J3/97-198

Date: July 23, 1997
From: G. William Walster and Eldon R. Hansen
To: J3
Subject: Composite Functions in Interval Mathematics

Abstract

Composite interval extensions of functions and relations are required to justify expression folding and other mathematical transformations at compile-time in an optimizing compiler supporting interval data types. The composite of interval extensions is proved to contain the corresponding interval extension of composites. This leads to a deeper analysis of interval mathematics and its relation to point mathematics. In turn, analysis of composites leads to new language and compiler requirements.

1 Introduction

The design of language support for interval data types must include allowable mathematical transformations at compile-time. Considering alternatives in the context of interval extensions to Fortran leads to the conclusion that intervals are fundamental mathematical objects, with their own unique properties, and the same (or greater) stature as integer, real, and complex variables.

The requirement to perform expression folding at compile-time depends on the existence of composites of interval extensions of functions and relations. Composites of interval extensions are proved to contain corresponding interval extensions of composites. These ideas are motivated by the requirement to define what an optimizing compiler may be permitted to do at compile-time. However, the implications are much broader because these same ideas can be used more generally to define interval mathematics and its relation to point mathematics.

2 Interval Arithmetic

Let $[a,\ b]$ and $[c,\ d]$ denote non-degenerate interval constants and let $X := [a,\ b]$ and $Y := [c,\ d]$ denote the assignment of the value of these interval constants to the interval variables X and Y, respectively. It will be shown that there is an important distinction to be made between interval variables and constants. Let "*op*" denote a member of the set of arithmetic operations $\{+,\ -,\ \times,\ /\}$. The rules of interval arithmetic (e.g., see [2]) require that $f(X,Y)$ contain the set $\{f(x,y) \mid x \in X,\ y \in Y\}$, where $f(X,\ Y) \equiv X\ op\ Y$ and $f(x,y) \equiv x\ op\ y$.

3 Point Composite Functions and Expression Folding

In point mathematics, composite functions are defined as follows:

Definition: Given the functions $f(x)$ and $g(x)$, define $h(x) = f(g(x))$ and the domain of h such that x is in the domain of g and g is in the domain of f.

Definition: Given $h(x,y)$ in addition to f and g, define $q(x,y) = h(f(x), g(y))$, with the domain of q being defined by the domains of f, g, and h.

The Fortran standard [3] permits a compiler to substitute "mathematically equivalent" expressions within a given statement. However the standard clearly defines the "=" operator to be an assignment of value, not a macro definition. Expression folding is thereby expressly forbidden. Nevertheless, in the interest of run-time performance, optimizing compilers perform expression folding of the following sort:

Original Code:	Folded Code:
X=A+B **Y=X-A**	**Y=B**

Implicit functions can be used to prove the above codes are mathematically equivalent:

$$
\begin{aligned}
y &= f(g(a,b),a);\ \text{where} \\
g(a,b) &= a+b \ \text{ and } \ f(x,a) = x-a, \ \ \text{from which it follows that} \\
y &= a+b-a = b.
\end{aligned}
$$

However, when evaluated using floating-point arithmetic, rounding errors can cause the two codes to produce completely different results. For example, if **A** is sufficiently larger than **B** , there will be no agreement between the folded and unfolded forms. One way to give a compiler explicit permission to perform expression folding, is to permit the "=" operator to be to be interpreted as mathematical identity in the macro substitution sense. Note, this is a permissive interpretation, not a required interpretation.

In the above example it is clear that the folded code is both more accurate and faster. In general, the best way to compute a given result may not always be obvious. Accuracy and speed may sometimes be in conflict, or in some situations assignment of value may be required. Current compilers perform expression folding only for speed, usually under high levels of optimization, but not under explicit program control. Algorithm developers will want to explicitly control when expression folding is permitted.

To illustrate the need for both expression folding and assignment of value, consider the case in which $y = \sum_{i=1}^{n} x_i$. Later it may be desired to compute $d_j = y - x_j$ for some value of j. If x_j is much greater than $\sum_{i \neq j} x_i$, $y - x_j$ will be inaccurate if computed using floating-point arithmetic. However, in many cases, it may be sufficiently accurate and much faster to compute $d_j = y - x_j$ rather than $\sum_{i \neq j} x_i$. In such cases, assignment of value is required to prevent expression folding and in others, macro definition is required to permit it.

One way to control when expression folding is permitted is with two operators: "$\equiv$" and ":=", defined to mean identical algebraic equality as in the definition of a macro and assignment of value, respectively. Given current practice and the relative frequency with which $\equiv$ and := will be used, the existing "=" operator can be interpreted as $\equiv$ and introduce a new ":=" operator can be introduced to force assignment of value. Backward compatibility with existing code can be easily achieved by forbidding expression folding with a pragma and/or a command line option. This is equivalent to forcing both = and := in assignment statements to be interpreted as assignment of value.

4 Interval Composites and Expression Folding

To justify interval expression folding it is necessary to define the interval extension of a composite function, the composite of interval extensions, and how they are related. Given real functions, $f(x)$, $g(x)$ and the composite function $h(x) = f(g(x))$, any interval extension $h(X)$ of the composite function $h(x)$ must contain $\{h(x) \mid x \in X\}$. If inclusion monotonic interval extensions of f and g exist, an inclusion monotonic interval extension of the composite function h can be constructed from the composite of the interval extensions:

$$h(X) = f(g(X)) = f(\{g(x) \mid x \in X\}). \tag{1}$$

Inclusion monotonicity of f and g, guarantees that $h(X)$ is inclusion monotonic and contains: $\{h(x) \mid x \in X\}$. Letting $X = x$, at once yields: $H(x) = h(x) = f(g(x))$; where $H(x)$ denotes the interval extension, $h(X)$, evaluated at the point, x. Provided intrinsic and user-defined routines are valid inclusion monotonic interval extensions of their respective underlying point functions or relations, any composite of interval extensions always yields a valid inclusion monotonic extension of the underlying composite function or relation. It may not be sharp, but it will be a valid inclusion monotonic interval extension.

What follows at once is that interval expression folding can be performed in exactly the same way it is with point expressions. With points, different results of varying accuracy may be computed. With intervals, sharpness may vary. It is the challenge of an optimizing interval compiler to use transformations that will result in fast code to compute sharp results, while guaranteeing containment.

5 Independent and Dependent Intervals

An interval variable can be viewed in one of two different, but mathematically equivalent ways. It may be a single unknown point in the interval. In this view, the endpoints merely serve as bounds on the point. Alternatively, it may be the set of single points in the interval. In this view, the interval serves

to specify the set of single points. In either view, every single point in every occurrence of the same interval variable is identical. The individual points are therefore dependent. In contrast, an interval constant is merely the set of all possible values in the interval. There is no possibility of dependence between multiple occurrences of an interval constant.

Point and interval constants and variables obey the commutative and associative laws of algebra for addition and multiplication. Point constants (equivalently, degenerate interval constants), point variables, and interval variables, also obey the distributive law of algebra and have additive and multiplicative inverses. However, each occurrence of a non-degenerate interval constant must be treated as if it were a new interval variable, independent of all other occurrences of the same interval constant. Let $[a,\ b]$ be a constant interval. Then, for example, computing $[a,\ b] - [a,\ b]$ and $[a,\ b]/[a,\ b]$ is the same as computing $X - Y \supseteq \{x - y \mid x \in X,\ y \in Y\} = [a - b,\ b - a]$ and $X/Y \supseteq \{x/y \mid x \in X,\ y \in Y\} = [\min(a/b, b/a),\ \max(a/b, b/a)]$; where $X := Y := [a,\ b]$ and in the case of division, $[a,\ b]/[0,\ 0] = \emptyset$ It is not possible for two non-degenerate interval constants to be completely dependent or identically equal. An interval variable is completely dependent on itself and identically equal to itself. Contrary to interval constants, interval variables have additive and multiplicative inverses: $X - X \supseteq \{x - x \mid x \in X\} = 0$ and $X/X \supseteq \{x/x \mid x \in X\} = 1$.

This useful distinction between interval constants and variables has been largely overlooked in the development of interval arithmetic. So far, interval arithmetic has been developed under the assumption that intervals are constants. As shown above, interval variables have all the algebraic properties of points. This fact can be used to great advantage at compile-time to perform algebraic transformations that would otherwise not be possible.

A somewhat different view of the distinction between interval variables and constants can be seen by letting $f(X)$ and $g(X)$ denote the interval extensions of the real function and relation $f(x)$ and $g(x)$, respectively. Functions are single-valued and relations are multi-valued. Examples of simple functions are: $f(x) = a$ or $f(x) = x$. Examples of simple relations are: $g(x) = [a, b]$ or $g(x) = [a,\ b]x$. Both functions and relations can have interval extensions. For example, $f(X) \supseteq \{f(x) \mid x \in X\} = a$ and X are interval extensions of the functions $f(x) = a$ and $f(x) = x$, respectively. $g(X) \supseteq \{g(x) \mid x \in X\} = [a,\ b]$, and $[a,\ b]X$ are interval extensions of the relations $g(x) = [a,\ b]$ and $g(x) = [a,\ b]x$, respectively. Only interval exten-

sions of functions can be completely dependent or identically equal. Interval extensions of relations can be partially, but never completely dependent. Pure interval constants are completely independent.

There is a case in which dependence among intervals has been recognized. The ability to perform subtraction with cancellation is described in [1], page 10. Suppose the interval $Y = \sum_{i=1}^{n} X_i$ is computed. Later it is desired to compute $D_j = Y - X_j$ for some value of j. In the cases where it is sufficiently accurate to compute $D_j = Y - X_j$ rather than $\sum_{i \neq j} X_i$, the following subtraction with cancellation algorithm can be used:

$$X \ominus Y = [a - c,\ b - d] \text{ ; where} \tag{2}$$

$X = [a, b]$, $Y = [c, d]$, and for this operation to be valid, Y must be an additive component of X.

6 Independent and Dependent Functions

There is yet another way to distinguish between dependent and independent intervals. Two *point* functions (or relations) f and g are dependent if they share at least one common variable argument. For example, $f(x, y)$ and $g(u, v)$ are independent, while $f(x, y)$ and $g(u, x)$ are dependent. Two *interval* functions (or relations) f and g are dependent if they share at least one common variable argument. For example, $f(X, Y)$ and $g(U, V)$ are independent, while $f(X, Y)$ and $g(U, X)$ are dependent. Neither point nor interval constants count. That is, $f(2, y)$ and $g(2, v)$ are independent. Similarly, $f([a,\ b], Y)$ and $g([a,\ b], V)$ are independent.

Composites of dependent functions (or relations) can sometimes result in significant simplifications. For example, consider the function $h(x, y) = x\ op\ y$, where $op \in \{+,\ -,\ \times,\ /\}$. An example of a composite h of two dependent functions is: $h(f(x, y), f(u, x))$; where $f(x, y) = x + y$ and $h(x, y) = x - y$, from which it follows that

$$h(f(x, y), f(u, x)) = y - u \tag{3}$$

A consequence of the extension of composite functions to intervals is that any dependent interval expression can be simplified in exactly the same ways

as its point counterpart. For example, given inclusion monotonic interval extensions of f and h, it follows that:

$$h(f(X,Y), f(U,X)) \supseteq \{y - u \mid y \in Y,\ u \in U\} = Y - U \qquad (4)$$

7 Interval Arithmetic on Dependent Intervals

Two intervals that are functions of the same interval variable are dependent. Consequently, $X\ op\ X$ is different from $X\ op\ Y$ or $[a,\ b]\ op\ [a,\ b]$ for $op \in \{+,\ -,\ \times,\ /\}$, because X depends on itself. In this case, the following rules apply for $X := [a,\ b]$:

Expression	Result
$X + X$	$[2a, 2b]$
$X - X$	0
$X * X = X^2$	$\left\{ \begin{array}{c} [\min\{a^2,\ b^2\},\ \max\{a^2,\ b^2\}] \quad \text{if} \quad a > 0 \quad \text{or} \quad b < 0 \\ [0,\ \max\{a^2,\ b^2\}], \quad \text{otherwise} \end{array} \right\}$
X/X	1

As noted above, analogous simplifications do not exist for $[a,\ b]\ op\ [a,\ b]$, where $[a,\ b]$ is a constant interval. These results must be obtained by substituting a for c and b for d in the normal relations for two independent intervals. Non-degenerate interval constants are independent, even though their corresponding endpoints are equal. Thus, for example, while $2 - 2 = 0$, $[2,\ 3] - [2,\ 3] = [-1,\ 1]$. These results expose the fundamental difference between interval arithmetic on constants and interval mathematics on variables.

The analogous distinction is not made for points, at least not explicitly. The rules of algebra of points are typically qualified with a statement such as: "except for division by zero". The real numbers can be extended to include $\pm\infty$ to cover division of a non-zero number by zero. However, $0/0$ is not defined and $f(x)/g(x)$, when both f and g approach zero at a point, requires

additional analysis, such as with L'Hopital's rule. Nevertheless, no thought is ever given to canceling x/x or $f(x)/f(x)$ and replacing them by 1, either in complete expressions or when such cancellations can be made in a larger expression. The reason may be that there exists an unstated recognition of the subtle difference between point numbers and variables that is more obvious in the case of intervals.

8 Interval Expression Folding

There is a restriction on interval expression folding that does not exist with points: because interval constants are independent, they may not be treated as identically equal. Thus, consider the following example:

Original Code:	Folded Code:
X=[0,1]*B **Y=[0,1]*B** **Z=Y-X**	**Z=[-1,1]*B**

While it is permissible to take into account the dependence between the two occurrences of the interval variable **B** , the two occurrences of the interval constant **[0,1]** are independent.

9 Fortran PARAMETER Statements

Assignment of value can lead to a sharper result when operating with interval constants. Consider the following two examples:

Example: A

Original Code:	Folded Code:
X=[0,1] **Y=X-X**	**Y=[-1,1]**

Example: B

Original Code:	Folded Code:
X:=[0,1] **Y=X-X**	**Y=0**

In example A, the constant **[0,1]** is substituted for **X** in the expression for **Y** . In example B, the value **[0,1]** is assigned to the variable **X** . This permits the cancellation to be made in the expression for **Y** . For this reason, the definition of the "= " operator in Fortran PARAMETER declarations must be that of a macro. In this case, the PARAMETER is simply a named constant. Only if the new assignment of value operator ":= " is used in its definition, can a PARAMETER be regarded as a read-only variable.

10 Conclusion

From the inception of interval arithmetic, unnecessary constraints have been imposed that may have resulted from following the traditional approach in point mathematics of building constructs, one on top of another. In this respect, interval mathematics is different. Any interval computation is an extension of an underlying real function or relation. As a consequence, any computer language used to implement intervals must not operationally prescribe how interval expressions are to be evaluated. To do so can unnecessarily constrain compiler developers from constructing ever sharper implementation methods. Rather, there is only one requirement for a valid interval implementation: containment. Speed and sharpness are goals. Containment is a constraint. Compiler developers must be free to pursue whatever means at their disposal to construct code that will quickly compute sharp containing intervals. This freedom imposes some new requirements on languages that support intervals and on testing methods used to validate implementation correctness.

References

[1] E. Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, Inc., New York, 1992.

[2] R. E. Moore. *Methods and Applications of Interval Analysis.* SIAM Publ., 1979.

[3] X3J3. International Standard Programming Language Fortran. Technical report, ISO/IEC 1539-1, 1996.