

# Widest-Need Expression Processing

G. William Walster

October 20, 2002

## Abstract

Widest-need expression processing supports mixed-mode interval expression evaluation while it satisfies the containment constraint of interval arithmetic and avoids unnecessary interval width. While widest-need expression processing was developed for a non-interval application, it is the ideal method for evaluating mixed-interval expressions.

## 1 Introduction.

The Fortran 95 standard requires literal and named constants (objects of the `PARAMETER` attribute) to be values of their type and kind type parameter value (KTPV). Therefore, in code example 1 the literal constants, 0.1 or 0.1E0 and R in line 3 must have the same value. Similarly, 0.1D0 must equal the value of D in line 4.

### Example 1

```
REAL ::R
DOUBLE ::D
R = 0.1      ! line 3
D = 0.1D0    ! line 4
```

The standard also requires that DR and DD in code example 2 have *different* values.

### Example 2

```
DOUBLE :: DR, DD
DR = 0.1
DD = 0.1D0
```

DR is the value that is obtained from code example 3 line 4, and is not equal to 0.1D0:

### Example 3

```

REAL :: R
DOUBLE :: DR
R = 0.1
DR = R          ! line 4

```

As a consequence, users must be careful to specify the desired KTPVs of literal and named constants. Otherwise, surprisingly inaccurate results may be obtained. Changing the KTPV of variables in expressions containing literal constants is not possible without also making explicit changes to literal constants. The term “named constant” is something of a misnomer. The term “read-only variable” better connotes the fact that the object of a `PARAMETER` attribute is restricted to be an element of the same set of values as a variable with the same type and KTPV as the named constant.

Throughout this note, unless explicitly stated otherwise, `INTEGER`, `REAL`, and `INTERVAL` constants are literal constants. Constant expressions and named constants are always explicitly identified as such.

## 2 Interval Arithmetic

The purpose of interval arithmetic is to provide a simple, automatic and mechanical way to perform machine computations while rigorously bounding errors from all sources. In particular, machine rounding errors and their propagation throughout algorithms must be rigorously bounded. Rigorous bounds are produced by defining an interval:

$$[a, b] = \{x \in \mathbb{R}^* \mid a \leq x \leq b\} \quad (1)$$

and arithmetic on pairs of intervals  $[a, b] \text{ op } [c, d]$ , where  $\text{op} \in \{+, -, *, /\}$ :

$$[a, b] \text{ op } [c, d] \supseteq \text{hull} \left( \left\{ z \mid \begin{array}{l} z \in x \overline{\text{op}} y \\ x \in [a, b] \\ y \in [c, d] \end{array} \right\} \right); \quad (2)$$

$\text{hull}(\cdot)$  is the interval hull relation, and  $\overline{\text{op}}$  is the topological closure of the operation,  $\text{op}$ . See [4] for definitions and details.

## 3 Interval Support Design Questions

Implementing computer language support for interval data types presents a number of interesting questions, the answers to which can dramatically impact an `INTERVAL` data type’s ease of use.

1. How shall `INTERVAL` literal constants be denoted?
2. Should interval expressions containing components with different interval KTPVs be permitted?
  - (a) If so, how should they be evaluated?

3. Should interval expressions containing non-interval types be permitted?

(a) If so, how should they be evaluated?

### 3.1 Representing INTERVAL Constants

The obvious Fortran notation for an interval literal constant is:

`[A,B];`

where A and B are INTEGER or REAL constants, but not named constants.

A convenient notation is adopted to denote degenerate intervals in which both endpoints have the same value:

$[A, A] \equiv [A]$

This same convention is used in f95.

## 4 The Containment Constraint.

The answers to the remaining questions depend on the fundamental requirement of any interval implementation system: *containment*. That is, evaluating any interval expression *must* produce an interval containing the set of values that can be produced by any point expression contained in the interval expression. This set is called the *containment set* of the expression and is proved in [5] to be the closure of the expression. Computing narrow-width interval results and computing results fast are both desirable goals. However, containment is a constraint. Violating the containment constraint is a fatal error.

## 5 Design Question Answers

With context provided by the interval containment constraint and the narrow width interval goal, answers to questions 2 and 3 become clear.

### 5.1 Internal Approximation of Literal Interval Constants

Within the Fortran Standard, there is no accuracy requirement for the internal approximation of mathematical constants. However, in the INTERVAL constant `[A,B]`, if A and B are INTEGER or REAL constants, the information exists during compilation to construct INTERVAL constants that both satisfy the containment constraint *and* are as narrow as possible, or *sharp*. See [2] for a discussion of how the containment constraint and sharpness goal lead to context-dependent interval constant approximations.

The information exists at compile-time to convert `[0.1]` into a 1 ulp-width<sup>1</sup> interval containing the external mathematical value  $1/10$  (denoted *ev* (0.1)) with

---

<sup>1</sup>The mnemonic for *unit in the last place* is: ulp..

Code	Mathematical Interpretation
[0.1, 0.2]	$[a, b]$ where $a \leq \text{ev}(0.1)$ and $\text{ev}(0.2) \leq b$
[0.1]	$[a, b]$ where $a \leq \text{ev}(0.1)$ and $\text{ev}(0.1) \leq b$

Table 1: Mathematical value of `INTERVAL` constants.

any KTPV that is appropriate. However, according the standard 0.1 is a real floating-point number that approximates the ideal decimal number  $\text{ev}(0.1)$ . Since the standard is silent regarding the accuracy of the approximation, it is difficult using only standard Fortran numerical constructs to build a 1 ulp-width interval with a given KTPV that contains any arbitrary mathematical value. While a run-time callable intrinsic can be constructed with a `CHARACTER STRING`-type argument to interpret decimal numeric values and convert them into sharp containing intervals, this kind of run-time infrastructure should be unnecessary in compiler-provided language support for interval data types.

To remove the conflict between the standard and the interval containment constraint, `f95` implements interval support to maximize user utility. Support for intervals in `f95` 6.0 adheres as closely as possible to both the letter and the spirit of the Fortran standard. *However*, when the standard conflicts with either the interval containment constraint or ease of use, new syntax and semantics are introduced to achieve an acceptable level of interval-support quality.

Interval constants consist of a pair of enclosing square brackets, within which are found, either a single `INTEGER` or `REAL` decimal number, or a pair, separated by a comma. Spaces are optional. Neither named constants nor constant expressions are permitted.:

In the examples in Table 1 of default interval constants, 0.1 and 0.2 are default real constants. The function  $\text{ev}(\cdot)$  is used to denote a Fortran constant's external value.

For more details regarding the construction and interpretation of default and non-default interval constants, see [3] and [2].

## 5.2 Mixed kind-type interval expressions

The next design question is whether to permit different KTPV sub-expressions within an interval expression. For example, should code examples 4 and 5 be permitted?

### Example 4

```
[0.1_4, 0.2_4] + [0.3_8]
```

### Example 5

```
INTERVAL(4) :: X
INTERVAL(8) :: Y,Z
X = [0.1_4, 0.2_4]
Y = [0.3_4]           ! line 4
Z = X + Y             ! line 5
```

What values should be used for the interval constants in code example 4, and what KTPV should the resulting interval have?

In code example 5, the first question is whether the context of the expression should be permitted to influence the value of literal constants. For example, should the fact that Y is a KIND=8 interval in line 4 influence how the KIND=4 interval constant [0.3\_4] is interpreted?

The second question is whether in line 5:

- i. should X be promoted to KIND=8 before performing the interval addition of  $X + Y$ ; or
- ii. should Y be truncated to a containing interval? For example, in line 5 of code example 6, should the fact that Z is KIND=8 influence how  $X*Y$  is computed?

#### Example 6

```

INTERVAL(4) :: X, Y
INTERVAL(8) :: Z
X = [0.1_4, 0.2_4]    ! line 3
Y = [0.3_4]
Z = X*Y               ! line 5

```

In standard Fortran, there is no accuracy requirement for the evaluation of any `INTEGER` or `REAL` numeric expression. Therefore, there is no standard metric with which to compare the accuracy of compiler implementations. Speed is the only evaluation criterion. As a consequence, Fortran expression evaluation rules have been completely specified to operationally define standard compliance. The responsibility for result accuracy rests totally on the developer and end user. The difficulty is that it remains possible for standard conforming Fortran to produce very different results on different platforms, or on the same platform if run in a non-deterministic way, such as occurs on a parallel machines. Without an obvious criterion to use when choosing between alternative standard-conforming implementations, even on a given word-length processor, the only remaining evaluation criterion is speed.

Intervals change these rules. With intervals, given the containment constraint is satisfied, there are two evaluation criteria: speed *and* interval width. Runtime is obvious. So is the width of an interval result. Therefore, it is always possible to evaluate whether one interval expression evaluation strategy is better than another. The question is: how to balance speed and interval width against each other when they conflict?

Fortunately, rules for a reasonable compromise system have been defined by Robert Corbett [1]. The system is called “widest-need” expression processing. While not developed in the context of interval arithmetic, widest-need expression processing is nevertheless a reasonable system with which to implement mixed-mode interval expression processing.

Under widest-need expression processing, the context of the entire expression is taken into account to determine the maximum interval KTPV, denoted

KTPV\_max, that occurs anyplace within an expression. The logical steps in the process are:

1. Promote all INTEGER and REAL constants and variables to intervals with the KTPV required to exactly represent the non-interval constant or variable.
2. Compute the maximum KTPV, KTPV\_max.
3. Convert all (now interval) constants and variables to KTPV\_max, *before* interval expression evaluation begins.

The cost is additional compile-time to promote all non-interval constants and variables and then to initially scan each expression tree to determine KTPV\_max. As soon as a KIND=16 constant or variable is encountered, the search for KTPV\_max can be terminated.

What does widest-need expression processing do in the above examples? For:

[0.1\_4, 0.2\_4] + [0.3\_8]

in code example 4, the first constant is promoted to KIND=8 before performing the interval addition. The sharpest possible constant expression result can be achieved if exact decimal arithmetic is employed prior to constructing the containing interval result. This has not been done in f95 and remains an outstanding “quality of implementation opportunity”.

In the following examples the equivalent `-xia=strict` code using the intrinsic `INTERVAL` constructor exposes the logical steps that are automatically performed under widest-need expression processing. The code using the `INTERVAL` constructor can be run using either the `-xia=widestneed` or the `-xia=strict` command-line flag to enable or suppress widest-need expression processing, respectively. Widest-need is the default interval expression processing mode with the command line flag `-xia` or `-xia=widestneed`. Example 8 contains only the `-xia=strict` lines from the equivalent widest-need code lines in example 7.

#### Example 7

```
INTERVAL(4) :: X, Y
INTERVAL :: Z
X = [0.1, 0.2]      ! line_3
Y = 0.3             ! line 4
Z = X+Y             ! line 5
```

In `-xia=strict` mode, lines 3, 4 and 5 must be changed to:

#### Example 8

```
X = [0.1_4, 0.2_4]      ! line_3
Y = INTERVAL([0.3],KIND=4) ! line 4
Z = INTERVAL(X, KIND=8) * INTERVAL(Y, KIND=8) ! line 5
```

In line 3, because the KTPV of X is 4, the interval constant on the right-hand side must also have the same KTPV. Default intervals have KTPV=8.

Therefore, either the interval constant must be constructed with KTPV=4 constants, or it must be explicitly converted using the `INTERVAL` constructor, as in:

```
X = INTERVAL([0.1, 0.2],KIND=4) ! line_3
```

To guarantee containment of  $ev(0.3)$  in line 4, the interval constant `[0.3]` must be given as the argument of the `INTERVAL` constructor and then explicitly converted to KTPV=4. Alternatively, the KTPV=4 interval constant, `[0.3_4]` can be used, but this requires a code change.

In line 5, because `Z` is a default interval with KTPV=8, both `X` and `Y` must be converted to KTPV=8 before the right-hand side can be evaluated.

Note, however, that `Z` can be more sharply computed if the KTPVs of `X` and `Y` are 8. Code that executes under `-xia=strict` mode always runs under widest-need mode and produces the same answers. It is an error in the compiler if code that runs in `-xia=strict` mode ever produces different results when run in `-xia=widestneed` mode.

### 5.3 Mixed type interval expressions

Now consider expressions that contain both interval and non-interval constants and variables. Should these be allowed? If so, how should they be processed?

Consider the illustrative code example:

#### Example 9

```
REAL(16) ::Q
INTERVAL(4)::Y
INTERVAL ::Z
Q = 0.1_16
Y = 0.3 ! line 5
Z = Q + Y ! line 6
```

The desired outcome is to promote to an interval any non-interval variable or constant in a mixed type expression. In addition, the promotion should be to the highest interval KTPV seen anywhere in the expression.

In code example 9 line 5, because the type of `Y` is `INTERVAL`, the literal constant `0.3` is promoted to a containing interval. The maximum of the KTPV of `Y` and `0.3`, which is 4 in this case, is used when setting the KTPV of the interval containing `0.3`.

In line 6, because the KTPV of `Q` is 16, and because `Z` and `Y` are both intervals, `Q` is promoted to an interval and `Y` must be promoted to a KTPV=16 interval. After promotion the expression `Q + Y` is evaluated. Only after the KTPV=16 result is computed, is it converted to a KTPV=8 containing interval before assigning the final result to `Z`.

If the principle of widest-need expression processing is used, the following logical steps are taken:

1. The KTPV is determined that will permit each `INTEGER` and `REAL` constant or variable to be promoted to a degenerate interval.
2. The value of `KTPV_max` is found.
3. All interval constants and variables are promoted to intervals with `KIND = KTPV_max`.

The following code is the `-xia=strict` equivalent of lines 5 and 6, in example 9.

#### Example 10

```

Y = INTERVAL([0.3],KIND=4)           ! line 5
Z = INTERVAL(INTERVAL(Q,Q,KIND=16) + & ! line 6
  INTERVAL(Y,KIND=16),KIND=8)

```

All type and KTPV conversions must be made explicitly.

## 5.4 Conclusion

Comparing the readability of `-xia=widestneed` and the equivalent `-xia=strict` code illustrates the advantages of widest-need expression processing. To uniformly decrease the width of word-length-dependent interval computations, it is only necessary to change variable's KTPV. It is not necessary to touch expression code.

## References

- [1] R. P. Corbett. Enhanced arithmetic for fortran. *SIGPLAN Notices*, 17(12):41–48, Dec. 1982.
- [2] G. W. Walster. Literal interval constants. Technical report, Sun Microsystems, August 1999.
- [3] G. W. Walster. *Interval Arithmetic Programming Reference: Forte™ WorkShop 6 Fortran 95*. Sun Microsystems, May 2000. See: <http://www.sun.com/forte/fortran/interval/index.html>.
- [4] G. W. Walster. The “Simple” Closed Interval System. Technical report, Sun Microsystems, February 2000.
- [5] G. W. Walster, E. R. Hansen, and J. D. Pryce. Extended real intervals and the topological closure of extended real relations. Technical report, Sun Microsystems, February 2000.