An Interval Library for Haskell

Genésio G. da Cruz Neto * (ggn@di.ufpe.br)
André L. de M. Santos (alms@di.ufpe.br)

Universidade Federal de Pernambuco, Brazil

Abstract. During the last three decades, interval arithmetic has proved to be useful in many real-life numerical applications.
Although some success has been obtained in using functional languages for scientific computing, there is no work relating experiences in applying this paradigm to interval computations.
In order to make the research in using interval arithmetic in a functional-language setting feasible, this paper presents an interval library for the pure functional language Haskell, based on the Glasgow Haskell Compiler.

1 Introduction

There have been a number of attempts to represent real numbers in a functional language. In these approaches, instead of an interval arithmetic, an exact real arithmetic is used to protect the correctness of results against rounding errors.

Following a classification given in[PE97], three approaches have been proposed for exact real arithmetic. One based on the use of redundant sequences of linear maps[Avi61,BC90], another was based on continued fraction expansions[Gos72,PJ84,Vui90,Les91], and a more recently one uses möbius transformations[Vui90,NK95,PE97,Pot98]. The main drawback of these approaches, however, is that they involve far too much symbolic manipulation.

Interval arithmetic is an alternative to many numerical intensive applications, which may offer a much lower computational overhead. Furthermore, in many real-life situations, the values of the input data comes from measurements. Since measurements are never absolutely precise, they can only be represented by intervals.

Using interval arithmetic[Moo66,AH83], instead of a single approximate number $\tilde{r}$, a pair of bounding numbers is computed which represent $\tilde{r}$ and the upper bound error of this approximation. This technique allows rigorous enclosure of roundoff errors, truncation errors, and errors in input data.

* Bolsista da CAPES Brasilia/Brasil

Interval computations were first applied naively. Namely, many of the resulting intervals (in the 1960's numerical algorithms) were unjustifiably large and thereby conveyed very little information. In the last three decades, however, many sophisticated ideas and algorithm modifications have been proposed that enable the replacement of the excessively wide intervals by reasonably narrow ones.

According to [KK96], modern interval computations have been successfully applied to the following areas:

- *Global Optimization*: Interval arithmetic is widely recognized as valuable in algorithms for global optimization.
- *Constraint Propagation*: In this technique, relationships among intermediate quantities in arithmetic expressions in constraints are used recursively to compute ever-narrower bounds on solution variables. Constraint propagation has been successfully applied to robotics, computer-aided geometric design, general optimization and equational-solving problems.
- *Linear Problems:* In many linear problems, it is desirable that uncertainties in the input data are reflected by describing them as intervals. Unfortunately, there is no feasible (polynomial time) algorithm that computes bounds on the solution sets of all interval linear systems with overestimation less than δ, for any $\delta > 0$[Roh96]. Fortunately, heuristic methods such as interval Gauss elimination or interval Gauss-Seidel method work well for many real-life systems.
- *Statistic, Fuzzy Logic and knowledge Representation*: Experts in these areas have found valuable the ability of intervals to represent uncertainties.
- *Computer Aided Proofs:* Since interval arithmetic allows automatic result verification, it has been successfully used to obtain computational existence and uniqueness proofs.

Many reasons make functional languages attractive to interval computations. To begin with, many of the desirable features a language must have to support self-validated numerical computations[WvG92] (e.g. abstract data types, dynamic arrays, modules, overloading, and arbitrary precision numbers) have already been implemented in functional languages, such as Haskell. Secondly, functional languages have been successfully applied to numerical applications[HP96]. Finally, the mathematical background of the functional paradigm may offer facilities for formal reasoning about interval programs.

This paper introduces an interval library for the purely functional language Haskell[PH97], based on the Glasgow Haskell Compiler[PJHH+93]. The main motivations for this work was to develop an interval package in Haskell, to research the suitability of using functional languages to program interval computations.

2 Interval Arithmetic: an overview

This section presents a review of the most important formulas and concepts of interval arithmetic, based on references [AH83,KK96].

An Interval: A real interval A is defined by

$$A = [\underline{a}, \bar{a}] = \{a \in \mathbb{R} |\ \underline{a} \leq a \leq \bar{a}\}, \quad \underline{a}, \bar{a} \in \mathbb{R}$$

The values $\underline{a}$ and $\bar{a}$ are called lower and upper bounds. We denote the set of real intervals by $\mathbb{IR}$. Two intervals $A = [\underline{a}, \bar{a}]$ and $B = [\underline{b}, \bar{b}]$ are called equal, that is, $A = B$, if they are equal in the set theoretic sense.

$$A = B \Leftrightarrow \underline{a} = \underline{b}, \bar{a} = \bar{b}$$

Arithmetic: The interval operations on $\mathbb{IR}$ with $A, B \in \mathbb{IR}$ are defined by

$$A \odot B = \{a \odot b \mid a \in A \text{ and } b \in B\}, \quad \odot \in \{+, -, *, /\}$$

with $\odot$ being a binary operation on the set of real numbers. It is assumed that $0 \notin B$ in the case of division. The operations on intervals $A = [\underline{a}, \bar{a}]$ and $B = [\underline{b}, \bar{b}]$ may be calculated explicitly as

1. $[\underline{a}, \bar{a}] + [\underline{b}, \bar{b}] = [\underline{a} + \underline{b}, \bar{a} + \bar{b}]$;
2. $[\underline{a}, \bar{a}] - [\underline{b}, \bar{b}] = [\underline{a} - \bar{b}, \bar{a} - \underline{b}]$;
3. $[\underline{a}, \bar{a}] * [\underline{b}, \bar{b}] \ = [min(\underline{a} * \bar{b}, \underline{a} * \underline{b}, \bar{a} * \underline{b}, \bar{a}, \bar{b}), max(\underline{a} * \bar{b}, \underline{a} * \underline{b}, \bar{a} * \underline{b}, \bar{a}, \bar{b})]$;
4. $[\underline{a}, \bar{a}]\ /\ [\underline{b}, \bar{b}] = [\underline{a}, \bar{a}] * [1/\bar{b}, 1/\underline{b}]$

Directed rounding: The key property of interval arithmetic is isotonicity (see definition below). With preserved isotonicity, all interval calculations deliver guaranteed bounds.

$$a \in A, b \in B \Rightarrow (a \odot b) \in (A \odot B), \quad A, B \in \mathbb{IR}, \quad \odot \in \{+, -, *, /\}$$

The above operations of interval arithmetic are only valid for idealized computers. To take the isotonicity into consideration, the above operations have to be modified by using *directed roundings*. The IEEE-754 standard[oEE85] provides directed rounding modes, denoted by $\downarrow$ for downward and $\uparrow$ for upward rounding.

For example, the modified version of the interval subtraction is

$$[\underline{a}, \bar{a}] - [\underline{b}, \bar{b}] = [\downarrow (\underline{a} - \bar{b}), \uparrow (\bar{a} - \underline{b})]$$

Unary Operations: if $r(x)$ is a continuous unary operation on $\mathbb{R}$, then

$$r(X) = [min_{x \in X}\ r(x), max_{x \in X}\ r(x)]$$

defines a unary operation on $\mathbb{IR}$. Examples of unary operations on $\mathbb{IR}$ are

$$ln\ X,\ sin\ X,\ cos\ X,\ etc.$$

For a detailed description of interval arithmetic see [AH83].

3 An Interval Library for Haskell

This section presents an interval library based on the Glasgow Haskell Compiler, which fits into the current standard Haskell type class hierarchy [PH97]. This library supplies all the interval arithmetic operations and transcendental functions, plus some utility routines specially used in interval computations.

3.1 The Interval data type

In this library, an interval is represented as a Haskell abstract data type with the following syntax:

```
data Interval = !Double :> !Double deriving (Eq,Show)
```

where the deriving instance of equal implements exactly the interval equality definition given in the previous section.

3.2 Relational Operators

In our implementation (see code bellow), the overloaded interval operators `<=` and `<` denote the *subset* relations, while `>=` and `>` denote the *superset* relations. These are the standard definitions of the comparison operators used by most interval packages.

```
instance Ord (Interval) where
   (x :> y) <= (x' :> y') = ((x >= x') && (y <= y'))
   (x :> y) <  (x' :> y') = ((x > x')  && (y < y'))
```

3.3 An Interval Arithmetic:

The Haskell arithmetic operations on intervals may be implemented explicitly as

```
instance Num Interval  where
     (x :< y) + (x' :< y') = (x+x') :< (y+y')
     (x :< y) - (x' :< y') = (x-y') :< (y-x')
     (x :< y) * (x' :< y') = (minimum [x*x',x*y',y*x',y*y']) :<
                             (maximum [x*x',x*y',y*x',y*y'])
instance Fractional Interval where
     (x :> y) / (x' :> y') = if (0 <= (x' :> y'))
                                then error "/{IntArit}: Division by Zero"
                                else (x :> y) * ((1 / y') :> (1 / x'))
```

Specifying intervals as an instance of `Num` and `Fractional` (standard Haskell Numeric Classes), we guarantee that they can be treated in the same way as any other Haskell number.

As one can observe, the above code implements the interval multiplication and division operations naively. The version of the interval library actually implemented optimizes these operations by testing the signs of the interval bounds in order to avoid unnecessary computations.

3.4 Using Directed Roundings

The Haskell interval arithmetic implementation described previously does not provide isotonicity, i.e. they do not use the proper ANSI/IEEE rounding modes to compute the upper and lower bounds.

To correctly implement interval arithmetic in Haskell we have to use some basic Haskell IO operations in conjunction with the appropriate functions to switch the rounding modes:

```
 (x :> y) + (x' :> y') =
    unsafePerformIO (
         do roundDown
            let a = x + x'
            roundUp
            let b = y + y'
            roundNear
            return (a :> b) )
```

"`unsafePerformIO`", "`do`" and "`return`" are Haskell IO commands; since we restore the rounding mode to "Near", which is supposed to be the default behaviour, it is reasonably safe to encapsulate the computation under an `unsafePerformIO`.

"`roundDown`", "`roundUp`" and "`roundNear`" are functions which set the rounding mode downward, upward and to the nearest floating point number respectively. These rounding mode functions were implemented by using the Glasgow Haskell compiler primitives to call extern C functions.

The rounding modes are only switched inside the interval operations. Therefore, once these operations are safely controlled by the library code, the overall referential transparency is preserved.

3.5 Implementing Transcendental Functions:

In order to have a fast implementation of transcendental functions, this library interfaces with a public-domain C version of Bias(Basic Interval Arithmetic Subroutines)[Knu93].

Bias defines a concise standard for basic interval arithmetic operations from scalar up to matrix operations as well as for all important interval auxiliary functions.

All interval transcendental functions were defined as part of an overloaded instance of the Haskell class *Floating*. This instance uses special Glasgow Haskell Compiler primitives to implement each elementary function by calling appropriated Bias routines.

3.6 Auxiliary Functions

Some auxiliary functions specially tailored to interval computations are also supplied. Among them, one can find functions to access the interval bounds (`inf` and `sup`), to calculate the diameter of an interval(`diam`), to obtain the midpoint of an interval(`mid`), to find out the distance between two intervals(`dist`), and to compute the intersection of two given intervals (`/\`).

4 Example: The Jacobi Iterative Method

The Interval Jacobi method for the resolution of linear system of equations is based on the following iterative mathematical operator:

$$X^{i+1} = X^i \cap (B_i - \textstyle\sum_{k \neq i} A_{ik} X^i_k)/A_{ii}$$

To illustrate the use of the interval library presented here, part of the Haskell code which implements the interval iterative Jacobi method is given below.

```
main :: IO ()
main = show_vector (iteration input_x0 t distance_vector 0.0 1000)

t xi =  array (bound)
            [(i,(xi!i /\
                (((input_b!i) / (input_a!(i,i))) -
                  ((sum [ (input_a!(i,k))*(xi!k) | k <- range bound, k /= i]) *
                     (1 / input_a!(i,i))))))
                                  | i <- range bound ]
                 where bound = bounds xi

iteration x0 t d acc n =  sequence x0 0
    where sequence x0 count =
                      let x1 = (t x0)
                          in ( if (((d x1 x0) <= acc) || (count >= n))
                               then x1
                               else sequence x1 (count+1) )
```

where "`show_vector`" prints each element of a given vector, "`distance_vector`" calculates the distance between two vectors, "`input_a`" is an interval input matrix, "`input_b`" is an interval input vector, and "`input_x0`" is the initial interval input vector.

5 Performance Results

This section presents performance figures of the benchmark programs below. The programs correspond to implementations of the Jacobi method presented above for input matrices with dimension 9 and 16. The input data (linear interval equations) were generated by the use of a difference method in the resolution of the Dirichlet boundary-value problem[SB76]).

- *Haskell interval*: Haskell implementation using intervals.
- *Haskell rational*: Haskell implementation using the type *Rational*.
- *Haskell punctual*: Haskell implementation using double floating point numbers.
- *Profil interval*: Profil implementation using intervals.
- *Profil punctual*: Profil implementation using double floating point numbers.

Profil[Knu94] is a fast interval C++ class library based on BIAS. It takes advantage of the object oriented characteristics of C++ to implement all arithmetic operations and functions by using calls to overloaded Bias C routines.

The type *Ratio Integer*, also called *Rational*, is the current Haskell alternative for infinite precision arithmetic. *Rational* is a numeric type which represents the ratio of two arbitrary-precision integers. In the Glasgow Haskell compiler, its is implemented using the GNU multiple precision package.

The Haskell programs were compiled with the Glasgow Haskell Compiler version 2.10 and the Profil programs were compiled with the gcc version 2.4.5 (all the programs were compiled with the "-O2" flag). An unix SUN Sparcstation 4 with 64Mb of main memory was the platform used for the measurements.

Prog	9	16
Haskell interval	1.512s	3.127s
Haskell rational	77.36s	5659.84s
Haskell punctual	0.834s	2.488s
Profil interval	0.115s	0.262s
Profil punctual	0.087s	0.131s

Fig. 1. Performance figures

As one can observe from figure 1 the performance of the functional interval programs are far better than that of the rational ones. Comparing interval with punctual programs, we can see that the overhead of using an interval arithmetic in a functional language is proportional to the overhead obtained in Profil.

The functional numerical programs used are one order of magnitude more inefficient than their imperative counterparts in Profil. However, Profil is also one order of magnitude faster than most other imperative interval packages[KK96]. We conjecture that an efficient implementation of overloaded matrix/vector operations using C calls to Bias interval C routines would narrow this performance gap.

6 Related Work

Lins et al [LCC95] first introduced the advantages of joining together interval arithmetic and functional programming. They advocated this as an efficient solution to the problem of having a discrete representation of the real numbers in the functional paradigm. But the paper does not present any practical result or implementation.

7 Conclusions & Future Work

Apart from other useful applications, interval arithmetic is a more efficient alternative to many real-life error based situations than exact real arithmetic.

Functional interval programs demand a high computation overhead in comparison to their imperative implementation in Profil. Nevertheless, functional languages provide many high-level constructs which might be useful for the interval computation community.

The following works are scheduled for the near future:

- Realize a detailed comparison between the interval analysis approach and the exact real arithmetic approaches.
- Implement an efficient functional high accuracy scalar product.
- Try to extend the interval library with overloaded matrix/vector operations by using C calls to Bias matrix/vector routines.
- Compare this work with other interval software by using a test suite specially designed to compare interval packages[Cor93]. This work will allow also the verification of all interval library functions.

8 Acknowledgement

The idea of interfacing a functional language with a C version of the Bias interval library arose in discussion with Rafael Lins. Some time ago, we were trying to implement this idea in the Γ-CMC functional compiler, but it was not feasible.

This work is being developed with support from the Brazilian Research Agency CAPES (process BEX0182/98-2), and also from the computer science department of the Glasgow University.

References

[AH83] G. Alefeld and J. Herzberger. *Introduction to Interval Computation.* New York, Academic Press, 1983.

[Avi61] A. Avizienis61. Signed-digit number representation for fast parallel arithmetic. *IRE Transactions on eletronic computers*, (10):389–400, 1961.

[BC90] H. J. Boehm and R. Cartwright. Exact real arithmetic: formulating real numbers as functions. In D. A. Turner, editor, *Research topics in functional programming*, The University of Texas Year of Programming Series. Almqvist and Wiksell, 1990.

[Cor93] G. F. Corliss. Comparing software packages for interval arithmetic. In *Preprint presented at SCAN'93*, 1993.

[Gos72] R. W. Gosper. Continued fraction arithmetic. Technical Report HAKMEM Item 101B, MIT AI MEMO 239, MIT, 1972.

[HP96] (editors) P. Hartel and R. Plasmeijer. Special issue on state-of-the-art application of pure functional programming languages. *Functional Programming*, 5(3), 1996.

[KK96] R. B. Kearfott and V. Kreinovich, editors. *Applications of Interval Computations*, volume 3 of *Applied Optimization*. Kluwer Academic Publishers, 1996.

[Knu93] O. Knuppel. Bias - basic interval arithmetic subroutines. Technical report, Technische Universitat Hamburg-Harburg, 1993.

[Knu94] O. Knuppel. Profil/bias - a fast interval library. *Computing*, pages 277–287, 1994.

[LCC95] R. D. Lins, M. A. Campos, and M. B. Correia. Functional programming and interval arithmetic. *Reliable Computing*, pages 146–150, 1995.

[Les91] D. R. Lester. Vuillemin's exact real arithmetic. In *1991 Glasgow Functional Programming Workshop*, pages 225–238, 1991.

[Moo66] R. E. Moore. *Interval Analysis*. N. J., Prentice Hall Inc., Englewood Cliffs, 1966.

[NK95] A. M. Nielsen and P. Kornerup. Msb-first digit serial arithmetic. *Universal Computer Science*, 1(7):527–547, July 1995.

[oEE85] American National Standards Institute / Institute of Electrical and Electronics Engineers. IEEE standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std. 754-1985, 1985.

[PE97] P. J. Potts and A. Edalat. Exact real computer arithmetic. Technical report, Imperial College, March 1997.

[PH97] (editors) J. Peterson and K. Hammond. Report on the programming language Haskell version 1.4. Technical report, http://www.haskell.org/, April 1997.

[PJ84] S. L. Peyton Jones. Arbitrary precision arithmetic using continued fractions. Technical report, University College London, 1984. INDRA Note 1530.

[PJHH+93] S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a technical overview. In *UK Joint Framework for Information Technology (JFIT)*, 1993.

[Pot98] P. J. Potts. *Exact Real Arithmetic using Möbius Transformations*. PhD thesis, Imperial College, 1998.

[Roh96] J. Rohn. Linear interval equations: Computing enclosures with bounded relative overestimation is NP-hard. In R. B. Kearfott and V. Kreinovich, editors, *Applications of Interval Computations*, volume 3 of *Applied Optimization*, chapter 4. Kluwer Academic Publishers, 1996.

[SB76] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, 1976.

[Vui90] J. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transaction on computers*, 39(8):1087–1105, 1990.

[WvG92] J. Wolff von Gudenberg. Programming language support for scientific Computation. *Interval Computations*, 4(6):116–126, 1992.