A GMP-FC++ implementation of a calculator for exact real number computation based on LRT

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

Universidad Autónoma del Estado de México (UAEM)
Facultad de Ingeniería
Toluca, México
rmarcial,alejandra,xosehernandez@fi.uaemex.mx

Abstract. In the last decade, there have been several implementations of exact real number computation. All of them try to establish an standard to use in different programming languages. However, none of them has succeeded on that goal. In this paper, we present another implementation of a calculator for exact real number computation based on the sound and complete theoretical programming language LRT. The calculator is programmed in FC++ and GMP. FC++ has the elegance of functional programming while GMP allows to compute faster to the required number of digits. We present the logistic map implementation, to show preliminary efficiency results compared to other functional programming implementations.

Key words: Exact real-number computation; Sequential Computation; PCF; Semantics of programming languages.

1 Introduction

The exact real number computation paradigm avoids the rounding off errors that occurs in floating point arithmetic. The main reason is that the reals are a field while the floating point numbers are not. Exact real number computation allows to calculate with real numbers to the required precision without carry on rounding errors. There have been several theoretical proposals to exact real number computations [17, 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 [14], 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, 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 [12]. Moreover, an implementation of LRT in Haskell has been presented on Marcial et. al [11]. However, its efficiency compared to a sign digit representation [16] was very poor. In this paper, we present an implementation of LRT on the programming language FC++. FC++ is an extension of C++ which allows to translate functional programs in almost an straightforward way, meaning that the elegancy of functional programs is 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 2, the language LRT is defined. In section 3 a brief explanation of the implementation is discussed followed by a comparison of efficiency of an important function presented in the literature. Finally the conclusions and further work is discussed.

2 The LRT Language

We introduce the LRT language, which amounts to the language considered by Escardó [5] with the parallel conditional removed and a constant $\mathsf{rtest}_{l,r}$ added. This is a call-by-name language.

2.1 Syntax

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

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

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 $\mathtt{nat}$),

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 $\mathtt{n}, \mathtt{true}$ and $\mathtt{false}$), the succesor, predecesor and equal test for zero operations for naturals numbers ((+1), (-1) and (=0)), the classical `if` operator of almost any programming language; 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 $\mathtt{cons}, \mathtt{tail}$ and `rtest` constructors. The others are the well known PCF constructors and can be consulted at [7, 15] .

Let $D \subseteq [-1,1]$, the function $\mathtt{cons}_a \colon D \to D$ 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.

2.2 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 [15] 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 $\bot$) as its neutral element [5]:

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

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

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

and

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

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 $b \leq a$

(4) $\mathbf{tail}_a(\mathbf{cons}_b M) \to \mathbf{Ycons}_{[0,1]}$ if $b \geq a$

(5) $\mathbf{tail}_a(\mathbf{cons}_b M) \to \mathbf{cons}_{b \setminus a} M$ if $a \sqsubseteq b$ and $a \neq b$

(6) $\mathbf{tail}_a(M) \to \mathbf{tail}_a(M')$

(7) $\mathbf{if\ true}\ M\ N \to M$

(8) $\mathbf{if\ false}\ M\ N \to N$

(9) $\mathbf{if}\ M\ N_1\ N_2 \to \mathbf{if}\ M'\ N_1\ N_2$

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 [13] 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, 13].

3 The Implementation

Due to the lack of space, in this section we only 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 logistic map comparing its efficiency with previous functional programming implementations [16, 11].

We represent in FC++ the real numbers by the datatype RExpMan which consists of a pair of the form *(mantissa, exponent)* where the *mantissa* is an infinite list of rational intervals in [-1,1] and the *exponent* is an integer. This exponent allows to represent real numbers outside the unit interval. For example 3.17 can be represented by 0.79×4, which in our notation is represented by $(0.79, 4)$, and 0.79 is represented by an infinite list. The datatype is defined in FC++ in the following way:

```
typedef struct{
    mpf_t upper;
    mpf_t lower;
}Interval;

typedef struct{
List< Intervalo > listaRExpMan;
int exponent;
}RExpMan;
```

An Interval is a pair of GMP numbers of the form $(lower, upper)$. A real number is represented by an infinite list of intervals and an exponent. Notice that we have not restricted the GMP intervals to be in the interval $[-1, 1]$, however their use in the implementation do.

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;
```

The intuition behind this program is the following. An unfolding of the program gives the interval $\mathbf{cons}(0, 1)$. Since the procedure calls itself, a second unfolding gives the intervals $\mathbf{cons}(0, 1)\mathbf{cons}(0, 1)$. 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 number of intervals according to what is required.

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);

            // (aUp - aLow) / 2  --> A
           mpf_sub(aux, a.upper, a.lower);
           mpf_div_ui(aux, aux, 2);

           // ( A ) * x  --> B
           mpf_mul(iC.upper, aux, x.upper);
           mpf_mul(iC.lower, aux, x.lower );

           // (aUp + aLow) / 2
           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. We present the first two rules of the operational semantics. The others are coded similarly.

```
struct Evaluation :
   public CFunType< List< Interval >, List< Interval > >{
   List< Interval > operator()(List< Interval > eI) const {
   Interval cons1, cons2;
   cons1 = head(eI);
   eI = tail(eI);
   cons2 = head(eI);
    // cons_a(cons_b(M)) →  cons_ab(M)
   if(cons1.type == 0 && cons2.type == 0){
      eI = tail(eI);
      return cons( conz(cons1, cons2) , eI);
       // cons_aM →  cons_aM'
   }else if( cons1.type == 0 )
         return Evaluation()(cons( cons1, Evaluation()(eI) ) );
```

Example 2. Considering Example 1, a call to *take 1 (Evaluation(InfiniteListOne))* returns the interval $\mathbf{cons}(1/2, 1)$. The reason is that a call by-need determines that two members of the infinite list *InfiniteListOne* are needed in order to obtain an element of the call. A *take 2* call returns the interval $\mathbf{cons}(3/4, 1)$. Thus, *Evaluation(InfiniteListOne)* produces a shrinking sequence of intervals converging to 1.

It is worth to note that this 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.

A last operational semantics rule presented in this paper is the non-deterministic **rtest** operator. This operator can be programmed in two ways as pointed out in the previous section. One of them is the following:

```
struct RTest : public CFunType< mpf_t, mpf_t, List< Intervalo >, RTestDev >{
   RTestDev operator()(double l, double r, List< Intervalo > i) const {
      Intervalo aux = head(i);
      RTestDev rtestdev;
         // x̄  <= r
      if(mpf_cmp_d(aux.upper, r) <= 0 ){
         rtestdev.b = true;
         rtestdev.listaRTest = i;
         return rtestdev;
```

```
            // x >= l
        }else if(mpf_cmp_d(aux.lower,l) >= 0 ){
                        rtestdev.b = false;
                        rtestdev.listaRTest = i;
                        return rtestdev
                       // neither x <= r nor x >= l
                    }else{
                        return RTest()(l, r, curry(evaluation, i) );
    }
    }
} rtest;
```

We hope that the discussion of the previous codes, together with equation 2 allows the reader to understand the implementation of **rtest**.

3.1 The logistic Map

The logistic map is a function $f : [0, 1] \to [0, 1]$ defined by

$$f(x) = ax(1 - x)$$

for a given constant a. Devaney [4] stated that it was first considered as a model of population growth by Pierre Verhulstby in 1845. For example, a value 0.5 may represent 50% of the maximum population of cattle in a given farm. The problem consists on, given a real number x_0, to compute the orbits

$$x_0, f(x_0), f(f(x_0)) \dots f^n(x_0), \dots,$$

which collect the population value of successive generations. The purpose is to compute an initial segment of the orbit for a given initial population x_0. It has been identified that choosing $a = 4$ is a chaotic case. The main problem is that its value is sensitive to small variations of its variables. The result of computing orbits for the same initial value $x_0 = 0.671875$, in simple and double precision in the C programming language is shown in Table 1. Also, Table 1 shows the exact result and the value obtained using our FC++ implementation. As it can be noticed the tables are equal up to $n = 7$. From row 8th up to 39th the double, exact and FC++ column report equal results. From row 40th the C double implementations show a small deviation from the exact result and at the last 63rd row this deviation is more evident. It is worth to mention that every exact real number computation implementation must produce the correct result as is the case in our implementation. The main drawback of the functional languages implementations is the execution time taken to compute the orbits. The implementation presented here improves the efficiency to compute the result as can be seen in column five of table 1 compared to column six of the same table. It is fair to say that implementations like iRRAM, programmed on C++, run much faster than the one presented here, however we have not explore the different mechanisms employed by iRRAM. In a further version of this paper we will present another implementation considering mainly efficiency. In this first version, we wanted to show that it is possible to go from the theory to the practice in a smooth way.

n	Simple precision	Double precision	FC++ implementations	Exact Result	FC++ Time	Sign Digit Time
0	0.671875	0.671875	0.671875	0.671875	0	0
1	0.881836	0.881836	0.881836	0.881836	3 ms	20 ms
2	0.416805	0.416805	0.416805	0.416805	5 ms	30 ms
3	0.972315	0.972315	0.972315	0.972315	13 ms	80 ms
4	0.107676	0.107676	0.107676	0.107676	22 ms	190 ms
5	0.384327	0.384327	0.384327	0.384327	42 ms	550 ms
6	0.946479	0.946479	0.946479	0.946479	61 ms	1.13 s
7	0.202625	0.202625	0.202625	0.202625	103 ms	1.14 s
8	0.646272	0.646273	0.646273	0.646273	138 ms	2.24 s
9	0.914417	0.914416	0.914416	0.914416	189 ms	4.77 s
10	0.313033	0.313037	0.313037	0.313037	222 ms	10.69 s
11	0.860174	0.860179	0.860179	0.860179	352 ms	24.7 s
12	0.481098	0.481084	0.481084	0.481084	607 ms	53.16 s
13	0.998570	0.998569	0.998569	0.998569	850 ms	1.78 min
14	0.005708	0.005716	0.005716	0.005716	854 ms	3.10 min
15	0.022702	0.022735	0.022735	0.022735	870 ms	4.54 min
16	0.088747	0.088875	0.088875	0.088875	1.13 s	9.80 min
17	0.323485	0.323907	0.323907	0.323907	1.42 s	20.43 min
18	0.875370	0.875965	0.875965	0.875965	2.46 s	46.59 min
19	0.436386	0.434601	0.434601	0.434601	2.47 s	$\geq$ 1 hour
20	0.983813	0.982892	0.982892	0.982892	2.53 s	$\geq$ 1 hour
25	0.652836	0.757549	0.757549	0.757549	8.23 s	$\geq$ 1 hour
30	0.934926	0.481445	0.481445	0.481445	39.49 s	$\geq$ 2 hours
35	0.848152	0.313159	0.313159	0.313159	2.25 min	$\geq$ 2 hours
39	0.014638	0.006038	0.006038	0.006038	5.66 min	$\geq$ 3 hours
40	0.057695	0.024007	0.024009	0.024009	5.75 min	$\geq$ 3 hours
50	0.042173	0.629401	0.625028	0.625028	10.83 min	$\geq$ 4 hours
55	0.108415	0.749775	0.615752	0.615752	112 min	$\geq$ 5 hours
60	0.934518	0.757153	0.315445	0.315445	15.7 min	$\geq$ 6 hours
63	0.770667	0.690457	0.996571	0.996571	18.38 min	$\geq$ 6 hours

Table 1. Results of computing the logistic map for simple and double precision in the C programming language, our implementation and the exact result. From values $n = 8$ and $n = 40$ the simple and double precision respectively deviate from the exact result. Additionally, the two last columns show a time comparison result taken to compute the values in our implementation and a sign digit representation.

4 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. S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.
2. A. Bauer and I. Kavkler. Implementing real numbers with rz. In *Proceedings of the Fourth International Conference on Computability and Complexity in Analysis, CCA*, pages 365–384. ENTCS, 2008.
3. A. Ciaffaglione and P. D. Gianantonio. A certified, corecursive implementation of exact real numbers. *Theoretical Computer Science*, 351(1):39–51, 2006.
4. R. L. Devaney. *An Introduction to Chaotical Dynamical Systems*. Addison-Wesley, California, 2do edition, 1989.
5. M. H. Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162(1):79–115, August 1996.
6. P. Pélissier G. Hanrot, V. Lefévre and P. Zimmermann. The MPFR library. INRIA. http://mpfr.org.
7. C. A. Gunter. *Semantics of Programming Languages*. The MIT Press, 1992.
8. B. Lambov. The reallib project. BRICS, University of Aarhus. http://brics.dk/~barnie/RealLib.
9. J. R. Marcial-Romero. *Semantics of a sequential language for exact real-number computation*. PhD thesis, University of Birmingham, December 2004.
10. J. R. Marcial-Romero and M. H. Escardó. Semantics of a sequential language for exact real-number computation. *Theoretical Computer Science*, 379(1-2):120–141, 2007.
11. J. R. Marcial-Romero, J. A. Hernández, and Héctor A. Montes-Venegas. Comparing implementations of a calculator for exact real number computation. In Harald Ganzinger, editor, *Proceedings of the Mexican International Conference on Computer Science, ENC*, pages 13–23. IEEE Computer Society Press, July 2009.
12. J. R. Marcial-Romero and A. Moshier. Sequential real number computation and recursive relations. In *Proceedings of the Fourth International Conference on Computability and Complexity in Analysis, CCA*, pages 171–189. ENTCS, 2008.

13. J. R. Marcial-Romero and A. Moshier. Sequential real number computation and recursive relations. *Mathematical Logic Quarterly*, 54(5):492–507, 2008.
14. N. M uller. iRRAM - exact arithmetic in C++. Universit at Trier. http://www.informatik.uni-trier.de/iRRAM.
15. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(1):223–255, 1977.
16. Dave Plume. A calculator for exact real number computation. 4th Year Project Report, Department of Computer Science and Artificial Intelligence, University of Edinburgh, 1998.
17. P. J. Potts, A. Edalat, and M.H Escardó. Semantics of exact real arithmetic. In *In Proceedings of the Twelveth Annual IEEE Symposium on Logic In Computer Science*. IEEE Computer Society Press, 1997.