Public Review Comment #6 by Jean Vezina

To: J3
From: Craig Dedo
Date: November 11, 2002
Subject: Public Review Comment #6 by Jean Vezina

COMMENTS ON THE DRAFT FORTRAN 2000 STANDARD (ISO/IEC 1539-1) BY JEAN VEZINA

Mail Address:
Jean Vezina
6292 Villanelle
St-Leonard, Quebec
Canada
H1S 1W1
e-mail addresses:
jvezina@attglobal.net
Jean.Vezina@tdam.com

INTRODUCTION

The new proposed Fortran 2000 standard enhances the language considerably by introducing a number of modern new features such as object-oriented programming facilities and numerical exception handling. However, there are a number of features, some of these relatively minor, that we would like to be introduced immediately in the language. These features have been frequently requested in the various discussions groups about Fortran in the past and we feel that they will be beneficial to the Fortran community. Specifically, we ask for a few changes to the draft standard and three new features:

The changes to the draft standard we request is to rename the **NONKIND** attribute to **EXTENT** in parameterized derived type declarations and a few corrections to the C interoperability feature.

The new features we propose are:
1. Unsigned integers.
2. A **CONVERSION EXPLICIT** statement, to disallow mixing types and kinds in intrinsic numeric expressions, assignments and initializations.
3. The **FIRSTLOC** and **LASTLOC** array transformational functions

The following pages describe our requested changes and a proposed implementation of the three suggested features.

If, because of time or resource constraints, these features cannot be included in the next Fortran standard, then we would like that these be included in the repository of requirements for the successor of Fortran 2000.

1- NONKIND should be renamed to EXTENT

In parameterized derived type declarations, there are currently two categories of parameter specification “arguments”: KIND to specify the kind of the declared variables in the derived type definition and NONKIND for array bounds and character string lengths. The word NONKIND is somewhat restrictive as it already impairs the possibility of adding additional parameter attributes in a future revision of the standard. This is because NONKIND means “anything that is not a kind”. Adding future type parameter attributes (such as TYPE for genericity) will be problematic as they will be also “nonkinds”. We therefore suggest replacing the NONKIND keyword by its current meaning: EXTENT. Then any future improvements to parameterized derived types will be not hampered by the choice of an inadequate keyword.

J3 Response: We agree and we will make this change in the Fortran 2000 Committee Draft.

2- Comments on the C interoperability feature

We have noticed one error and one serious omission.

The error is in the C function prototype given at **Note 15.22** at page 390.

The prototype in the note is:

```
short func(int i; double *j; int *k; int l[10]; void *m)
```

This is not valid C. The semicolons must be replaced by commas such as:

```
short func(int i, double *j, int *k, int l[10], void *m)
```

J3 Response: Thank you for pointing out this error. We will correct it.

The omission is the lack of a SIZEOF function or its equivalent. A number of C routines related to I/O or communication APIs have a size parameter that it is traditionally computed by using the C library function sizeof. Many Fortran compiler vendors that provide extensions to access C routines also supply a nonstandard sizeof function to allow the size of items in a C sense to be computed. Otherwise, workarounds such as using the IOLENGTH keyword of INQUIRE or SIZE(TRANSFER … will be required. These workarounds are not guaranteed to give the correct results.

We therefore suggest the addition of a C_SIZEOF function to the ISO_C_BINDING module. The C interoperability feature will then standardize a feature that is already implemented in a number of extended Fortran 95 compilers.

Note: Some commenters suggested that the sizeof function (or its equivalent) should be added to the Fortran 2000 intrinsic function list rather than to the ISO_C_BINDING module. This is a good alternative.

J3 Response: Thank you for this idea. This feature was on the proposed list of features for Fortran 2000, but it did not receive enough support to make it into this edition of Fortran. We will suggest that it be added to the list of features for consideration for the next revision of the Fortran standard.

3- Unsigned integers

Justification for requirement:

In a number of scientific and engineering applications, the data are represented using unsigned integers. For example, in image processing applications, pixels are represented by integer quantities varying from 0 to n. Also, in many signal processing applications, the signal levels are digitized into discrete values varying again from 0 to n. These data are therefore stored using an unsigned integer format. Some data encryption and pseudo-random number generation algorithms are also more naturally expressed using unsigned integers. In addition, the C interoperability facility introduced in F2K will cause Fortran programs to interact with C procedures that may expect unsigned integer arguments. Finally, many “industry standard” file formats such as TIFF include unsigned integers as one of their possible data representations. The unformatted stream I/O facility introduced in F2K will finally allow Fortran programs to access portably these kinds of files, requiring programs to process data stored in unsigned integer format.

The current methods of dealing with unsigned integers in Fortran are awkward and not portable. We can list three workarounds that are currently used along with their problems:

1- Using the character data type to hold 8-bit unsigned integer values

Considering byte values as characters is not always possible as some implementations of Fortran limit the values allowed in the CHAR and ICHAR intrinsic functions to the 127 ASCII characters, thus not accessing the full byte.

2- Converting the unsigned integer to a signed integer capable of representing the maximum value of the unsigned integer, for example a 32-bit signed integer to hold a 16-bit unsigned integer.

This method has the inconvenient of wasting space, which may be significant for some applications such as image processing where huge amounts of data are processed.

3- Transforming the unsigned values to signed values by subtracting an appropriate value.

Since this method requires extra processing, it is not optimal.

Consequently, we feel that an unsigned integer facility should be added in this standard in order to satisfy user needs. Our proposal is described beginning at the following page.

The proposal

Our proposal provides a complete unsigned integer facility with minimal syntactical additions to the language.

The model set

The model set for unsigned integers is similar to the one for signed integers **(Section 13.4: Numeric Models** of the draft) with the exception that there is no **s** (sign) multiplier. For bit manipulation purposes, the model defined at section 13.3 is useable directly.

Implementation

Unsigned integers should be provided as additional KINDs of the basic INTEGER type. To select an unsigned integer kind, the SELECTED_INT_KIND is extended by adding an optional UNSIGNED logical argument. Thus SELECTED_INT_KIND(2,.TRUE.) or SELECTED_INT_KIND(2,UNSIGNED=.TRUE.) will return a kind suitable for an unsigned integer where the range 0 to 99 can be represented.

Using kinds for implementing unsigned integers has the advantage of requiring no additional syntax to specify constants and to reuse the intrinsics. Unsigned integers should be allowed in all places where nondefault kind integer quantities (variables or expressions) are permitted, except at places where a negative value is likely to be returned (such as IOSTAT).

Conversion rules when unsigned integer operands are mixed with other types in expressions

When unsigned integer operands are mixed with REAL and COMPLEX operands in an expression, the same rules apply as for ordinary integers: the unsigned integer is converted to the type and kind of its corresponding REAL and COMPLEX operand.

When unsigned integer operands of different kinds are mixed, the kind of the result is that of the operand that allows for the maximum representable value. For example if the kind for the first operand allows for values ranging from 0 to 255 and the kind of the second operand allows a range from 0 to 65536, the kind of the result will be of the second operand.

When unsigned integers are mixed with signed integers, the issue is not clear and we present four possibilities:

1- Disallow mixing signed and unsigned integers.

This is the "safe" approach if any of the following suggestions seems too complicated or error prone.

2- The kind of the result is that of the signed integer operand.

This is consistent with the Fortran behavior (inferred, not explicitly mentioned in the standard) of converting from the simpler data type to the more complex data type. Unsigned integers have a simpler representation than signed integers and are thus "lower" in the hierarchy. There are some risks of overflow or underflow, however, if the programmer is not careful.

3- Use a behavior similar to C, that is, use the "size" of the variable to determine the kind of the result.

It is agreed that the C rules are very error prone, particularly when both operands have the same "size".

4- The kind of the result is that of a signed integer capable of representing the maximum positive value of either operand.

The rule is the following: Let two operands **A** and B. The kind of the result is that of a signed integer capable of representing MAX(HUGE(A),HUGE(B)). If such a kind is not available, the program is not standard conforming.
Of course, only one possibility should be selected by the committee.

J3 Response: This feature was on the list of features for Fortran 2000, but it did not receive enough support in order to make it into this version of Fortran. We will suggest that it be added to the list of features for consideration for the next revision of the Fortran standard.

4- CONVERSION EXPLICIT Statement

Justification for requirement:

It has been mentioned several times that allowing mixed types and kinds in intrinsic numeric expressions, assignments, and initializations is error prone and may conduce to hard to find bugs. For example, when a default real constant is inadvertently assigned to a double precision variable, the programmer is often surprised by the precision loss. Also, when integer variables are mixed with real variables in an expression, if the programmer is not careful, unexpected truncations may occur because of operator hierarchy. A negative criticism of the Fortran language has been made on that basis.

What we suggest to solve this problem is the introduction of a **CONVERSION EXPLICIT** statement that, when present, forces the programmer to explicitly specify type conversion by means of the usual Fortran type conversion intrinsics.

Description of the feature

The **CONVERSION EXPLICIT** statement has exactly the same scope of an **IMPLICIT** statement. Its position in a program unit should be the same as the **IMPLICIT** statement.

Its effect is that automatic conversion of types and KIND promotions in intrinsic numeric expressions, assignments, and initializations are disallowed. The exponentiation operator remains a special case where an integer exponent should be allowed to be used with a real or complex base.

Note: in an early draft of "Fortran 8x" published in the eighties, the statement CONVERSION NONE was proposed for the feature. This is an acceptable alternative spelling.

J3 Response: Your idea raises several interesting related possibilities. We will suggest that it be added to the list of features for consideration for the next revision of the Fortran standard.

5- FIRSTLOC and LASTLOC array location functions

Justification for requirement:

Searching for the first or last location of the occurrence of a particular value in an array is a frequent programming task. However, this operation is awkward to program efficiently using Fortran array notation. As a result, serial DO loops are needed to accomplish this simple task.We suggest two intrinsic functions that provide this functionality. We can give two examples of the usefulness of such a search capability:
1- Finding the position of the first item that matches a given criterion in an unordered list.
2- Finding the position of the first and last nonzero pixels in a digitized image represented as a two-dimensional array.

Description of the feature

We have prepared two entries similar to those found in the Fortran 2000 draft to describe these two functions.

FIRSTLOC (MASK [, DIM, KIND])

Description. Determine the location of the first .TRUE. element of MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK shall be of type logical. It shall not be scalar.
DIM (optional) shall be scalar and of type integer with a value in the range 1 £ DIM £ n, where n is the rank of MASK. The corresponding actual argument shall not be an optional dummy argument.
KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM is absent, the result is an array of rank one and of size equal to the rank of MASK; otherwise, the result is of n − 1 and shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

Case (i): The result of FIRSTLOC (MASK) is a rank-one array whose element values are the values of the subscripts of an element of MASK which is the first element having the value .TRUE. occurring in array element order. If there is at least one element having a .TRUE. value, the ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of MASK. If all the elements of MASK have the value .FALSE., then the ith subscript returned has the value $e_i + 1$ (See note). If MASK has size zero, all elements of the result are zero.

Case (ii): If MASK has rank one, FIRSTLOC (MASK, DIM = DIM) is a scalar whose value is equal to that of the first element of FIRSTLOC (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ of the result is equal to

FIRSTLOC (MASK($s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n$), DIM=1)

Examples:

Case (i): The value of FIRSTLOC((/.FALSE.,.TRUE.,.TRUE.,.FALSE./)) is [2]; The value of FIRSTLOC((/.FALSE.,.FALSE.,.FALSE./)) is [4].

Case(ii): The value of FIRSTLOC((/.FALSE.,.TRUE.,.TRUE.,.FALSE./), DIM=1) is 2.

If B has the value $\begin{bmatrix} 1 & 2 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, FIRSTLOC(B == 2, DIM=1) is [2, 1, 3] and FIRSTLOC(B == 2, DIM=2) is [2,1].

LASTLOC (MASK [, DIM, KIND])

Description. Determine the location of the last .TRUE. element of MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK shall be of type logical. It shall not be scalar.
DIM (optional) shall be scalar and of type integer with a value in the range $1 \le DIM \le n$, where n is the rank of MASK. The corresponding actual argument shall not be an optional dummy argument.
KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM is absent, the result is an array of rank one and of size equal to the rank of MASK; otherwise, the result is of rank $n - 1$ and shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

Case (i): The result of LASTLOC (MASK) is a rank-one array whose element values are the values of the subscripts of an element of MASK which is the last element having the value .TRUE. occuring in array element order. If there is at least one element having a .TRUE. value, the ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of MASK. If all the elements of MASK have the value .FALSE., then all elements returned are zero (See note). If MASK has size zero, all elements of the result are zero.

Case (ii): If MASK has rank one, LASTLOC (MASK, DIM = DIM) is a scalar whose value is equal to that of the first element of LASTLOC (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ of the result is equal to

LASTLOC (MASK($s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n$), DIM=1)

Examples:

Case (i): The value of LASTLOC((/.FALSE.,.TRUE.,.TRUE.,.FALSE./)) is [3];
The value of LASTLOC((/.FALSE.,.FALSE.,.FALSE./)) is [0].

Case(ii): The value of LASTLOC((/.FALSE.,.TRUE.,.TRUE.,.FALSE./), DIM=1) is 3.
If B has the value $\begin{bmatrix} 1 & 2 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, LASTLOC(B == 2, DIM=1) is [2, 2, 0]
and LASTLOC(B == 2, DIM=2) is [2,2].

Note : The choice of returning a position of $extent_i$+1 when there are no .TRUE. elements in FIRSTLOC and 0 in the case of LASTLOC for the same case is consistent with the behavior observed in the case of an identical search operation performed with DO loops. However, the INDEX intrinsic function returns 0 when

the substring is not found irrespective of the BACK parameter. If the committee chooses to replace the proposed behavior by something else, the sentences in the function description describing the behavior and the values in the examples corresponding to the all .FALSE. condition are shown in red to ease their replacement.

J3 Response: We will suggest that it be added to the list of features for consideration for the next revision of the Fortran standard.

References

02-007r3, Fortran 2000 Committee Draft

[End of J3 / 02-318r1]