may be a branch target statement, or the end of a `DO` loop, if it has a label. A `CONTINUE` statement with no label however, cannot be of use in any way. Therefore we produce a `Note` level warning. If the `CONTINUE` statement is within a `DO` loop, this is upgraded to `Questionable`, as a C programmer might think it has the effect of the `CYCLE` statement (as it does in C). For example,

  ```
  Program continues
    Continue
    Do i=1,10
      Print *,i**2
      Continue
    End Do
  End Program
  ```

  will produce the warnings

  ```
  Note: file.f90, line 2: CONTINUE statement with no label
  Questionable: file.f90, line 5: CONTINUE statement with no label inside DO loop - did you
                                  mean CYCLE?
  ```

- When allocating coarrays that are of parameterised derived type, a difference in the length type parameter values on different images is now detected as a runtime error. For example,

  ```
  Allocate(w[*],Mold=z)
  ```

  where `Z` is a dummy argument with different type parameter values on different images, may produce a runtime error like

  ```
  Runtime Error: pco068.f90, line 17: Type parameter K1 in coarray allocation has value 1 on
                                      image one but 13 on image 2
  ```

- Empty `SELECT CASE` and `SELECT TYPE` constructs are reported as Questionable, as they do not do anything other than evaluate the selector.

  For example, if the file `sub.f90` contains:

  ```
  Subroutine sub(x,n)
    Class(*) x
    Integer n
    Select Case (n)
    End Select
    Select Type (x)
    End Select
  End Subroutine
  ```

the following warnings will be produced:

```
Questionable: sub.f90, line 4: Empty SELECT CASE construct
Questionable: sub.f90, line 6: Empty SELECT TYPE construct
```

Also, evaluation of the selector may not happen if optimisation (the $-O$ option) is used.

- The $-C$=*intovf* option will now detect integer overflow in the `SHAPE` and `SIZE` intrinsic functions. For example,

```
Program shape_overflow_example
  Real a(1,123456,3)
  Call sub(a)
Contains
  Subroutine sub(x)
    Use Iso_Fortran_Env
    Real,Intent(In) :: x(:,:,:)
    Integer(int16) sh(3)
    sh = Shape(x,int16)
    Print *,sh
  End Subroutine
End Program
```

If compiled with $-C$=*intovf*, instead of producing the nonsense results

```
 1 -7616 3
```

it will produce the runtime error message:

```
Runtime Error: shape_overflow_example.f90, line 9: INTEGER(int16) overflow for intrinsic
 SHAPE, true result value is 123456
```

- The $-C$=*calls* option now treats dummy arguments that are procedure pointers as being different from functions returning data pointers, and from functions returning procedure pointers (at any level of procedure call). These errors can only arise if a procedure is invoked with an incorrect `INTERFACE` block specification.

  For example, if the procedure

```
Subroutine sub(f,x)
  Interface
    Function f(y)
      Real,Pointer :: f
    End Function
  End Interface
  Real,Intent(In) :: x
  Real,Pointer :: p
  p => f(x)
  Print *,Associated(p)
End Subroutine
```

  is invoked (via an incorrect interface block) as follows

```
Program bad
  Interface
    Subroutine sub(f,x)
      Real,External,Pointer :: f
      Real,Intent(In) :: x
    End Subroutine
  End Interface
  Real,External,Pointer :: pp
  Intrinsic sqrt
  pp => sqrt
  Call sub(pp,1.5)
End Program
```

the *−C=calls* option (with *−gline* to get a traceback) will produce the runtime error message

```
Runtime Error: sub.f90, line 1: Incorrect interface block for SUB - Dummy argument F
  (number 1) is not a procedure pointer
Program terminated by fatal error
sub.f90, line 1: Error occurred in SUB
bad.f90, line 11: Called by BAD
```

instead of a segmentation fault.

8 Miscellaneous enhancements

- The polish options *−idcase=* and *−kwcase=* now have an extra possibility: 'Camel_Case'. This differs from 'Capitalised' only in that a letter after an underscore will also be uppercase, not just the initial letter. For *−kwcase=Camel_Case*, the keywords affected are `Non_Intrinsic`, `Non_Overridable` and `Non_Recursive`, and the OpenMP keyword `Num_Threads`. Note that case specifications are not case-sensitive, and can be abbreviated to the first letter, except for Camel_Case which can be abbreviated to 'cam'.

- The polish option *−idcase=* now has an extra possibility: 'Asis'. This preserves the case of all identifiers, including user-defined operators (but not intrinsic operators), as they appear in the input. For example, the input "`eX = Ex + 1`" will produce exactly the same inconsistent casing in the output. This option is not available with enhanced polish or other tools.

- The enhanced polish option *−case:* allows different case settings for different kinds of name. The colon is followed by a comma-separated list of "*kind=case*", where *case* is a case specification (`UPPERCASE`, `lowercase`, `Capitalised`, `Camel_Case`), and *kind* is one of the categories listed below:

`comp`	Component
`constr`	Construct name
`intr`	Intrinsic procedure
`param`	PARAMETER
`proc`	Procedure
`tbp`	Type-bound procedure
`tparam`	Derived type parameter
`type`	Derived type
`var`	Variable

  For example, *−case:var=lower,proc=u* specifies lowercase for variables and UPPERCASE for procedures. If there is no setting for a particular kind of name, it will fall back to an appropriate category; `param`, `type`, `comp`, `tparam` and `proc` all fall back to `var`, `intr` will fall back to `proc`, and `tbp` will fall back to `comp` or `proc`. If there is no rule or fall-back rule, the *−idcase=* option setting (or default) is used.

- The enhanced polish *−casex:* specifies exceptions to the case rules. The colon is followed by a comma-separated list of names in the exact case required. For example, *−casex:MaxVal,XYz* will result in every occurence of a name equivalent to `maxval` or `xyz` appearing as `MaxVal` or `XYz` respectively.

- The *−quiet* option suppresses the compiler (or tool) banner message, and also the summary line at the end, so that only diagnostic messages will appear.

- Extension messages now provide information on what kind of extension it is. This is either the edition of the Fortran standard that added the feature (e.g. "`Extension(F2018)`"), whether it is a NAG extension ("`Extension(NAG)`"), or whether it is an obsolete extension, e.g. to FORTRAN77, that has been superseded by standard features ("`Non-standard(Obsolete)`").

- Informational messages have been split into three levels: Note (highest), Info, and Remark (lowest). By default, Note messages are produced; this can be suppressed with the *−w=note* option. The *−info* option causes both Info and Remark messages to appear.