Reliable Computing, **2**, 2, 1996, pp. 161-166.

Interval Operations Involving NaNs

Evgenija D. Popova

0 Introduction

Ten years ago IEEE standard [1] for floating-point arithmetic became official. Each IEEE floating-point format supports: its own set of finite real numbers, $\pm\infty$, two distinguished values $+0$ and -0 and a set of special values called NaNs (Not-a-Number). Arithmetic operations include operations on numeric, non-numeric or mixed operands in four rounding modes. A number of exceptional situations may arise during numerical computations. Every exception, when it occurs must raise a flag that a program may subsequently sense and/or take a trap intended to handle the detected exceptional condition. The mandatory default response to the exceptional situations is not to trap on them, but to compute and deliver to the destination a default result, specified for each possible exception.

Now, an increasing number of computers and software feature IEEE arithmetic. Contrary to the programs written before IEEE Standard became official, programs which are written to be used under IEEE arithmetic should be prepared to expect any exceptions that can arise and deal with them properly. Some recent works [3] show that algorithms working uniformly and robustly across rather different systems and languages are a lot easier to design and usually more efficient if they rely on non-trapping exception handling paradigm.

Most recent interval arithmetic implementations [8] are in Standard conforming environment. Recently, some specifications of Basic Interval Arithmetic Subroutines (BIAS) appeared [2, 5] showing a movement toward standardization of the user interfaces for interval arithmetic software.

Although IEEE Standard has been intended to facilitate, between other things, the implementation of interval arithmetic [6] nowadays there are no general implementation requirements for interval arithmetic under IEEE systems. Since there is no meaning of the arithmetic operations on intervals involving NaNs their implementation is up to the implementor's option. Moreover, no agreement exists about how to deal with the exceptions arising on interval operations and no default interval response has been proposed. The emphasis in computing was traditionally on speed but we have to develop also credible and accurate programs. For a program to be credible, the result it produces must never be misleading.

Goals of this paper are to consider some algorithmic aspects of the implementation of interval arithmetic involving NaNs or signed zero (Section 1) and to propose (Section 2) a simple model of interval arithmetic exceptions and their handling in IEEE non-trapping mode facilitating thus an extension of the BIAS for IEEE systems.

1 Interval operations involving NaNs or ± 0

Here we shall point out some of the pitfalls for the interval arithmetic implementation in IEEE environment. We presume `Invalid Operation` (`IO`) trap is disabled, the IEEE system works in the default non-trapping mode.

Definition. *An interval over the set of floating-point numbers supported by an IEEE format is called unordered if its end-points compare unordered.*

According to the Standard two operands are in relation "unordered" only when at least one operand is a NaN. In addition to the `TRUE/FALSE` response an `IO` exception shall be signaled when unordered operands are compared using a predicate not involving "?" ("?" being a predicate for unordered comparison).

Proposition. *Interval operations (and functions) implemented by using floating-point comparison not involving unordered will signal* `IO` *exception on unordered and mixed type operands.*

Corollary. *Unlike scalar floating-point arithmetic where quiet NaNs propagate through arithmetic operations without precipitating exceptions, interval arithmetic multiplication and division operations, implemented by predicates not involving unordered, will signal* `IO` *exception on unordered or mixed type operands.*

Examples for interval operations satisfying the above proposition are operations multiplication and division, interval hull and intersection, all relational operations. Suppose a classical unexceptional Algorithm 1.1 for interval hull is implemented in an IEEE environment and let the hull of the intervals $[\mathtt{qNaN}, -5]$ and $[12, 16]$ have to be computed by this algorithm. Although the first comparison operation will signal `IO` exception, in non-trapping mode it will return `FALSE` as a default result. Thus a misleading result $[12, 16]$ will be produced instead of the indeterminate but more correct result $[\mathtt{qNaN}, 16]$. Same is the reason for obtaining $[-3, 6] = [-3, \mathtt{qNaN}] \times [-2, 1]$ using a classical algorithm for interval multiplication. Furthermore, different implementations of the multiplication and division operations may result in different but equally dangerous numerical results. That is why some additional programmer's effort is required to ensure a reasonable interval result. Various implementation schemes are possible: Algorithm 1.2 checks the `IO` exception flag after each floating-point comparison to detect the existence of an unordered operand. Function `exc_io()` returns `TRUE` if the `IO` status flag is raised, then the procedure `io_reset()` clears it and a quiet NaN constant is assigned to the corresponding end-point of the result. Algorithms 1.3 and 1.4 use unordered predicate to test the existence of unordered arguments instead of handling `IO` status flag. In IEEE style (`IO` is raised when NaN is created from non-NaN operands) Algorithm 1.3 prevents the occurrence of `IO` exception while Algorithm 1.4 signals `IO` on unordered operands. If a "?" predicate is not supported, an implementation may use predicates $x == x$ and $x! = x$, which do not signal `IO` and deliver `FALSE`, resp. `TRUE` on unordered arguments, or a classification function in order to account for NaNs.

Algorithm 1.1 $[x, y] = [a, b] \cup [c, d]$

```
if (a ≤ c) then x = a
else x = c

if (b ≤ d) then y = d
else y = b
```

Algorithm 1.2 $[x, y] = [a, b] \cup [c, d]$

```
if (a ≤ c) then x = a
elseif (exi_io()) then x = q_NaN
                         io_reset()
   else x = c
if (b ≤ d) then y = d
elseif (exi_io()) then y =q_NaN
                         io_reset()
   else y = b
```

Algorithm 1.3 $[x, y] = [a, b] \cup [c, d]$

```
if (a ? c) then x =q_NaN
elseif (a ≤ c) then x = a
   else x = c
if (b ? d) then y =q_NaN
elseif (b ≤ d) then y = b
   else y = d
```

Algorithm 1.4 $[x, y] = [a, b] \cup [c, d]$

```
if (a ≤ c) then x = a
elseif (a ? c) then x =q_NaN
   else x = c
if (b ≤ d) then y = b
elseif (b ? d) then y =q_NaN
   else y = d
```

Interval arithmetic implementations are so far left ambiguous about the behavior of interval operations with respect to the special elements supported by IEEE formats causing confusion and controversy insofar as programmers have to agree upon their definitions. For example, interval arithmetic in PASCAL-XSC is always trapping on operands involving NaNs and on interval division when the divisor has zero at some end-point despite the result of such operation is a mathematically well defined semi-infinite interval and infinities participate in all other interval operations. Although IEEE comparisons say $+0$ and -0 are equal, division operation is affected by zero sign; $1/(+0) = +\infty$ but $1/(-0) = -\infty$. Zero sign propagates through certain arithmetic operations according to rules derived from continuity considerations; for instance $(-2) \times (+0) = -0$, $(-0)/(-3) = +0$ and $\bigtriangledown(x - x) = -0$ for every finite real x. Let us consider the expression $[2, 3]/([0, 5]-[-2, 0]) = [2, 3]/[-0, 7]$ computed under IEEE arithmetic. The division operation will produce: $[-\infty, 3/7]$, if implemented by min/max functions; $[2/7, -\infty]$, if implemented by checking signs of the intervals and $[2/7, \infty]$, if implemented by test for zero end-point. First two completely wrong results will be due to not sensing the sign of zero while the correct result in terms of Kahan's outer intervals should be $[-\infty, -\infty] \cup [2/7, \infty]$.

Two implementing paradigms are possible with respect to the zero elements of the

IEEE system. One is the algebraic sign of zero not to be interpreted by the interval arithmetic which will lead to a simpler but restricted implementation. The other is to consider the algebraic sign of zero as specified by the Standard. This will complicate the basic interval software but will allow implementation of a wider understanding of intervals [4]. We can consider intervals with end-points zero as open or closed; for instance $[-0, 1]$ includes 0 as an internal point but $[+0, 1]$ does not. Whatever is the implementor's decision about these two paradigms, it should be followed for all interval operations.

2 Interval Exceptions and their Handling

General Principle: *Since interval operations are compound operations besides empty set intersection and division by interval containing zero as internal point interval operations themselves will signal no exceptions. All the exceptions arising on execution of an interval operation are floating-point exceptions arising on floating-point operations which compound the corresponding interval operation.*

Next we specify credible results for interval operations on unordered operands irrespective of whether `IO` will be signaled by underlying floating-point comparisons (Section 1) or not.

- The result delivered by interval operations multiplication and division should involve at least one quiet NaN as end-point on unordered or mixed type operands.
- The result delivered by the operations interval hull and intersection should be an interval with a quiet NaN at that end-point at which NaN is involved in the arguments.
- `IO` exception should be signaled on all interval relational operations and `FALSE` should be delivered as a default result if some operand is unordered.
- The result delivered by an auxiliary interval function of unordered argument should, if a floating-point result is to be delivered, deliver a quiet NaN as a result.
- `IO` exception may be signaled by an interval standard function when its argument does not belong to the definition domain of that function. The default result delivered if the exception occurs without trap may be the result of the same function of an argument which is intersection of the user-defined argument and the definition domain of the corresponding function.

The proposed scheme of interval exceptions and their handling has the advantages to: permit an undoubted and correct implementation of interval arithmetic operations and functions on IEEE arithmetic; permit a maximal closure to the interval algorithms for non IEEE arithmetic; allow full user control on the floating-point exceptional situations and their handling; permit a non-contradictory performance in both trapping and non-trapping mode; be applicable to the most extensions and generalizations of the conventional interval arithmetic.

3 Conclusion

Keeping to rigorous definitions of the operations on intervals involving NaNs and interval arithmetic exception handling will benefit the end users of interval software in being able to rely on its credible execution in IEEE environment and software developers to write portable code which uses features of the Standard. An implementation of the proposed model is provided by the PASCAL-XSC module EXI_ARI [7] for extended interval arithmetic where conventional interval arithmetic is involved as a special case. This implementation proves that the proposed scheme is suitable and gives the opportunity to be tested in different situations.

Acknowledgements. This work was partially supported by the Bulgarian National Science Fund under grand No. I-507/95.

References

[1] *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754–1985, New York, 1985.

[2] Corliss, G. F.: *Proposal for a Basic Interval Arithmetic Subroutines Library (BIAS).* Tech. Rep., Marquette Univ. Dept. of Maths, Statistics and Computer Science, Milwaukee, Wisc., 1991.

[3] Demmel, J.; Li, X.: *Faster Numerical Algorithms via Exception Handling*, IEEE Trans. on Computers, **43**, No. 8, 1994, pp. 983–992.

[4] Kahan, W. M.: *Interval Arithmetic Options in the Proposed IEEE Floating Point Arithmetic Standard.* In: Nickel, K. (Ed.): *Interval Arithmetic 1980*, Academic Press, 1980, pp. 99–128.

[5] Knüppel, O.: *BIAS — Basic Interval Arithmetic Subroutines.* Bericht 93.3, TU Hamburg-Harburg, Hamburg, 1993.

[6] Moore, R. E.: *Interval Analysis.* Prentice-Hall, N. J., 1966.

[7] Popova, E.: *Extended Interval Arithmetic in IEEE Floating-Point Environment.* Interval Computations, 4, 1994, pp. 100–129.

[8] Wolff von Gudenberg, J.: *Programming Language Support for Scientific Computation.* Interval Computations, 4(6), 1992, pp. 116–126.

Addresses:

E. D. Popova, Institute of Biophysics, Bulgarian Academy of Sciences, Acad. G. Bonchev str., bldg. 21, BG-1113 Sofia, Bulgaria, e-mail: epopova@bgearn.acad.bg.