A Scientific Calculator for Exact Real Number Computation Based on LRT, GMP and FC++.

Alejandra Lucatero*, J. Raymundo Marcial-Romero* and J. A. Hernández*

ABSTRACT

Language for Redundant Test (LRT) is a programming language for exact real number computation. Its lazy evaluation mechanism (also called call-by-need) and its infinite list requirement, make the language appropriate to be implemented in a functional programming language such as Haskell. However, a direction translation of the operational semantics of LRT into Haskell as well as the algorithms to implement basic operations (addition subtraction, multiplication, division) and trigonometric functions (sin, cosine, tangent, etc.) makes the resulting scientific calculator time consuming and so inefficient. In this paper, we present an alternative implementation of the scientific calculator using FC++ and GMP. FC++ is a functional C++ library while GMP is a GNU multiple presicion library. We show that a direct translation of LRT in FC++ results in a faster scientific calculator than the one presented in Haskell.

RESUMEN

El lenguaje de verificación redundante (LRT, por sus siglas en inglés) es un lenguaje de programación para el cómputo con números reales exactos. Su método de evaluación lazy (o mejor conocido como llamada por necesidad) y el manejo de listas infinitas requerido, hace que el lenguaje sea apropiado para su implementación en un lenguaje funcional como Haskell. Sin embargo, la implementación directa de la semántica operacional de LRT en Haskell así como los algoritmos para funciones básicas (suma, resta, multiplicación y división) y funciones trigonométricas (seno, coseno, tangente, etc) hace que la calculadora científica resultante sea ineficiente. En este artículo, presentamos una implementación alternativa de la calculadora científica usando FC++ y GMP. FC++ es una librería que utiliza el paradigma Funcional en C++ mientras que GMP es una librería GNU de múltiple precisión. En el artículo mostramos que la implementación directa de LRT en FC++ resulta en una librería más eficiente que la implementada en Haskell.

Recibido: 20 de Diciembre de 2011
Aceptado: 14 de Febrero de 2012

INTRODUCTION

During the last decades there have been several paradigms to compute with the real numbers. Among them we can cite floating point number arithmetic [16], interval analysis [15], algebraic manipulation [20], exact real number computation [21], etc.

The exact real number computation paradigm has several advantages compared to the others, for example, it avoids the rounding off errors that occurs in floating point arithmetic. Like interval analysis, it bounds the result, however it guarantees the computation of a smaller interval at each step of the computation which do not occur in interval analysis. Algebraic manipulations can be used in exact real number computation however, when no further reduction can be done a exact method is used to compute the solution compared to algebraic manipulation which has to turn to floating point arithmetic or interval analysis.

There have been several theoretical proposals to exact real number computations [19, 5, 10]. Most of them have succeeded to prove that the theory is sound and complete. However, when they have been implemented, none of them has achieved to be efficient and straightforward to translate from the theory to the practice. On the other hand, implementations such as IRRAM [16], MPFR [6] and RealLib [8] have been developed in C and C++, however, in order to run faster, they have lost the elegancy of functional programming and also,

Palabras clave:
Cálculo con números reales; programación funcional; semántica operacional
Keywords:
Real number computation; functional programming; operational semantics

* Computer Engineering Department, Universidad Autónoma del Estado de México, Cerro de Coatepec s/n, Ciudad Universitaria, C.P. 50100, Toluca, Estado de México, México. E-mail: ritsuka00@gmail.com, jrmarcialr@uaemex.mx, xoseahernandez@gmail.com

they have slightly deviated from the theory. Although we consider that faster implementations is what is required in practice, we believe that there is increased confidence in the correctness of an implementation the closer it is to the original theory.

A pair of implementations which have a close harmony between theory and practice are Era [2] and a corecursive sign digit representation [3], implemented in Objective Caml and Coq respectively. However, Era, as said by the authors, is slower than IRRAM since C++ generally compiles to more efficient code than Objetive Caml. Respect to the corecursive implementation, although an excellent theory is presented, the efficiency is never mentioned.

A further well established theory for exact real number computation is LRT (Language for Redundant Test) [10]. It has been proved that any computable first order function can be defined in LRT [13]. Moreover, an implementation of LRT in Haskell has been presented on Marcial et. al [11]. However, its efficiency compared to a sign digit representation [18] was very poor. In Marcial et. al [12] an implementation of a basic calculator based on LRT and programmed using FC++ and GMP was presented. It was shown that the execution time improved compared to other Haskell implementations.

In this paper, we present the results of an extension of the basic calculator to a scientific calculator, showing that the time also improved in trigonometric functions as well as other operations. FC++ is an extension of C++ which allows to translate functional programs in almost an straightforward way, meaning that the effects of functional programing are there. Since the types of FC++ are the standard of C++ (which implies that numbers are truncated or rounded), we use instead the types of GMP. GMP is a well known library which computes arithmetic operations faster than many other implementations. Additionally, GMP allows to compute to any required precision.

The paper is divided as follows, in Section *The LRT Language*, the language LRT is defined. In Section *The Scientific Calculator* a brief explanation of the trigonometric implementations is presented. Finally the conclusions and further work is discussed.

THE LRT LANGUAGE

We introduce the LRT language, which is a variant of Real PCF [5]. This is a call-by-name language.

Syntax

The language LRT is an extension of PCF (Programming Computable Functions) [17] with a ground type for real numbers and suitable primitive functions for real-number computation. Its raw syntax is given by

$$\begin{aligned} x &\in \mathit{Variable},\\ t &::= \mathtt{nat} \mid \mathtt{bool} \mid \mathtt{I} \mid t \to t,\\ P &::= x \mid \mathtt{n} \mid \mathtt{true} \mid \mathtt{false} \mid (+1)(P) \mid (-1)(P) \mid \\ &\quad (=\mathtt{0})(P) \mid \mathtt{if}\ P\ \mathtt{then}\ P\ \mathtt{else}\ P \mid \mathtt{cons}_{[\underline{a},\overline{a}]}(P) \mid \\ &\quad \mathtt{tail}_{[\underline{a},\overline{a}]}(P) \mid \mathtt{rtest}_{l,r}(P) \mid \lambda x : t.P \mid PP \mid \mathtt{Y}P, \end{aligned}$$

where *Variable* is a set of variables, t represents a set of types, in this case the language has three ground types, the natural numbers type (represented by `nat`), the booleans (represented by `bool`) and the unit real number type (represented by `I` which denotes the set of intervals in $[-1,1]$, as it was shown in [9] the complete computable real line can be easily represented in this language, even more the implementation presented here considers the complete real line). The type $t \to t$ denotes higher order types. The constructs of the language (represented by P) are the variables (represented by x), the constants for natural numbers and booleans (represented by `n`, `true` and `false`), the successor, predecesor and equal test for zero operations for naturals numbers ((+1), (-1) and (=0)), the classical `if` operator; three operation for exact real number computation (`cons`, `tail` and `rtest`) where the subscripts of the constructs `cons` and `tail` are rational intervals (sometime written as a or $[\underline{a},\overline{a}]$) and those of `rtest` are rational numbers. The last three constructors of the languages are those of the lambda calculus ($\lambda x : t.P, PP$ and $\mathtt{Y}P$) where the first denotes abstraction, the second application and the third recursion.

Because the intention of this paper is not to present the denotational semantics of the language which is based on powerdomains [10], we just present the mathematical objects which describe the `cons`, `tail` and `rtest` constructors. The others are the well known PCF constructors and can be consulted at [7, 17] .

Let $D \subseteq [-1,1]$, the function $\mathtt{cons}_a : D \to D$ is the unique increasing affine map with image the interval a, i.e.,

$$\mathtt{cons}_{[\underline{a},\overline{a}]}([\underline{x},\overline{x}]) = \left[\frac{\overline{a}-\underline{a}}{2}\underline{x} + \frac{\overline{a}+\underline{a}}{2}, \frac{\overline{a}-\underline{a}}{2}\overline{x} + \frac{\overline{a}+\underline{a}}{2}\right]$$

That is, rescale and translate the interval $[-1,1]$ so that it becomes $[\underline{a},\overline{a}]$, and define $\mathtt{cons}_{[\underline{a},\overline{a}]}([\underline{x},\overline{x}])$ to be the interval which results from applying the same rescaling and translation to $[\underline{x},\overline{x}]$. In order to keep the notation simple, when the context permits we use x to

represent $[\underline{x}, \overline{x}]$, meaning that the same operation is applied to both end points of the interval obtained, for example the cons function can be written as:

$$\mathtt{cons}_{[\underline{a},\overline{a}]}(x) = \frac{\overline{a} - \underline{a}}{2}x + \frac{\overline{a} + \underline{a}}{2} \tag{1}$$

The function $\mathtt{tail}_{[\underline{a},\overline{a}]}(x)\colon D \to D$ is a left inverse, i.e.

$$\mathtt{tail}_a(\mathtt{cons}_a(x)) = x.$$

More precisely, the following left inverse is taken, where κ_a is $\overline{a} - \underline{a}$ and τ_a is $\overline{a} + \underline{a}$:

$$\mathtt{tail}_{[\underline{a},\overline{a}]}(x) = \max(-1, \min((2x - \tau_a)/\kappa_a, 1)).$$

This definition guarantees that the range of the tail function is in the interval [-1,1]. The details of why this is a convenient definition can be consulted in [5]. It is worthy to mention that an infinite shrinking sequence of cons intervals represent a real number in the interval $[-1, 1]$, the operational semantics defined below gives a rule for constructing a real number.

The definition of the function $\mathtt{rtest}_{l,r} : D \to \{\mathtt{true}, \mathtt{false}\}$, where $l < r$ are rational numbers, can be formulated as

$$\mathtt{rtest}_{l,r}(x) = \begin{cases} \mathtt{true}, & \text{if } x \subseteq (-\infty, l], \\ \mathtt{true}\ \mathit{or}\ \mathtt{false}, & \text{if } x \subseteq (l, r), \\ \mathtt{false}, & \text{if } x \subseteq [r, \infty). \end{cases} \tag{2}$$

The function $\mathrm{rtest}_{l,r}$ is operationally computable because, for any argument x given intensionally as a shrinking sequence of **cons** intervals, the computational rules systematically establish one of the semidecidable conditions $l < \overline{x}$ and $\underline{x} < r$ where l, r are rational numbers.

Operational Semantics

We consider a small-step style operational semantics for our language. We define the one-step reduction relation $\to$ to be the least relation containing the one-step reduction rules for evaluation of PCF [17] together with those given below.

We first need some preliminaries. For intervals a and b in $[-1, 1]$, we define

$$ab = \mathsf{cons}_a(b),$$

where cons is the function defined previously. This operation is associative, and has the interval $[-1, 1]$ (denoted by $\perp$) as its neutral element [5]:

$$(ab)c = a(bc), \qquad a\perp = \perp a = a.$$

In the interval domain literature [1], $a \sqsubseteq b$ iff $b \subseteq a$. Moreover,

$$a \sqsubseteq b \iff \exists c \in D.\ ac = b,$$

and this c is unique if a has non-zero length. In this case we denote c by

$$b \setminus a.$$

For intervals a and b, we define

$$a \le b \iff \overline{a} \le \underline{b}$$

and

$$a \uparrow b \iff \exists c.\ a \le c \text{ and } b \le c.$$

With this notation, the rules for Real PCF as defined in [5] are:

(1) $\mathbf{cons}_a(\mathbf{cons}_b M) \to \mathbf{cons}_{ab} M$
(2) $\mathbf{cons}_a M \to \mathbf{cons}_a M'$
(3) $\mathbf{tail}_a(\mathbf{cons}_b M) \to \mathbf{Ycons}_{[-1,0]}$ if $b \le a$
(4) $\mathbf{tail}_a(\mathbf{cons}_b M) \to \mathbf{Ycons}_{[0,1]}$ if $b \ge a$
(5) $\mathbf{tail}_a(\mathbf{cons}_b M) \to \mathbf{cons}_{b\setminus a} M$ if $a \sqsubseteq b$ and $a \ne b$
(6) $\mathbf{tail}_a(M) \to \mathbf{tail}_a(M')$
(7) **if true** $M\ N \to M$
(8) **if false** $M\ N \to N$
(9) **if** $M\ N_1\ N_2 \to$ **if** $M'\ N_1\ N_2$

For our language *LRT*, we add:

(10) $\mathbf{rtest}_{l,r}(\mathbf{cons}_a\ M) \to \mathbf{true}$ if $\overline{a} < r$
(11) $\mathbf{rtest}_{l,r}(\mathbf{cons}_a\ M) \to \mathbf{false}$ if $l < \underline{a}$
(12) $\mathbf{rtest}_{l,r}\ M \to \mathbf{rtest}_{l,r}\ M'$ if $M \to M'$.

Remarks:

1. Rule (1) plays a crucial role and amounts to the associativity law. The idea is that both a and b give partial information about a real number, and ab is the result of gluing the partial information together in an incremental way. See [5] for a further discussion including a geometrical interpretation.

2. Rules (2), (6), (9) and (12) are applied whenever any of the other rules are matched.

3. Rule (3) represents the fact that we already know that the rest of the real number we are looking for is an infinite sequence of the interval $[-1, 0]$, i.e.
$$\mathbf{Ycons}_{[-1,0]} = \mathbf{cons}_{[-1,0]}(\mathbf{cons}_{[-1,0]}(\ldots))$$

4. Rule (4) is similar to rule (3).

5. Rule (5) is applied when the partial information accumulated at some point contains the interval of the next input.

6. Rules (7) and (8) are the classical conditional rules.

7. Notice that if the interval a is contained in the interval $[l, r]$, rules (10) and (11) can be applied.

8. Rules (10)-(12) cannot be made deterministic given the particular computational adequacy formulation which is proved in [10].

9. In practice, one would like to avoid divergent computations by considering a strategy for application of the rules. In [10] total correctness of basic algorithms and in [14] total correctness of first order functions are shown, hence any implementation of any strategy will be correct.

For a deeper discussion of the relation between the operational and denotational semantics of LRT, the reader is referred to [10, 14].

THE SCIENTIFIC CALCULATOR

The scientific calculator consists of basic operations (addition, subtraction, multiplication and division), trigonometric function (sin, cosine, tangent, cotangent, etc.) and additional function like x^y, $\sqrt{x}$, π, among others.

In order to describe how an algorithm in LRT works we present a particular example. The average function defined by:

$$x \oplus y = \frac{x+y}{2}$$

can be implemented in LRT as follows:

```
faverage(x, y)  =
  if rtest_{l,c} (x)
   then
    if rtest_{l,c} (y)
     then Cons_L(faverage(Tail_L x, Tail_L y))
     else
      if rtest_{c,r} (y)
       then Cons_{C_1}(faverage(Tail_L x, Tail_C y)
       else Cons_C(faverage(Tail_L x, Tail_R y))
   else
    if rtest_{c,r} (x)
    then
      if rtest_{1,c} (y)
       then Cons_{C_1 1}(faverage(Tail_C x, Tail_C y))
       else
        if rtest_{c,r} (y)
         then Cons_C(faverage(Tail_C x, Tail_C y))
         else Cons_{C_2}(faverage(Tail_C x, Tail_R y))
    else
      if rtest_{1,c} (y)
       then Cons_C(faverage(Tail_R x, Tail_L y))
       else
        if rtest_{c,r} (y)
         then Cons_{C_2}(faverage(Tail_R x, Tail_C y))
         else Cons_R(faverage(Tail_R x, Tail_R y))
```

where

$$l = -1/2, \quad c = 0, \quad r = 1/2,$$
$$L = [-1, 0], \quad C = [-1/2, 1/2],$$
$$R = [0, 1], \quad C_1 = [-3/4, 1/4], \quad C_2[-1/4, 3/4].$$

The intuition behind this program is the following. If both x and y are in the interval L, then we know that $x \oplus y$ is in the interval L, if both x and y are in the interval R, then we know that $x \oplus y$ is in the interval R, and so on. The boundary cases are taken care of by the rtest conditional.

What is interesting is that, despite the use of the multi-valued construction `rtest`, the overall result of the computation is single valued. In other words, different computation paths will give different shrinking sequences of intervals, but all of them will shrink to the same number. A proof of this fact and of correctness of the program is provided in [9]. This can be seen as follows: an unfolding of $1/2 \oplus 1/2$ gives the expresion $\mathtt{Cons_R(faverage(0 \oplus 0))}$. This means that the result of the operation is in the interval $R = [0, 1]$. A second unfolding gives $\mathtt{Cons_R Cons_C(faverage}(1 \oplus 1))$, due to it is a call by need language, the first two conses are reduced using Equation 1 . This means that the result is in the interval $[1/3, 2/3]$. A repeated unfolding gives the required result $1/2$.

We present and explain a pair of GMP-FC++ implementations of the operational semantics described in the previous section. The idea is to illustrate the straightforward translation of the algorithms presented in [10] to our framework and present an implementation of the trigonometric functions comparing its efficiency with previous functional programming implementations [18, 11].

Example 1 *An easy example is the representation of the real number* 1 *which can be coded as follows:*

```
struct InfiniteListOne :
 public CFunType<List<Intervalo>> {
 List<Intervalo> const {
 Interval i1;
 mpf_init_set_ui(i1.lower,0);
 mpf_init_set_ui(i1.upper,1);
 return cons(i1,curry(InfiniteListOne));
 }
} listainfinita;
```

An unfolding of the program gives the interval **cons**$(0,1)$. *Since the procedure calls itself, a second unfolding gives the intervals* **cons**$(0,1)$**cons**$(0,1)$. *This procedure does not have a basic case, so a potential infinite list of intervals of the form* **cons**$(0,1)$ *is generated. Since FC++ can be used as a call-by-need language, a call to the procedure InfiniteListOne returns the reduction of intervals from left to right to the required precision applying Equation 1 .*

The **cons** operation presented in Equation 1 is implemented as follows:

```
struct Conz :
 public CFunType<Interval,Interval,Interval>{
 Interval operator()(Interval a, Interval x)
   const {
         mpf_t aux;
         mpf_init2(aux, Prec);
         mpf_init2(iC.lower, Prec);
         mpf_init2(iC.upper, Prec);
         mpf_sub(aux, a.upper, a.lower);
         mpf_div_ui(aux, aux, 2);
         mpf_mul(iC.upper, aux, x.upper);
         mpf_mul(iC.lower, aux, x.lower );
         mpf_add(aux, a.upper, a.lower);
         mpf_div_ui(aux, aux, 2);
         mpf_add(iC.lower, iC.lower, aux);
         mpf_add(iC.upper, iC.upper, aux);
         mpf_clear(aux);
  return iC;
  }
} conz;
```

According to Equation 1 **cons** is a lineal function which takes two intervals as inputs an returns a single interval as output stated in the code by < Interval, Interval, Interval >. The initialization of variables in GMP is done by the function mpf_init2. This function takes two arguments, the variable to be initialized and its precision in terms of bits. In this case an auxiliary temporal variable and a global interval variable, in which the result is returned, are initialized. Basic operations like addition, subtraction, etc. are computed in GMP with especial procedures. These operations begin with the word mpf_. The comments included in the code, indicate which operation is performed. The reader can compare the operations against Equation 1. Finally, the procedure mpf_clear, free dynamic memory allocation used by GMP, in this procedure the unique local variable is aux. A similar implementation is coded for the **tail** function.

To approximate a real number, the first rule of the operational semantics is applied to the elements on the *mantissa* as many times as precision is required. If the first rule is not applied, a further evaluation of the input list should be done.

It is worth to note that the implementation of the operational semantics only works with real numbers in the interval $[-1,1]$. The final result to the desired precision is calculated multiplying both interval end points at the head of the *mantissa* by 2 to the power of the exponent.

We hope that the discussion of the previous codes allows the reader to understand the implementation.

The trigonometric functions

In order to define the trigonometric functions an algorithm which compute the limit of Cauchy sequences is defined. The trigonometric functions can be defined as the limits of such sequences. The algorithm which compute the limit of Cauchy sequence was first presented in [18].

To define the limits function, an auxiliary function which takes three real number (x,y,z) as input (in the form of infinite lists) is defined. The numbers have the properties that $x \leq y \leq z$. The function examine x and z and generates a stream of conses whose range contains all possible values of y and then uses y directly to return the reminder of the output such that the whole output represent the number y. Importantly, this function determines information about y from the bounds of x and z before examining y itself.

This methods seems weird since y is an input of the function, however the real number y is passed as a pointer to other functions which computes its value, hence it is better to use x and z to compute the value of y before examining y. To compute limits it is required that this function has the property that if x and z are closer

than some finite amount a cons is generated, and that if x and z are equal, y is not examined at all.

The function may be used to convert a real number represented as a stream of nested intervals into a single stream of conses. The function tries to find common digits from the first interval in the stream of nested intervals when possible, end then uses the remaining intervals to determine the rest of the conses in the stream.
Because the nested intervals converge to a single value, we can get arbitrarily many output conses by examining a finite number of intervals. In order to compute the limit of a given Cauchy sequence, we can express the sequence as a stream of nested lower and upper bounds of the limit. Many useful functions can be defined as the limits of Cauchy sequences.
Suppose we have a stream of nested intervals represented as $(x_1, z_1), (x_2, z_2), (x_1, z_1), \ldots$ with the following properties:

$$[x_1, z_1] \supseteq [x_2, z_2] \supseteq [x_3, z_3] \supseteq \cdots$$

Let f to be the function that takes the infinite representation of three real numbers x, y, z as input and outputs the stream y by examining x and z. We can compute the stream y by the stream of nested intervals $(x_1, z_1), (x_2, z_2), (x_1, z_1), \ldots$ as the result of:

$$\begin{aligned} y &= \lim[(x_1, z_1), (x_2, z_2), (x_1, z_1), \ldots] \\ &= f(x_1, \lim[(x_2, z_2), (x_1, z_1), \ldots], z_1) \end{aligned}$$

Computing trigonometric functions is performed as follows: suppose we wish to compute the value fo $f(x)$, if we find a series in terms of x which either tends towards a $\lim f(x)$ with a known rate of convergence or converges to but oscillates round the $\lim f(x)$. Once we have this series we can generate a sequence of upper and lower bounds on this limit at each term of the original sequence. We know that the sequence converges so we cab use this fact to generate an infinite and strictly nested stream of intervals containing the limit.

For example, the sine function can be defined using the following series:

$$sin(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)}$$

Using this sum, we can easily construct a sequence $S(x)$ such that $S_n(x) \to sin(x)$ as $n \to \infty$. In other words

$$\lim_{n\to\infty} S_n(x) = sin(x)$$

which is equivalent to compute the limit of the nested intervals $(S_0, S_1), (S_1, S_2), \ldots$. The FC++ code to compute this function can be consulted at http://fi.uaemex.mx/rmacial/FC++.

Table 1 .

Time reported by both the Haskell Glasgow and the FC++ compilers. All operations were calculated at precision $1E-5$

Operation	Time reported in seconds for each implementation	
	Haskell	**FC++**
$\sum_{i=1}^{20} \frac{i}{2i+1}$	15.99987s	1.41s
$\sum_{i=1}^{50} \frac{i}{3i+1}$	30.4s	3.39s
$\prod_{i=1}^{16} \frac{i}{2i+1}$	111.8s	36.8s
$j = \frac{1}{3}$		
$for\ i = 2\ to\ 20$	1583.99s	786.81s
$j = j\ div\ \frac{i}{2i+1}$		
$sin(1/3 + cos(7/9))/cos(1/3 + sin(7/11))$	12.54s	0.716s
$e^{(tan(3/11)-tan(2/13))}$	9.27s	1.048s
$\pi * arctan(1/3) + (cos(2/3) * tan(13/15))$	23.97s	0.556s
$cos(1/2) + tangente(1/4)$	5.3s	0.06s
$sin(3/11 + e^{(1/3)}) * sin(4/13 - e^{(2/3)})$	27.109s	0.564s
$arctan(1/3) + \pi$	12.8s	0.7s
$(sin(1/3) + \pi)/(cos(1/3 + sin(7/11))$	35.8s	12.4s
$\pi * arctan(1/3) + (cos(2/3) * tan(2/3))$	43.2s	6.15s

Comparing with the three digits representation

The Haskell implementation we use in our comparison was presented at Marcial et al. [11]. We will not discuss the implementation in this paper, instead we refer the interested reader to the cited reference. We can say, however, that this implementation is more efficient than the Haskell in each operation. These results are showed at Table 1.

CONCLUSIONS

We have presented an implementation of LRT in the FC++ programming language using the GMP library.

Although C++ is an imperative language, FC++ is a functional C++ implementation, meaning that it allows a call by need evaluation and the definition of infinite lists. The algorithms presented in [11] were straightforward translated to this setting and the time reported is considerably improved compared to an implementation based on a pure functional programming language. In order to show that this implementation is faster, we used the logistic map which is caotic function. However, our implementation is still slower than at least another C++ implementation called iRRAM. A first further work is the implementation of trigonometric functions using Taylor series, e.g. the limit function has to be defined. A second further work is the improvement of the efficiency of the implementations in order to be as competitive as the C++ based.

REFERENCES

[1] Abramsky, S., Jung, A. (1994). Handbook of Logic in Computer Science.*Clarendon Press,* 3: 1-168.

[2] Bauer, A., Kavkler, I. (2008). Implementing real numbers with rz. *Proceedings of the Fourth International Conference on Computability and Complexity in Analysis, CCA,* pp.365-384.

[3] Ciaffaglione, A., Gianantonio, P.D. (2006). A certified, corecursive implementation of exact real numbers. *Theoretical Computer Science,* 351(1):39-51.

[4] Devaney, R.L. (1989). *An Introduction to Chaotical Dynamical Systems.* Addison-Wesley, California, 2nd edition.

[5] Escardó,M. H. (1996). PCF extended with real numbers. *Theoretical Computer Science,* 162(1):79-115.

[6] Pélissier, P., Hanrot, G., Lefévre, V., Zimmermann, P. *The MPFR library. INRIA.* http://mpfr.org.

[7] Gunter, C. A. (1992). *Semantics of Programming Languages.* The MIT Press.

[8] Lambov, B. *The reallib project.* BRICS, University of Aarhus. http://brics.dk/~barnie/RealLib.

[9] Marcial-Romero, J. R. (2004). Semantics of a sequential language for exact real-number computation. *PhD thesis,* University of Birmingham.

[10] Marcial-Romero, J. R., Escardó, M. H. (2007). Semantics of a sequential language for exact real-number computation. *Theoretical Computer Science,* 379(1/2):120-141

[11] Marcial-Romero,J. R., Hernández,J. A., Montes-Venegas, Héctor A. (2009). Comparing implementations of a calculator for exact real number computation. *Proceedings of the Mexican International Conference on Computer Science,* IEEE Computer Society Press. pp.13-23.

[12] Marcial-Romero, J. R., Lucatero, A., Hernández, J. A. (2011). A GMP-FC++ implementation of a calculator for exact real number computation based on LRT. *Proceedings of the Seventh Latin American Workshop on Logic / Languages, Algorithms and New Methods of Reasoning,* pp.71-82.

[13] Marcial-Romero, J. R., Moshier, A. (2008). Sequential real number computation and recursive relations. *Proceedings of the Fourth International Conference on Computability and Complexity in Analysis,* pp.171-189.

[14] Marcial-Romero, J. R., Moshier, A. (2008). Sequential real number computation and recursive relations. *Mathematical Logic Quarterly,* 54(5):492-507.

[15] Moore, R. E. (1966). *Interval Analysis.* Prentice-Hall, Englewood Cliffs.

[16] M uller, N. iRRAM - exact arithmetic in C++. Universit at Trier. http://www.informatik.uni-trier.de/iRRAM.

[17] Plotkin, G. D. (1977). LCF considered as a programming language. *Theoretical Computer Science,* 5(1):223-255.

[18] Plume, Dave. (1998). A calculator for exact real number computation. 4th Year Project Report, Department of Computer Science and Artificial Intelligence, University of Edinburgh.

[19] Potts,P. J., Edalat, A., Escardó, M.H. (1997). Semantics of exact real arithmetic. *In Proceedings of the Twelveth Annual IEEE Symposium on Logic In Computer Science.* IEEE Computer Society Press.

[20] Schalk, A. (1993). Algebras for Generalized Construction. PhD thesis, TU Darmstadt.

[21] Weihrauch, K.(2000). *Computable Analysis.* Springer