

Compiler Support of Interval Arithmetic with Inline Code Generation and Nonstop Exception Handling

G. William Walster

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
1 (800) 786.7638
1.512.434.1511

Copyright 2002 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, EJB, EmbeddedJava, Enterprise JavaBeans, Forte, iPlanet, Java, JavaBeans, Java Blend, JavaServer Pages, JDBC, JDK, J2EE, J2SE, and Solaris trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2002 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, Californie 94303 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, EJB, EmbeddedJava, Enterprise JavaBeans, Forte, iPlanet, Java, JavaBeans, Java Blend, JavaServer Pages, JDBC, JDK, J2EE, J2SE, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REpondre A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Contents

Introduction	1
Implemented Interval Endpoints.....	3
The Empty Interval and the Domain of Enclosures	5
Enclosures at Accumulation Points	5
Alternative Design Choices.....	7
Conclusion	8
Appendix A. References	9

Compiler Support of Interval Arithmetic with Inline Code Generation and Nonstop Exception Handling

Introduction

To guard against producing erroneous results, IEEE 754 floating-point exception handling is used in situations where no single floating-point number is the unambiguously correct result. See [?]. In extended real interval systems, no exception handling is required for undefined point events. Point operations that produce any possible set of real results can be enclosed in extended intervals with finite or infinite endpoints. For the theoretical foundation of extended interval arithmetic, see [?]. For a detailed description of the “Simple” interval system and its implementation, see [?] and [?], respectively. For the Sun Microsystems Inc. Forte™ Developer 6 Fortran 95 compiler implementation, see [?]. Hereafter, reference is made simply to the f95 compiler.

Whatever interval system is devised and however it is implemented, the fundamental containment constraint of interval arithmetic must be satisfied, see [?]. That is, the set of all possible values of any expression must be contained in the expression’s interval evaluation. Subject to this single requirement, any design optimization criteria can be chosen. Some obvious candidates include:

- Runtime speed, given existing hardware
- Narrow width interval results
- Complete and transparent (easy to write and read) language syntax and semantics
- Compilation speed
- Source-code compatibility as new features and/or hardware support are introduced

With intrinsic compiler support, inlining is a practical way to produce optimized code that executes faster than function calls generated by a pre-compiler, C++ class, or Fortran 90 module.

The minimum width of any interval result is the interval hull of the containment set (or topological closure) of the computed expression, see [?] for an overview. Any narrower result is guaranteed to produce a containment failure. Any wider result is not as narrow as it can be. The containment set of any interval expression is always defined. This is true, even when interval arguments of the expression include values for which the expression is undefined in the real or extended real systems.

Language support for interval code should be easy to use.

Flexibility to choose between compile-time and run-time should be possible. Flexibility is needed so fast compilation is possible when debugging, but fast production code can be generated at the expense of compile time.

Finally, as new interval language features are introduced and new interval-specific hardware support is made available, existing source code must continue to compile and produce correct interval results.

Some of the above goals conflict. For example, runtime speed, compile-time speed and narrow width all interact to varying degrees. Two fundamentally different approaches can be taken to cope with the resulting complexity:

- Make few design choices, but rather provide users with options from which to choose, or
- Make many design choices by anticipating those option combinations that interval code developers will want

Providing many options is prudent if there is neither consensus about, nor understanding of, the most desired design choices. Options are expensive to test, document, and maintain. Choosing among a plethora of options can also be a burden for users. The resources spent on unnecessary options can be better spent on features that developers prefer. Moreover, minimizing the alternative options compilers must handle can lead to faster executing code. In particular, fewer options make in-line-code more practical.

An advantage of closed interval systems is that exceptions are logically impossible. Closed interval systems have no restrictions on the domain of argument intervals. Therefore interval-specific exception handling machinery is unnecessary. Where possible, the existing IEEE-754 data structures and operation definitions are used to make initial software implementations as fast as possible, see [?]. The interesting challenge is to adapt IEEE data structures and operations to cope with needed interval constructs such as the empty and entire intervals. The empty interval is the empty set, and arises naturally in a number of ways, including the intersection of two disjoint intervals. The *entire interval* is, the interval $[-\infty, +\infty]$. The only overriding constraint of any interval system implementation is that the internal evaluation of any expression (function or relation) must produce a valid interval result that encloses the expression's containment set, see [?].

Implemented Interval Endpoints

Not all possible interval endpoints must be implemented to have a closed interval system. Only the entire interval, $[-\infty, +\infty]$, is logically required. A containment failure can always be avoided by returning the entire interval, even if this is not the narrowest possible result. Representing the empty interval is a high priority, as is representing single infinite endpoints. Two choices that remain to be made, include the interpretation of:

- Zero interval endpoints
- Infinite interval endpoints

Only the “Simple” system is implemented in the first release of support for intervals §95. The fact that the implemented interval language support is *opaque*, means that the structure of internal hardware representations is not needed either by developers

or end users. As a consequence, incremental increases in interval-specific hardware support can be transparently introduced while preserving source code and ASCII file compatibility. Necessarily, unformatted file compatibility cannot be maintained when interval data structures change.

Zero Interval Endpoints

Three possible interpretations exist for an interval with a zero endpoint:

1. An interval with a closed endpoint – that is, the unsigned point at zero
2. An interval with an open negative or positive zero endpoint, excluding the unsigned point at zero
3. The result of IEEE underflow, which is a signed closed interval endpoint that is bounded away from zero

The first alternative is necessary to represent zero interval endpoints and is the only zero-valued endpoint implemented in the “Simple” system. In this system, the sign of IEEE zero is ignored.

Infinite Interval Endpoints

Four possible interpretations exist for intervals containing an endpoint with absolute value greater than can be represented with finite IEEE floating-point numbers:

1. An interval with a closed endpoint that is a signed value of $+\infty$, or $-\infty$, see [?] for the topological justification of this interpretation
2. An interval with a closed endpoint that is the projective or unsigned infinity, which is the union of the signed infinities
3. An interval with an open negative or positive infinity, excluding both signed infinities and projective infinity
4. The result of IEEE overflow, which is a signed closed interval endpoint that is bounded away from infinity

Signed infinities are necessary to represent the entire interval, unless some special representation is used for the interval $[-\infty, +\infty]$. Signed infinities are the only infinite endpoints implemented in the “Simple” system.

The “Simple” System

The “Simple” system is designed to make a software implementation easy to develop and efficient to execute. Supporting additional interval endpoint interpretations in the implemented system will become practical with additional interval-specific hardware support.

The Empty Interval and the Domain of Enclosures

In [?], the containment set of values that must be included in any interval result is defined. This definition is important because it makes explicit the set of values that an interval enclosure must contain when an interval argument is partially or totally outside an expression’s domain of definition. The empty interval (equivalently, the empty set, denoted \emptyset) is the containment set of an expression evaluated at a point that is bounded away from the expression’s domain. The containment set is the set of values that an interval enclosure must contain. Popova [?] also made the empty-set proposal for any points strictly outside a function’s domain of definition. In effect, Popova simply defined the set of values that the interval evaluation of a function, f , over the interval, X_0 , must contain to be:

$$f(X_0) \supseteq \text{hull}(\{z \mid z \in f(\{x_0\}), x_0 \in X_0 \cap \mathcal{D}_f\}); \quad (1)$$

where: $\text{hull}(S) = [\inf(S), \sup(S)]$ is the interval hull of the set, S ; and \mathcal{D}_f denotes f ’s domain of definition. It is assumed that $f(\emptyset) = \emptyset$. Points in braces, i.e., $\{x_0\}$, are singleton sets. An expression, f , of a singleton-set argument produces a result that is a set. See [?] and [?], for details.

Enclosures at Accumulation Points

The definition in (1) works for some functions, such as $\sqrt{\cdot}$, but leads to containment failures for others, such as division by zero. To prevent containment failures, limiting values must be included as arguments approach values on the boundary of expression domains. In [?], the set of values that must be included in the interval evaluation of any expression is proved to be the topological closure of the expression. The *closure* of an expression includes all possible limiting values of the expression as arguments approach the value in question. For example, with division by zero, all possible sequences $\left(\frac{1}{x_j}\right)$, given that $\lim_{j \rightarrow \infty} x_j = 0$, must be considered. In particular, if $x_j = \frac{-1}{j}$, or $x_j = \frac{1}{j}$, then $\lim_{j \rightarrow \infty} x_j = 0$. However, if $x_j = \frac{-1}{j}$, $\lim_{j \rightarrow \infty} \frac{1}{x_j} = \lim_{j \rightarrow \infty} (-j) = -\infty$. Similarly, if $x_j = \frac{1}{j}$, then $\lim_{j \rightarrow \infty} \frac{1}{x_j} = +\infty$. Therefore the containment set of the expression $f(x) = \frac{1}{x}$, evaluated at $x_0 = 0$ is denoted

$$\begin{aligned} \text{cset}(f, \{x_0\}) &= \overline{f}(\{x_0\}) & (2) \\ &= \{-\infty, +\infty\}; & (3) \end{aligned}$$

where $\overline{f}(\{x_0\})$ is the usual notation for the closure of the expression, f , at the point x_0 .

The containment set is the smallest set of values that eliminates the possibility of subsequent containment failures. In [?], containment sets and expression closures are proved to be identical. Over any arbitrary set, $\{X_0\}$ ¹, the containment set of the function, f , is simply the union of all possible containment sets evaluated at every possible point in the set, $\{X_0\}$. Therefore,

$$\text{cset}(f, \{X_0\}) = \{z \mid z \in \overline{f}(\{x_0\}), x_0 \in X_0\}.$$

Notice that because expression closures are always defined, there is no restriction on the arguments of any expression or its containment set. These results remove any restrictions on the domain of definition of interval enclosures. Therefore, no exceptional events exist in the compiler implementation of closed interval systems.

¹ All sets are enclosed in braces $\{\cdot\}$ to distinguish them from intervals, which are denoted using unbracketed upper case letters.

The definitions of closures and containment sets carry over to expressions of more than one variable. For additional details, see [?] and [?].

Alternative Design Choices

As discussed in Section , points on the boundary of open domains of expressions, the expression’s closures must be included to avoid containment failures. However, when expression arguments are strictly outside the domain of definition, alternative interpretations and actions are possible, including simply returning the empty interval. Possibilities include:

1. if values remain undefined:
 - a abort the program or otherwise raise an interval-specific exception;
 - b return a non-default IEEE $\mathbf{NaN}_{\mathbf{NaI}}$, representing a error condition, where the subscript \mathbf{NaI} is a mnemonic for “Not-an-Interval”, or;
 - c return the empty interval.
2. return the entire interval, \mathbb{R}^* , which is the entire extended real line, $[-\infty, +\infty]$; or
3. return $\mathbf{NaN}_{\mathbf{C}^*}$, representing the entire extended complex plane.

Option 1a is possible, but has the disadvantage that programers must write defensive code to prevent programs from aborting or raising exceptions when this is inappropriate. Option 1b is the equivalent of \mathbf{NaN} for points and will propagate through every operation and function evaluation. While programs returning $\mathbf{NaN}_{\mathbf{NaI}}$ will not abort, a returned $\mathbf{NaN}_{\mathbf{NaI}}$ implies that a programing error has been made. While a logic or programming error may have been made when an interval argument is partially or totally outside the domain of an expression, without program context, it is impossible to know whether any kind of error has taken place. See [?]. Therefore, the only condition under which it is conceivable that a $\mathbf{NaN}_{\mathbf{NaI}}$ can be justified is if an invalid interval is encountered. An invalid interval is one in which the infimum is greater than the supremum. Because invalid intervals can only arise as the result of an input operation or the explicit creation of an interval using the `INTERVAL`

constructor, invalid intervals are handled in exactly the same way as an attempt to read a character string into a floating-point variable. Rather than introduce a separate internal representation for this event, either \mathbb{R}^* is returned or processing is aborted. In §95 the former is done if **ERR=**, or **IERR** are defined in an interval input operation, or the `INTERVAL` constructor is used. Processing is aborted with an appropriate error message, otherwise. Option 1c is implemented in §95. Option 2, while not a containment failure, implies that some or all of the points on the real line may need to be contained in an enclosure. For a real system, this is not so. In the absence of an error, returning the empty interval is the correct and sharp result. Option 3 implies that some or all of the points in the complex plane may need to be contained in an enclosure. This makes sense in an extended *complex* interval system. However, the implemented system implemented is explicitly *real*. There is no requirement to introduce complex intervals as there is no requirement to include complex values in any *real* enclosure.

For additional justification for returning the empty interval when expression arguments are outside domains of definition, see [?].

Conclusion

Implementation design choices have been made to provide an optimal interval software development environment given resource and environment constraints. The goal of describing the logic used to choose among alternative designs is to promote understanding and acceptance of the implemented features.

References

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

1 (800) 786.7638
1.512.434.1511

<http://www.sun.com>

April 2002