J3/97-158: Compile-time Optimization of Point and Interval Expressions

G. William (Bill) Walster* Keith Bierman†

April 13, 1997‡

Abstract. The speed *and* sharpness of interval programs can be significantly increased by applying mathematical simplifications and other transformations at compile-time. This paper contains both the theoretical justification and the practical results needed to know when the desired transformations can be used.

1. Introduction

Optimizing compilers routinely transform arithmetic expressions to improve performance. For example, if $\mathbf{R}$ is a floating-point variable, the value $\mathbf{0}$ may be substituted for $\mathbf{R} - \mathbf{R}$ during compilation. Substituting one mathematically identical expression for another can be justified on both theoretical and practical grounds. Such substitutions can make code verification and debugging more difficult as some exceptional events[1] may not be detected. The extra development time is often easily justified because of improved performance.

Compiler providers routinely spend tens of millions of dollars to perform ever more advanced optimizations, because increased run-time performance saves end users hundreds of millions of dollars. Optimizing transformations are doubly enticing when applied to interval expressions, as they result both in faster execution *and* in sharper interval results. Interval arithmetic can be used to prove that no exceptional events are hidden by these transformations.

Throughout the modern interval era, beginning in 1966 [4], interval arithmetic has been motivated by the behavior in computers of floating-point arithmetic. The *execution* of interval arithmetic has influenced the definition of the most fundamental

*Sun Microsystems, MS UMPK16-304, 2551 Garcia Avenue, Mountain View, CA 95040-1100, bill.walster@eng.sun.com

†Sun Microsystems, MS UMPK16-304, 2551 Garcia Avenue, Mountain View, CA 95040-1100, keith.bierman@eng.sun.com

‡Thanks to Eldon Hansen amd George Corliss and Milton Barber for their many constructive suggestions concerning earlier drafts of this paper. Special thanks to Kaye Walster, for her patience and support during the paper's preparation and writing.

[1] An ***exceptional event*** in the present context is an IEEE floating-point exception [5], such as division by zero.

interval concepts. Until recently, interval arithmetic has been viewed primarily from an *operational* perspective at run-time, as opposed to a *mathematical* perspective at compile-time. An operational bias is easy to understand, given that there has been no prospect of an interval-supporting compiler to perform optimizing transformations.

The consequences of adopting a mathematical perspective at compile-time are worth considering, now that compiler support for interval arithmetic is imminent [7] [1]. The primary consequence is the theoretical justification for both point and interval compile-time identity transformations. To precisely define when these transformations may be applied, the concept of a *single-valued* interval expression is introduced.

Notation. Throughout this paper, *mathematical identity*, denoted by "$\equiv$", is distinguished from the *assignment of value*, denoted by ":=". Two expressions are *identical* if they are equal whenever the expressions have meaning [3]. Two functions, f and g, are *identical* if they share a common domain, D, and if $f(x) = g(x)$ for all $x \in D$ [3]. In the case of intervals, four distinct definitions of equality exist, three of which are needed herein.

In mathematical notation, lower and upper case letters are used to denote *point* and *interval* variables, respectively. To denote actual code fragments, Fortran notation in **BOLD UPPER CASE** is used. For example, the code, $(< \mathbf{A},\ \mathbf{A} >)$ and $(< \mathbf{A},\ \mathbf{B} >)$[2] denote degenerate and possibly non-degenerate literal interval constants, respectively, while $[a,\ a]$ and $[a,\ b]$, denote the same interval constants using mathematical notation. The code $(< \mathbf{A} >)$ is used to denote an interval that must contain the literal constant **A**. If **A** is not machine representable, $(< \mathbf{A} >)$ cannot be degenerate.

The interval extension of the point function, $f(x)$, is denoted: $f(X)$ or $F(x)$, depending on whether the interval extension is evaluated over the interval, X, or at the point, x, respectively. Either the upper case function argument or the upper case function letter is used to denote the interval extension.

1.1. Overview. In Section 2, the *mathematical* and *operational* perspectives are defined. The mathematical perspective is used to justify the current practice of applying algebraic transformations to point-expressions at compile-time.

In Section 3, the concept of *identically equal interval variables* is defined and distinguished from the other three definitions of interval equality. Both the determination of, as well as the consequences of, each type of interval equality are examined at compile-time and at run-time.

[2] $(< \mathbf{A},\ \mathbf{B} >)$ is the notation used to define the literal interval constant, $[a,\ b]$, in the proposed Fortran 2000 interval arithmetic standard [1].

In Section 4, *identically equal interval expressions* are introduced and used to make simple identity transformations.

In Section 5, *single valued intervals* are defined and the fundamental theorem of interval arithmetic is extended from functions to relations. Single-valued and multi-valued intervals are defined in terms of the underlying point-functions and relations, of which they are respectively extensions.

In Section 6, the application of mathematical identities at compile-time is generalized from interval variables to single-valued interval expressions.

In Section 7, some examples are presented of compile-time transformations that can be used to sharpen single-valued interval expressions.

In Section 8, compile-time transformations and other program modifications are used to illustrate how to sharpen interval expressions for which there is no uniformly sharpest representation.

In Section 9, justification is presented for the requirement to declare single-valued user-defined function subprograms and operators.

Section 10 summarizes the benefits from automatically determining whether a user's function-subprogram is single or multi-valued.

Finally, in Section 11 the required information is presented to automatically determine whether Fortran named constants are single or multi-valued.

2. Simplifying Point Expressions

A single point-expression, or a sequence of assignment statements, can be viewed in two ways: as a series of arithmetic operations that must be performed in a prescribed order, or as the operational definition of an underlying mathematical function of the variables contained therein. The former view is from the *operational* perspective, and the latter, is from the *mathematical* perspective. At run-time, each arithmetic expression is defined by the value of its variables and its operations. At compile-time, useful information is available about the underlying function, in addition to the operations used to compute it. Consider some illustrative examples, in which **E** represents the code for an arbitrary mathematical expression, E:

Code	Mathematics	Value
E − E	$E - E$	**0**
E/E	E/E	**1**
SIN(E) **2 + COS(E) **2	$\sin^2 E + \cos^2 E$	**1**
E ** 0	E^0	**1**

The two principal reasons why it is advantageous to program compilers to recognize and apply such identities are: First, code is often generated by other programs that may not be capable of recognizing these and other possible simplifying transformations. Second, **E** may be arbitrarily complex, making human recognition difficult, especially if **E** is automatically constructed.

When applied to point expressions, a potential problem with such transformations is that they may make it impossible to see program bugs, or even errors in the mathematics on which a program is based. For example, consider $\mathbf{E}/\mathbf{E}$ when $\mathbf{E} = \mathbf{0}$, or any circumstance in which an exception is raised in the process of computing **E**. A frequent cause of such exceptions is the attempt to compute **E**, or a function of **E**, at a point that is outside an expression's domain. For example, $x = 0$ is outside the domain of $f(x) = x/x$. For all other values of x, $f(x) = 1$. If a constant is substituted for an expression that would otherwise cause an exception to be raised, the exception will be hidden from view. While compilers are not required to detect all possible programming errors, ease of debugging is important, and compilers *are* required to produce codes that correctly execute mathematically correct expressions. For particular interval arguments, interval arithmetic can be used to *prove* that a mathematical expression cannot raise exceptions.

2.1. Multiple Assignment Statements. Compilers are capable of recognizing opportunities to perform ever increasingly complex optimizations across multiple assignment statements, or even across procedure boundaries. For example, a compiler may recognize that in place of $\mathbf{X} = \mathbf{E1} + \mathbf{E2}$, followed some time later by $\mathbf{Y} = \mathbf{X} - \mathbf{E1}$, the substitution $\mathbf{Y} = \mathbf{E2}$ may be made. Moreover, if **X** is never used elsewhere, if there are no side effects associated with its computation, and if neither **X** nor **E1** are re-defined between the first and second statements, then **X** need not be computed at all.

2.2. The Fortran Standard. The Fortran standard permits a compiler to perform any "mathematically equivalent" transformations. Section 7.1.7.1 of [8], states: "It is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise". Note 7.19 further states that Section 7.1.7.1 applies to "...all expressions". Section 7.1.7.3 states that in place of performing the specified operations in a given mathematical expression, "...the processor may evaluate any *mathematically equivalent* expression, provided that the integrity of parentheses is not violated". (Emphasis added.) Section 7.1.7.3 goes on to define *mathematically equivalent* thusly: "Two ex-

pressions are *mathematically equivalent* if, for all possible values of their *primaries*[3], their mathematical values are equal". The standard does not define "possible values". If the set of "possible values" is the same as the domain of an expression, *mathematically equivalent* in the Fortran Standard is the same as *mathematically identical.*

The requirement to preserve the "integrity of parentheses" is motivated by the desire to permit a programmer to specify the exact order in which operations are to be performed. This requirement can be viewed from both an operational and from a mathematical perspective. For example, $\mathbf{Y = X/(A + B)}$ and $\mathbf{Y = X/A + B}$ are both mathematically and operationally different. Whereas, $\mathbf{Y = X + (A + B)}$ and $\mathbf{Y = X + A + B}$ are mathematically identical and operationally different. When mathematically identical transformations are permitted, only those parentheses that change the mathematical definition of an expression must be observed. However, if identity transformations are *not* permitted, strict adherence to the specified operations is required.

2.3. Control of Identity Transformations. In production programs, the benefits from improved run-time performance resulting from compile-time transformations can be critically important. There are at least two situations in which it is necessary for a compiler to be strictly operational and to perform no identity transformations: debugging and testing. Therefore, it is necessary to specify whether or not identity transformations are permitted, using a pragma[4], a global variable, or a command-line flag.

2.4. Inter-Procedural Analysis. If different compilation units are compiled at the same time, or if inter-procedural information is otherwise made available to the compiler, algebraic transformations *across* procedures can be performed. For example, consider the two functions:

```
FUNCTION  FOO(X, Y)
   FOO = X + Y
RETURN
END
```

[3] *For the purposes of this paper, a primary is an argument of the underlying function that the mathematically equivalent expression in question operationally defines.*

[4] A *pragma* is a statment to the compiler about how to process subsequent statements. For example, the pragma

C$PRAGMA COMPILE_TIME_TRANSFORMATIONS =< ON, OFF >,

can be used in the present context to set the mode of compilation.

(1)

```
FUNCTION  BAR(X, Y)
   BAR = X – Y
RETURN
END
```

In place of **X** = **FOO**(**A**, **B**), followed by **Y** = **BAR**(**X**, **A**), a compiler could substitute **Y** = **B**.

The Fortran standard contains a "caveat emptor" warning in Section 7.1.3: "However, mathematically equivalent expressions of numeric type may produce different computational results." The interval guarantee of containment permits these "different results" to be used to obtain sharper intervals than otherwise would be possible.

3. Identically Equal Interval Expressions

Two identically equal point variables are interchangeable. When two interval variables *coincide*, their endpoints are equal, but they are not necessarily identically equal. A simple example of the distinction between coincidence and identical equality of two intervals is the "dependence problem", which is the source of the conclusion that for any non-degenerate interval, X, $X - X \neq 0$. When viewed from the operational perspective at run-time, the code **X** – **X** is no different from **X** – **Y**, when **X** and **Y** coincide.

3.1. The Four Definitions of Interval Equality. There are four possible definitions of interval equality, three of which are required to precisely define concepts in this paper.

Let $X = \{x \mid a \leq x \leq b\} = [a,\ b]$ and $Y = \{y \mid c \leq y \leq d\} = [c,\ d]$. The four types of interval equality are presented in the following table:

Type of Interval Equality	Symbol[5]	Definition
Identical	$X \equiv Y$	$x = y\ \forall x \in X$ and $y \in Y$
Certain	$X\ .CEQ.\ Y$	$a = b = c = d$
Set or Coincidence	$X\ .SEQ.\ Y$	$a = c$ and $b = d$
Possible	$X\ .PEQ.\ Y$	$b \geq c$ and $a \leq d$

[5] In the case of ***Certain***, ***Set***, and ***Possible*** equality, as well as the other order relations, the notation *.Cop.*, *.Sop.*, and *.Pop.* as well as .**Cop**., .**Sop**., and .**Pop**. is used to denote the three varieties of relational operators; where *op* = {*LT*, *LE*, *GT*, *GE*, *EQ*, or *NE*}, and **op** = {**LT**, **LE**, **GT**, **GE**, **EQ**, or **NE**}. The ***Possibly*** operators are not used herein and are included only for completeness.

Identical Equality. Two interval variables are *identically equal* if they *coincide*[6] *and* they are *dependent*. Coincidence is the consequence of X and Y having the same endpoints. *Dependence* is the consequence of x being equal to y $\forall x \in X$ and $y \in Y$. Because dependence cannot be determined from interval endpoints alone, testing for identical equality at run-time is not possible. When viewed from a mathematical perspective at compile-time, however, the assignment statement $\mathbf{X} = \mathbf{Y}$ can be interpreted to mean identical equality, because the symbolic names of the variables are known to the compiler.

Unlike point constants, interval constants do *not* share all the properties of interval variables. In particular, a non-degenerate interval constant cannot be identically equal to *anything*, including itself. The interval constant $[a,\ b] \not\equiv [a,\ b]$, if $b > a$, because there exists no scalar variable in an interval constant with which a dependence can be formed. Two interval constants can be *certainly equal* or *set equal,* see below.

Certain Equality. Two intervals are *certainly equal* if they are degenerate and equal. Certain equality can be determined from interval endpoints and therefore can be tested at run-time. Certain equality is the only case in which identical equality can be determined by comparing interval endpoints at run-time. Two interval constants are identically equal, for example, when $[a,\ a]\ .CEQ.\ [a,\ a]$. At run-time, only degenerate intervals can be shown to be identically equal. However, at compile-time, $(< \mathbf{A} >)$ can be treated as $[a,\ a]$ even though the point, $\mathbf{A}$, may not be machine-representable. An illustration of this common situation is: $\mathbf{A} = \mathbf{0.1}$ on a binary computer.

Set Equality or Coincidence. Two intervals are *set equal,* or coincide, if they represent the same set of points. This is true if their respective endpoints are equal. Coincidence of two intervals results from an assignment of interval values during program execution. After the assignment of the interval endpoints of X to Y in the statement $\mathbf{Y} = \mathbf{X}$, $X\ .SEQ.\ Y$, but x and y remain independent. Note that even from the mathematical perspective at compile-time, the code $\mathbf{X} = (< \mathbf{A},\ \mathbf{B} >)$ is an assignment of value.

The following example illustrates the fact that set equality and identical equality are different. Let $X = [1,\ 2]$ and define: $Y_1 = X^2$ and $Y_2 = 3X - 2$. While Y_1 and Y_2 do coincide (because $Y_1 = [1,\ 4]$ and $Y_2 = [1,\ 4]$, and thus, $Y_1\ .SEQ.\ Y_2$), clearly $Y_1 \not\equiv Y_2$.

3.2. Interval Equality Summary. The following table summarizes the notation used to distinguish among the different types of interval equality used in this paper.

[6] They have the same endpoints.

Code	Compile-time Definition	Run-time Definition	Run-time Result
$\mathbf{X} = \mathbf{Y}$	$x \equiv y$	$x := y$	**X .EQ. Y**
$\mathbf{X} = \mathbf{Y}$	$X \equiv Y$	$X := Y$	**X .SEQ. Y**
$\mathbf{X} = (< \mathbf{A},\ \mathbf{A} >)$	$X \equiv [a,\ a]$	$X := [a,\ a]$	**X .CEQ.** $(< \mathbf{A},\ \mathbf{A} >)$
$\mathbf{X} = (< \mathbf{A} >)$	$X \equiv [a,\ a]$	$X := [a,\ a]$	**X .CEQ.** $(< \mathbf{A},\ \mathbf{A} >)$
$\mathbf{X} = (< \mathbf{A},\ \mathbf{B} >)$	$X := [a,\ b]$	$X := [a,\ b]$	**X .SEQ.** $(< \mathbf{A},\ \mathbf{B} >)$
$\mathbf{X} = (< \mathbf{A} >)$	$X \equiv [a,\ a]$	$X := [a,\ b]$	**X .SEQ.** $(< \mathbf{A},\ \mathbf{B} >)$

The second column can be used to resolve any apparent ambiguity regarding which variables are points and which are intervals. For example, in the first row, second column, x and y are both lower case. This means that in the first row, first column, **X** and **Y** are both point variables. In the second row, they are all upper case, and therefore, intervals. There are two rows containing $\mathbf{X} = (< \mathbf{A} >)$ in the first column. In the first, **A** is machine representable. In the second, it is not.

4. Identically Equal Interval Expressions - Continued

To continue the development of identically equal interval expressions, the most important theorem of interval analysis is needed. The theorem's importance and identifying it as *the* "fundamental theorem of interval analysis" is credited to Louis Rall [6] by Eldon Hansen [2]. This theorem was first proved by Ramon Moore [4].

Theorem 1. *An inclusion monotonic interval extension*[7]*,* $f(X)$*, of a real function,* $f(x)$*, bounds the range of the function over its argument interval. That is,* $f(X)$ *has the following property:*

$$f(X) \supseteq \{f(x) \mid x \in X\}. \tag{2}$$

The fundamental theorem leaves unspecified what happens if X contains points outside the domain of f. Letting D_f denote the domain of f, there are two obvious conventions:

Convention a) Leave $f(x)$ undefined if $X \not\subseteq D_f$; or,

[7] $F(X) \equiv f(X)$, is an interval extension of $f(x)$ if $F(x) = f(x)\ \forall x$. That is, the interval extension equals the function when evaluating the extension over a degenerate interval. $f(X)$ is inclusion monotonic if $f(X) \supset f(Y)\ \forall X \supset Y$.

Convention b) Define:

$$f(X) \supseteq \{f(x) \mid x \in X \cap D_f\}, \text{ or} \tag{3a}$$

$$f(X) \supseteq \bigcup_{i=1}^{n} \{f(x) \mid x \in X_i\}; \text{ where } \bigcup_{i=1}^{n} X_i = X, \text{ and } X_i \subseteq D_f. \tag{3b}$$

Convention a) requires that $X \subseteq D_f$. In this case, any interval argument of a function must be within the function's domain, or an exception is raised. For example, both $\sqrt{[-1,\ 1]}$ and $[-1,\ 1]/[-1,\ 1]$ are undefined.

Convention b) only requires that $X \cap D_f \neq \phi$. In this case $\sqrt{[-1,\ 1]} = [0,\ 1]$, and $[-1,\ 1]/[-1,\ 1] = [-\infty,\ \infty]$. However, $\sqrt{[-2,\ -1]} = [-1,\ 1]/[0,\ 0] = \phi$, the empty interval, which can be denoted by $[\infty,\ -\infty]$. It is convenient to associate convention a) with the operational perspective and convention b) with the mathematical perspective.

Using the definitions of interval equality at run-time summarized in Section 3.2 and employing the operational perspective, $X - X \neq 0$, because $X - X$ is operationally the same as $X - Y$, given only that $X\ .SEQ.\ Y$. When performing the assignments $X := [a,\ b]$ and $Y := [c,\ d]$, only the *endpoints* of X and Y are defined within which the variables x and y are contained. Interval subtraction is operationally defined:

$$X - Y = \{x - y \mid x \in X,\ y \in Y\} = [a - d,\ b - c]. \tag{4}$$

Now suppose $X\ .SEQ.\ Y$. Nothing changes, except that:

$$X - Y = [a - b,\ b - a] \tag{5}$$

can be written in place of (4). It follows at once that:

$$[a,\ b] - [a,\ b] = [a - b,\ b - a] \neq 0. \tag{6}$$

Now consider $X - X$ from the mathematical perspective at compile-time. From the fundamental definition of the interval extension of the real function, $f(x) = x - x$:

$$f(X) = X - X = \{x - x \mid x \in X\} = 0\ \forall X. \tag{7}$$

If $\mathbf{0}$ can be substituted for $\mathbf{X} - \mathbf{X}$ at compile-time, both run-time performance and interval sharpness will be improved.

Now consider X/X. Just as it is desirable to substitute $\mathbf{0}$ for $\mathbf{X} - \mathbf{X}$, so it is to substitute $\mathbf{1}$ for $\mathbf{X}/\mathbf{X}$. Define the function $f(x) = x/x\ \forall x \in D_f = [-\infty,\ \infty] - 0$. That is, the domain of f is the real line, except for the single point at the origin

since $f(x) = 1\ \forall x$ except when $x = 0$. Therefore, the interval extension $f(X) = 1\ \forall X$ except when $X = [0,\ 0]$, in which case $f([0,\ 0]) = \phi$, because $[0,\ 0] \cap D_f = \phi$. In other words, $[0,\ 0]$ is strictly outside the domain of f. Any combination of arguments that results in an attempt to evaluate a function outside its domain is a bug.

Interval arithmetic provides a way to verify that no function arguments are strictly outside any function's domain. If no exceptions are raised when mathematical transformations are prohibited, this *proves* that no exceptions can be raised anywhere inside the sub-domain defined by the interval arguments. This means that within this sub-domain, no function argument can be outside the function's domain. Either as a consequence of analysis or as the result of empirical testing using interval arithmetic, the prudent programmer will include tests to verify that arguments are never strictly outside the domain of any functions.

The only case in which substituting **1** for $\mathbf{X}/\mathbf{X}$ at compile-time hides and undefined result is when $\mathbf{X} = [\mathbf{0},\ \mathbf{0}]$. There are other cases in which large parts of the real line can be outside the domain of a function. For example, consider $f(x) = \sqrt{x}/\sqrt{x}$. Then $D_f = [0,\ \infty]$.

It is difficult to conceive of a case in which substituting **1** for $\mathbf{X}/\mathbf{X}$ at compile-time might cause an incorrect result to be computed. There would be no hesitation in making the substitution with paper and pencil. Therefore:

Conjecture 1. *In a valid interval algorithm, substituting* **1** *for* $\mathbf{X}/\mathbf{X}$, *where* $\mathbf{X}$ *is a non-empty interval variable, will not cause a containment failure.*

5. Single-valuedness

Substitutions can only be made using identically equal expressions. To extend the concept of identically equal interval variables to interval functions, operators, and expressions, the concept of *single-valued intervals* is introduced. Single and multi-valued interval expressions are simply interval extensions of their underlying mathematical *functions* and *relations*, respectively.

5.1. Interval Extensions of Mathematical Functions and Relations. An interval function is a mapping of sets of points in its domain to sets of points in its range. (In the mathematics of points, an interval function is a mathematical relation.[3] To be a point-function, a relation must have only one value in its range for each value in its domain. In other words, point-functions are single-valued relations.)

Definition An interval function is an *interval extension* of a point function if it is equal to the point function when its interval arguments are all degenerate,

or points [4]. That is, letting F symbolize the interval extension of the point function, f:

$$F(x_1, ..., x_n) = f(x_1, ..., x_n); \tag{8}$$

for all x_i, $(i = 1, ..., n)$.

Thus, any interval extension of a point *function* is *single-valued*, because the underlying point *function* is necessarily single-valued. A second interpretation of a *single-valued interval* is: Each occurrence of a single-valued interval can be treated as one mathematically identical single value. Whereas each occurrence of a multi-valued interval is a distinctly separate and independent value.

The interval extension of a point *relation* is defined relative to its underlying point relation in precisely the same way that the interval extension of a point function is defined relative to its underlying point function. That is:

Definition If $g(x)$ defines a multi-valued relation between x and $g(x)$, then $G(X)$ is an *interval extension* of $g(x)$, if $G(x) = g(x)\ \forall x$

Inclusion monotonicity carries over from functions to relations as well:

Definition The interval extension of a relation is *inclusion monotonic* if $g(X) \supset g(Y)\ \forall X \supset Y$.

Following the same steps used to prove the fundamental theorem of interval analysis for functions, the theorem can be extended to include interval extensions of point relations:

Corollary 1. *The inclusion monotonic interval extension of a relation bounds the set of values that can be returned by the relation. That is, if $g(x)$ defines a multi-valued relation between x and $g(x)$, and if the interval extension, $g(X)$, is inclusion monotonic, then:*

$$g(X) \supseteq \{g(x) \mid x \in X\}. \tag{9}$$

Consider the relation $y \in [a,\ b]$. In this case, the set of possible values of y is the interval $[a,\ b]$. In the special case that $a = b$, the expressions $y \equiv a$ and $y := a$ are indistinguishable, as there is no difference between assignment of value and identical equality of points. Using interval notation, the relation $y \in [a,\ b]$ is the same thing as $Y := [a,\ b]$, because the interval Y represents the set $\{y \mid y \in Y\}$. The central idea is that there is no underlying *function* of which $Y := [a,\ b]$ is an interval extension.

In other words, the interval constant $[a,\ b]$ is not single-valued unless $a = b$. Every occurrence of $[a,\ b]$ is the same as a distinct and independent variable.

Consider the relation $y \in [a,\ b]x$. The set of possible values of y depends on the variable x. As with the relation $y \in [a,\ b]$, unless $a = b$, $y \in [a,\ b]x$ is not a function. Using interval notation, $Y := [a,\ b]x$. If $x \in X$, it is also possible to write the interval extension: $Y := [a,\ b]X = \{[a,\ b]x \mid x \in X\}$. Again, although Y depends on X and y depends on x, $y \in [a,\ b]x$ is a relation not a function, $[a,\ b]X$ is not a single-valued interval expression, and it is incorrect to write either: $Y \equiv [a,\ b]x$, $Y \equiv [a,\ b]X$, or $Y \equiv [a,\ b]$.

If **EMV** is the code for any multi-valued expression, the assignment statement **Y = EMV** represents an *assignment*, both at compile and at run-time. Thereafter at compile-time, **Y** is a single-valued interval variable. This is *not* a contradiction.

An interval operator is simply the interval extension of the corresponding real operator. If the real operator (or function of two or more variables) is single-valued, then so is its interval extension.

Any interval expression can be viewed as a mathematical definition of the function (or relation) of all the interval variables contained in the expression. Only if the relation is a function, is the corresponding interval expression single-valued. Only single-valued expressions can be identically equal. Let E_{MV} and E_{SV} represent arbitrary multi-valued and single valued expressions, respectively. Then:

$$\begin{aligned} E_{MV} - E_{MV} &\neq 0, \text{ however} \\ E_{SV} - E_{SV} &= 0. \end{aligned} \tag{10}$$

5.2. Properties of Single-valued (SV) and Multi-valued (MV) Interval Expressions . The same logic used in Section 5.1 can be used to establish the following properties of single-valued (SV) and multi-valued (MV) interval expressions:

1. Interval variables are SV.
2. Degenerate interval constants are SV.
3. Non-degenerate interval constants are MV
4. Interval expressions containing only SV sub-expressions are SV.
5. Interval expressions containing at least one MV sub-expression are MV.
6. Each occurrence of a MV interval expression is a distinct and independent interval value.
7. Each occurrence of the same SV interval expression is the same identical mathematical quantity.

6. Mathematical Identities

Mathematical identities take the form $f(x) = c \ \forall x \in D_f$, where c is a constant. Examples include:

$$\begin{aligned} x - x &= 0 \ ; \\ x/x &= 1 \ ; \text{and} \\ \sin^2 x + \cos^2 x &= 1. \end{aligned} \tag{11}$$

The validity of the interval extension of a mathematical identity depends on two conditions: all the identity's interval arguments must be single-valued; and no interval argument can be strictly outside the identity's domain. This result is a direct consequence of the definition of an interval extension. Consequently, any mathematical identity can be applied at compile-time to single-valued interval expressions. For an arbitrary single valued interval expression, E_{SV}, and a mathematical identity, $f(x) = c$, the generalization of Conjecture 1 to single-valued interval expressions is:

Conjecture 2. *If $f(x) = c \ \forall x \in D_f$ is a valid mathematical identity and* **ESV** *is a non-empty single-valued interval expression, then in a valid interval algorithm containing* **F(ESV)**, *substituting* **C** *for* **F(ESV)** *cannot cause a containment failure in the algorithm.*

7. Mathematically Identical Expressions

When computing any interval expression in which each variable appears only once, the result is an exact bound on the range of the function defined by the expression. When any variable appears more than once, the result may not be as sharp as possible. For example, if E, E_x, and E_y are arbitrary interval expressions, then:

$$EE_x + EE_y \supseteq E(E_x + E_y). \tag{12}$$

With the goal of improving the sharpness of computed results, under what conditions can a compiler perform an equivalence transformation such as that implied by (12)? That is, when can a compiler substitute $\mathbf{E} * (\mathbf{EX} + \mathbf{EY})$ for $\mathbf{E} * \mathbf{EX} + \mathbf{E} * \mathbf{EY}$ at compile-time? The answer is: whenever E is single-valued. Take a simple example of a case when E is not single-valued. Substituting $[a,\ b]$ for E in (12):

$$[a,\ b]E_x + [a,\ b]E_y. \tag{13}$$

Expression 13 is *not* mathematically identical to:

$$[a,\ b](E_x + E_y). \tag{14}$$

The reason is that $[a,\ b]E_x + [a,\ b]E_y$ is the same as the result of assigning $U := V := [a,\ b]$, followed by:

$$UE_x + VE_y. \tag{15}$$

If $[a,\ b](E_x + E_y)$ is substituted for expression (15), containment cannot be guaranteed, thereby violating the fundamental requirement of interval arithmetic.

Thus, the normal rules of algebra can be used to construct mathematically identical interval expressions, *provided* that substitutions only involve single-valued expressions.

8. Intersections of Mathematically Identical Expressions

If there is no single uniformly sharpest representation for a given interval expression, the sharpness of interval results can still be increased. This is done by using the intersection of the intervals obtained from evaluating different mathematically identical expressions. For example, consider: $E(E(E-1))$, where E is an arbitrary single-valued interval expression. There is no mathematically identical interval expression containing only one occurrence of E. Nevertheless, there are many mathematically identical expressions, such as: $E((E-1/2)^2 - 1/4)$.

Let E_j be interval extensions of mathematically identical expressions, for $j = 1, ..., n$. Then at compile time, the code to compute $\bigcap_{j=1}^{n} E_j$ at run-time can be generated.

A number of points deserve mention:

1. While *locally* more costly to compute alternative expressions and intersect their results, global run-time performance actually may be improved.

2. Employing a subroutine to perform the optimization necessary to compute the sharpest possible bounds on the range of any interval expression may improve overall run-time performance. The only requirement is that containment be guaranteed.

3. If parallel computing hardware is available, computing alternative expressions can be done in parallel.

9. User-defined Functions and Operators

For intrinsic functions and operators[8], information can be included in a compiler about which interval extensions are single-valued and which are multi-valued. For example, the only Fortran intrinsic interval functions that are multi-valued are:

[8]Intrinsic functions and operators are part of a language and therefore their properties can be "known" to the compiler.

- **INTERVAL**(**A**, **B**), and
- (< **A**, **B** >), provided that **B** > **A**.

To program a compiler to automatically identify single-valued user-defined functions and operators, it is sufficient to identify sub-expressions as SV or MV, and to follow the rules given in Section 5.2. In Fortran, only interval constants are inherently MV quantities. Even the **RANDOM_NUMBER** subroutine returns an interval variable, which is by definition, single-valued.

In the absence of language support to declare SV user-defined functions and operators, compile-time optimizations can be restricted to interval expressions containing only interval variables, and single-valued *intrinsic* functions and operators. If a programmer can declare, or a compiler can automatically identify, single-valued user-defined interval function sub-programs and operators, *all* expressions can be candidates for automatic compile-time optimization.

Without language or compiler support for interval single-valuedness, but with extra programming work, the compiler can be provided the information needed to identify mathematically identical expressions even if they contain user-defined operators and functions. Consider an expression, **EU**, that contains user defined function subprograms and/or operators. A compiler cannot substitute **0** for the expression **EU** − **EU**. Because all interval variables are single-valued, zero can be substituted for **X** − **X** if this expression is preceded by **X** = **EU**. This is valid even if **EU** is multi-valued. Nevertheless, automatic recognition at compile-time is always preferred.

10. Automatic Determination of Single-valuedness

Section 5.2 contains the properties that are needed to program a compiler to identify any sub-expression as SV or MV. There are cases in which these properties are insufficient to determine that a user-defined function subprogram is single-valued. In such cases, it is desirable for a programmer to be permitted to declare any user-defined function or operator to be single or multi-valued. For example, a routine can be used to compute the interval extension of a real function. In the process, an interval may need to be constructed to bound the error in the approximation algorithm. While a compiler must conclude that such a function subprogram is multi-valued, a programmer may want to override with an explicit declaration. For testing and debugging, it must be possible to declare a single-valued function to be multi-valued.

To the list in Section 5.2, one additional property must be added:

8. A single-valued function of a non-degenerate interval constant argument is MV.

Thus, for example:

$$\exp([a,\ b])/\exp([a,\ b]) \neq 1. \tag{16}$$

11. Fortran Named Constants

In Fortran a **PARAMETER** statement can be used to declare named constants. Neither named nor literal constants can have their value changed during program execution. As with literal constants, degenerate named constants are SV. What about non-degenerate named constants? If the named constant is assigned a value from a single-valued literal interval constant, then the named constant is SV, even though it may not be degenerate. However, if the named constant is assigned its value from an explicitly multi-valued interval constant or constant expression, then the result is MV. An example of the former is:

$$\textbf{PARAMETER PI} = (< \textbf{3.1415}... >). \tag{17}$$

An example of the latter is:

$$\textbf{PARAMETER C23} = (< \textbf{2.0},\ \textbf{3.0} >). \tag{18}$$

12. Conclusion

The distinction between operational and mathematical perspectives has been introduced and used to justify the application of mathematically identical transformations at compile-time. The concept of single-valued interval expressions has been defined to extend the concept of identical equality from variables to expressions. Optimizing transformations can be automatically applied to SV interval expressions. Interval expressions containing user-defined operators and function sub-programs can be declared and/or automatically identified as single or multi-valued. Automatic identification of possible compile-time transformations is facilitated by judiciously introducing intermediate interval variables, which are always single-valued.

Compile-time optimizations of interval code will produce a significant increase both in interval sharpness and in run-time performance. Interval arithmetic without compile-time transformations, can be used to prove that no function arguments can ever be strictly outside a function's domain.

References

[1] R. Baker Kearfott et al. A Specific Proposal for Interval Arithmetic in Fortran. Technical report, X3J3, March 1996.

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

[3] Glen James and Robert C. James. *James and James Mathematics Dictionary.* Van Norstrand Reinhold, 11 Fifth Avenue, New York, New York 10003, fourth edition, 1976.

[4] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1966.

[5] Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic, ANSI/IEEE STD 754-1985. Technical report, New York, 1985.

[6] L. B. Rall. *Computational Solution of Nonlinear Operator Equations.* Wiley, New York, 1969.

[7] G. William Walster. *Stimulating Hardware and Software Support for Interval Arithmetic*, pages 405–416 in R. B. Kearfott and V. Kreinovich, "Applications of Interval Computations". Dordrecht, The Netherlands, 1996.

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