# Comparing Implementations of a Calculator for Exact Real Number Computation

José Raymundo Marcial-Romero*, José Antonio Hernández Servín* y Héctor Alejandro Montes-Venegas*

**Recepción: 10 de agosto de 2011**
**Aceptación: 27 de mayo de 2012**

*** Facultad de Ingeniería, Universidad Autónoma del Estado de México, México**
**Correo electrónico: rmarcial@fi.uaemex.mx; xosehernandez@fi.uaemex.mx y h.a.montes@fi.uaemex.mx**

**Comparando implementaciones de una calculadora para la computación de números reales exactos**

**Resumen.** Al ser uno de los primeros lenguajes de programación teóricos para el cómputo con números reales, Real PCF demostró ser impráctico debido a los constructores paralelos que necesita para el cálculo de funciones básicas. Posteriormente, se propuso LRT como una variante de Real PCF el cual evita el uso de constructores paralelos introduciendo un constructor no determinista dentro del lenguaje. En este artículo se presenta la implementación de una calculadora para el cómputo con números reales exactos basada en LRT y se compara su eficacia con una aplicación de números reales estándar en un lenguaje de programación imperativo. Finalmente, la implementación se compara con una implementación estándar de computación de números reales exactos, basada en la representación de dígitos con signo, que a su vez se basa sobre la computación de números reales exactos.

**Palabras clave:** lenguajes de programación, cómputo con números reales, programación funcional.

**Abstract.** As one of the first theoretical programming languages for exact real number computation, Real PCF was shown to be impractical due to the parallel construct needed for even basic operations. Later, LRT was proposed as a variant of Real PCF avoiding the parallelism by introducing a non-deterministic constructor into the language. In this paper we present an implementation of a calculator for exact real number computation based on LRT and compare its efficacy with an application of the standard use of real numbers in an imperative programming language. Finally, our implementation is compared with a standard implementation of exact real number computation based on the sign digit representation, which is also based on exact real number computation.

**Key words:** programming languages, real number computation, functional programming.

## Introduction

In the last two decades, several researches have presented different frameworks for a programming language for exact real number computation (Potts *et al.,* 1997; Escardó 1996; Boehm, and Cartwright, 1990; Weihrauch, 2000). Particularly, Escardó (1996) proposed a theoretical programming language for exact real number computation, called Real PCF, with an abstract data type (representation independent) but a parallel constructor with a high computational cost both in time and storage, which is needed even for basic operations like addition. A further research project was to develop a theoretical programming language avoiding parallel constructors. Marcial-Romero (2004) and Marcial-Romero and Escardó (2004) presented a sequential non-deterministic programming language for exact real number computations called LRT. LRT can be seen as Real PCF (PCF stands for Programmable Computable Functions) without the parallel constructor and a non deterministic constructor added. The non-determinism allows avoiding the parallelism; a further explanation can be consulted in Marcial-Romero and Escardó (2007). Additionally, the non-determinism not only allows to define functions in LTR but also relations, therefore Marcial-Romero and Moshier (2008a and b) established a computational adequacy framework between LRT and Brattka

relational setting (Brattka, 1996). The relational setting of LRT, implicitly defines computable first order functions, thus they can be implemented in the language.

In his paper, Bauer and Kavkler (2008) refers to the fact that a direction in constructive mathematics is "*get closer to the practice*" without disconnecting the theory and the practice. Even more, he stated:

> [...] move practice closer to theory by making sure that practical implementations follow formal specifications that are computed directly from theoretical models (Bauer and Kavkler, 2008: 2).

In this paper we present an implementation of the LRT operational semantics and, compare firstly its efficacy against an application on an imperative programming language; and secondly its efficiency on a basic calculator (addition, subtraction, multiplication and division) implemented in LRT against a standard implementation using sign digit representation. The main difference between both calculators is that the former has a formal theoretical model while the further does not. The base language used to implement LRT was Haskell, which include characteristics to easily implement languages like LRT. Among the main characteristics of Haskell are the lazy evaluation and the natural use of infinite lists. The algorithms implemented are based on Plume's thesis (Plume, 1998) that converges faster than the algorithms proposed in Marcial-Romero (2004). Our motivation comes from Bauer and Kavkler (2008) suggestion stated above and we believe that by using faster libraries as the other implementations do, we will improve our calculator.

The paper is organized as follows: in Section 2 the language LRT is described. In Section 3 an example of a program in the language is explained. In Section 4, the main implementation details are presented. In Section 5 the four basic programs of the calculator are presented. In Section 6 the comparison are presented. Finally the conclusions are established.

## 1. The LRT Language

LRT amounts to the language considered by Escardó (1996) with the *parallel conditional* removed and a constant $\mathit{rtest}_{l,r}$ added. This is a call-by-name language. Because exact real-number computations are infinite, and there are no canonical forms for partial real-number computations, it is not clear what a call-by-value operational semantics ought to be.

### *1.1 Syntax*

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

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

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.[1] 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, predecessor 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 ($\lambda x{:}\ t.P \mid PP \mid YP$) where the first denotes abstraction, the second application and the third recursion.

The mathematical objects which describe the ***cons***, ***tail*** and ***rtest*** constructors are presented below. The others are the well known PCF constructors and can be consulted at Gunter (1992) and Plotkin (1977).

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

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

That is, rescale and translate the interval $[-1, 1]$ so that it becomes $[\underline{a}, \overline{a}]$, and define $\mathit{cons}_{[\underline{a},\,\overline{a}]}([\underline{x}, \overline{x}])$ 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 $x$ is used 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:

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

1. Represented by $I$ which denotes the set of intervals in $[-1, 1]$, as it was shown in Marcial-Romero (2004) the complete real line can be easily represented in this language, even more the implementation presented here considers the complete real line.

The function $tail_a D \to D$ is a left inverse, *i.e.*

$$tail_a(cons_a(x)) = x$$

More precisely, the following left inverse is taken, where $\kappa_a$ is $\overline{a}, \underline{a}$ and $\tau_a$ is $\overline{a}, \underline{a}$

$$tail_{\overline{a}}(cons_a(x)) = x$$

$$tail_{[\underline{a},\,\overline{a}]}(x) = \max\left(-1, \min\left(\frac{2x + \tau_a}{\kappa_a}, 1\right)\right)$$

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 Escardó (1996). It is worthy to mention that an infinite shrinking sequence of *cons* intervals represents 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 $rtest_{l,r}$: $D \to \{true, false\}$, where $l<r$ are rational numbers, can be formulated as:

$$rtest_{l,r}(x) = \begin{cases} true, & if\ x \subseteq [-1, l] \\ true\ or\ false, & if\ x \subseteq (l, r) \\ false, & if\ x \subseteq [r,1) \end{cases}$$

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

### *1.2 Operational Semantics*

A small-step style operational semantics for LRT is considered. The one-step reduction relation $\to$ is defined to be the least relation containing the one-step reduction rules for evaluation of PCF (Plotkin, 1977) together with those given below.

Firstly, some preliminaries are introduced. For intervals $a$ and $b$ in [−1, 1], define

$$ab = cons_a(b)$$

Where *cons* is the function defined previously. This operation is associative, and has the interval [−1, 1] (denoted by $\perp$) as its neutral element such that (Escardó, 1996):

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

In the interval domain literature Abramsky and Jung (1994), $a \sqsubseteq b$ iff $b \subseteq a$. Moreover,

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

and this $c$ is unique if a has non-zero length; in this case $c$ is denoted by $b\backslash a$.

For intervals $a$ and $b$, define:

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

and

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

With this notation, the rules for Real PCF as defined in Escardó (1996) are:

| | | |
|---|---|---|
| $cons_a(cons_b M) \to cons_{ab} M$ | | (1) |
| $cons_a M \to cons_a M'$ | If $M \to M'$ | (2) |
| $tail_a(cons_b M) \to Y\ cons_{[-1,\,0]} M$ | If $b \leq a$ | (3) |
| $tail_a(cons_b M) \to Y\ cons_{[0,\,1]} M$ | If $b \geq a$ | (4) |
| $tail_a(cons_b M) \to cons_{b\backslash a} M$ | If $a \sqsubseteq b$ and a≠b | (5) |
| $tail_a M \to tail_a M'$ | If $M \to M'$ | (6) |
| $if\ true\ M\ N \to$ M | | (7) |
| $if\ false\ M\ N \to$ N | | (8) |
| $if\ M\ N_1\ N_2 \to if\ M'\ N_1\ N_2$ | If $M \to M'$ | (9) |

For our langurtestge LRT, add:

| | | |
|---|---|---|
| $rtest_{l,r}(cons_a M) \to true$ | If $\overline{a} < r$ | (10) |
| $rtest_{l,r}(cons_a M) \to false$ | If $l < \underline{a}$ | (11) |
| $rtest_{l,r} M \to rtest_{l,r} M'$ | If $M \to M'$ | (12) |

*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.[2]

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

3. Rule (3) represents the fact that it is already known that the rest of the real number being looking for is an infinite sequence in the interval [−1, 0], *i.e.:*

$$Y\ cons_{[-1,\,0]} = cons_{[-1,\,0]}(cons_{[-1,\,0]}(\cdots))$$

2. See Escardó (1996) for a further discussion including a geometrical interpretation.

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 (11) and (12) can be applied.

8. Rules (10)-(12) cannot be made deterministic given the particular computational adequacy formulation which is proved in Marcial-Romero and Escardó (2007).

9. In practice, one would like to avoid divergent computations by considering a strategy for application of the rules. In Marcial-Romero and Escardó (2007) total correctness of basic algorithms and in Marcial-Romero and Moshier (2008) 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 Marcial-Romero and Escardó 2007; Marcial-Romero and Moshier (2008).

## 2. Running example

In order to motivate the use of the operational semantics given in the previous section, an example showing how to compute a real valued function is presented.

In the programming language considered in Escardó (1996), the *average* operation $(- \oplus -)$: $[0, 1] \times [0, 1] \to [0, 1]$ defined by:

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

can be implemented as follows:

$$\begin{aligned} x \oplus y = \ & \textit{pif } x < c \\ & \quad \textit{then pif } y < c \\ & \qquad \textit{then } cons_L\ (tail_L(x) \oplus tail_L(y)) \\ & \qquad \textit{else } cons_C\ (tail_L(x) \oplus tail_R(y)) \\ & \quad \textit{then pif } y < c \\ & \qquad \textit{then } cons_C\ (tail_R(x) \oplus tail_L(y)) \\ & \qquad \textit{else } cons_R\ (tail_R(x) \oplus tail_R(y)). \end{aligned}$$

Here

$$c = \frac{1}{2},\ L = [0, c],\ C = \left[\frac{1}{4}, \frac{3}{4}\right],\ R = [c, 1]$$

Because equality on real numbers is undecidable, the relation $x < c$ is undefined (or diverges, or denotes $\perp$) if $x = c$. In order to compensate for this, one uses a *parallel conditional* such that

$$\textit{pif } \perp \textit{ then } z \textit{ else } z = z$$

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 parallel conditional. For example, $1/2$ is both in $L$ and $R$, and an unfolding of the program for $x = y = 1/2$ *gives*

$$\begin{aligned} 1/2 \oplus 1/2 = \ & \textit{pif } \perp \\ & \quad \textit{then pif } \perp \\ & \qquad \textit{then } cons_L(1 \oplus 1) \\ & \qquad \textit{else } cons_C\ (1 \oplus 0) \\ & \quad \textit{then pif } \perp \\ & \qquad \textit{then } cons_C\ (0 \oplus 1) \\ & \qquad \textit{else } cons_R\ (0 \oplus 0). \end{aligned}$$

All branches of the conditionals evaluate to $1/2$, but in an infinite number of steps. This can be seen as follows: a repeat unfolding of $1 \oplus 1$ gives the infinite expression $cons_R\ (cons_R\ (cons_R \cdots))$. Denotationally speaking, the program computes the unique fixed point of $cons_R$, which is 1. Operationally speaking, the first unfolding says that the result of the computation, whatever it is, lives in the interval $R$, because, by definition, the image of $cons_R$ is $R$; the second unfolding says that the result is in the right half of the interval $R$, *i.e.* in the interval $[\frac{3}{4}, 1]$ the third unfolding tells us that the result is in the interval $[\frac{7}{8}, 1]$, and so on. Thus, the operational semantics applied to $1 \oplus 1$ produces a shrinking sequence of intervals converging to 1. The other cases are analogous.

Of course, a drawback of such a recursive definition is that, during evaluation, the number of parallel processes is exponential in the number of unfoldings. In order to overcome this, we switch back to the usual sequential conditional, and replace the partial less-than test by the *multi-valued* test discussed in the previous section.

$$\begin{aligned} average(x, y) = \ & \textit{if } rtest_{l,r}(x) \\ & \quad \textit{then if } rtest_{l,r}(y) \\ & \qquad \textit{then } cons_L\ (average\ (tail_L(x), tail_L(y)) \\ & \qquad \textit{else } cons_C\ (average\ (tail_L(x), tail_R(y))) \\ & \quad \textit{else if } rtest_{l,r}(y) \\ & \qquad \textit{then } cons_C\ (average\ (tail_R(x), tail_L(y))) \\ & \qquad \textit{else } cons_R\ (average\ (tail_R\ (x), tail_R\ (y))) \end{aligned}$$

The intuition behind this program is similar. 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 Marcial-Romero (2004).

## 3. The Implementation

In this section, the Haskell implementation of the operational semantics described in the previous section is presented. Also, the implementation of the algorithm for the average function presented in Marcial-Romero and Escardó (2007) and the rate of convergence of this algorithm compared to the three digit representation algorithm implemented by Plume (1998) is discussed.

The real numbers are represented in Haskell by the datatype *CREAL* 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 representing 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 Haskell in the following way:

$$\begin{aligned} &data\ CoTa = Cons(Rational, Rational)\\ &CREAL \quad = ([CoTa], Integer) \end{aligned}$$

Notice that we have not restricted the rational intervals to be in the interval [−1, 1], however their use in the implementation does. The *cons* and *tail* operations are easily implemented as follows:

$$cons :: (Rational, Rational) \to (Rational, Rational) \to (Rational, Rational)$$

$$cons(a_1, a_2)(x_1, x_2) = ((y_1 \times x_1) + y_2, (y_1 \times x_2) + y_2)$$
$$\text{where } \{y_1 = (a_2 - a_1)/2$$
$$y_2 = (a_2 + a_1)/2\}$$

$$tail :: (Rational, Rational) \to (Rational, Rational) \to (Rational, Rational)$$

$$tail(a_1, a_2)(x_1, x_2) = \left(max\left(min\left(\frac{2 \times x_2 + d}{c}\right), 1\right), (-1)\right),$$

$$min\left(max\left(\frac{2 \times x_2 + d}{c}\right), -1\right), (1)\right)$$

$$\text{where } \{ c = a_2 - a_1$$
$$d = -a - a_1\}$$

These implementations take two tuples of rational numbers, which represent the subsets on the interval [−1, 1] and return a new tuple of rational numbers. The *if* operator is the already predefined operator in Haskell.

Notice that the non-deterministic *rtest* operator can be implemented in two ways as pointed out in the previous section:

$$\begin{aligned} &rtest :: (Rational, Rational) \to [CoTa] \to bool\\ &\quad rtest\ l\ r\ (cons(x_1, x_2): xs)\\ &\quad | x_2 \le r = true\\ &\quad | x_1 \ge l = false \end{aligned}$$

$$\begin{aligned} &rtest` :: (Rational, Rational) \to [CoTa] \to bool\\ &\quad rtest`\ l\ r\ (cons(x_1, x_2): xs)\\ &\quad | x_1 \ge l = false\\ &\quad | x_2 \le r = true \end{aligned}$$

However adequacy of the language as presented in Marcial-Romero and Escardó (2007) ensures that any of them is a correct and totally convergent implementation.

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. This is achieved by the first rule of the operational semantics together with the other operational rules implemented as follows:

$$\begin{aligned} &evaluacion :: [CoTa] \to [CoTa]\\ &evaluacion\ (cons(a, b): [\,]) = [cons(a, b)]\\ &evaluacion\ (cons(a, b): cons(c, d): xs) = cons(cons(a, b)(c, d)): xs\\ &evaluacion\ (cons(a, b): xs) = evaluacion(cons(a, b): evaluacion\ xs) \end{aligned}$$

$$\begin{aligned} &evaluacion\ (tail(a, b): cons(c, d): xs) = if\ (b \le c)\\ &\qquad then [cons(-1, 0), cons(-1, 0), \cdots)]\\ &\qquad if\ (a > d)\\ &\qquad\quad then [cons(0, 1), cons(0, 1), \cdots)]\\ &\qquad if\ ((a < c)\ \&\&(d < b))\ ||\ ((a \le c)\\ &\qquad\quad \&\&(d < b)\ ||\ ((a < c)\ \&\&\ (d \le b))\\ &\qquad\quad then\ cons(tail(a, b)(c, d)): xs\\ &\qquad if\ (a < c)\ \&\&\ (b < d)\\ &\qquad\quad then\ cons(tail(a, b)(c, b)):\\ &\qquad\quad (tail(tail(c, d)(c, b))): xs\\ &\qquad if\ (c < a)\ \&\&\ (d < b)\\ &\qquad\quad then\ cons(tail(a, b)(a, d)):\\ &\qquad\quad (tail(tail(c, d)(a, d))): xs\\ &\qquad otherwise\ xs \end{aligned}$$

$$evaluacion(tail(a, b): xs) = evaluacion(tail(a, b): evaluacion(xs))$$

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 rational numbers at the head of the *mantissa* by 2 to the power of the exponent.

A real valued function $f$: $(CREAL)^n \rightarrow (CREAL)^m$ takes as input $n$ pairs of the type *CREAL* and returns m pairs of the type *CREAL*.

For example an implementation of the *average* function defined in section 3 is:

$$average(x, y) = if\ rtest\ -\frac{3}{4}, \frac{3}{4}\ (x)$$

$$then\ \text{if}\ rtest\ -\frac{3}{4}, \frac{3}{4}\ (y)$$

$$then\ cons(-1, \frac{3}{4})(average(tail(-1, \frac{3}{4})(x), tail(-1, \frac{3}{4})(y)))$$

$$else\ cons(-\frac{3}{4}, \frac{3}{4})(average(tail(-1, \frac{3}{4})(x), tail(-\frac{3}{4}, 1)(y)))$$

$$else\ if\ rtest\ -\frac{3}{4}, \frac{3}{4}\ (y)$$

$$then\ cons(-\frac{3}{4}, \frac{3}{4})(average(tail(-\frac{3}{4}, 1)(x), tail(-1, \frac{3}{4})(y)))$$

$$else\ cons(-\frac{3}{4}, 1)(average\ (tail(-\frac{3}{4}, 1)(x), tail(-\frac{3}{4}, 1)(y)))$$

It can be noticed that the rational numbers $l$ and $r$ where substituted by −3/4 and 3/4 respectively. These numbers can be arbitrarily fixed if the conditions $-1 < l < r < 1$ are considered.[3] However at the implementation level not always the shortest algorithm guarantees a fast convergence to the desired precision. In this case because the rate of convergence of this program is $(3/4)^n$, this program converges slower than a program whose rate of convergence is $(1/2)^n$. Considering the conditions stated previously, it can be easily shown that for this particular program, there are no values of $l$ and $r$ which improve or reach the rate of convergence $(1/2)^n$ in all possible executions of the program.

**Lemma 1** There are not values of $l$ and $r$ in [−1, 1] such that $-1 < l < r < 1$ and every execution path of the *average* program converges at rate of convergence of $(1/2)^n$ or faster.

***Proof***

To guarantee convergence at range of $1/2^n$ or faster, the rescaling fa constor of the $cons_{[\underline{x}, \overline{x}]}$ equation should satisfy:

$$((\overline{x}, \underline{x}])/2) \leq (1/2)$$

by the rescaling factor of $cons_L$

$$((r + 1)/2) \leq (1/2) \Rightarrow r \leq 0$$

and the rescaling factor of $cons_R$

$$\left(\frac{1 - l}{2}\right) \leq (1/2) \Rightarrow l \geq 0$$

Contradicting the assumption that $l < r$. ■

The above lemma does not imply that there is not a program which converges faster, in fact the above program was presented in Marcial-Romero (2004) only as evidence that basic operations like additions can be implemented in LRT.

If we consider the sign digit algorithm Plume (1998) for the *average* function which guarantees a rate of convergence of $1/2^n$ and translate it to LRT, we have the following program:

$$faverage :: CREAL \rightarrow CREAL \rightarrow CREAL$$

$$faverage(x, y) = if\ rtest\ -\frac{1}{2}\ 0\ (x)$$

$$then$$

$$if\ rtest -1/2\ 0\ (y)$$

$$then\ cons(-1, 0)(faverage(tail(-1, 0)x, tail(-1, 0)y))$$

$$else$$

$$if\ rtest\ 0\ 1/2\ (y)$$

$$then\ cons\left(-\frac{3}{4}, \frac{1}{4}\right)\left(faverage\left(tail(-1, 0)\ x, tail\left(-\frac{1}{2}, \frac{1}{2}\right)y\right)\right)$$

$$else\ cons\left(-\frac{1}{2}, \frac{1}{2}\right)(faverage(tail(-1, 0)x, tail(0, 1)y))$$

$$else$$

$$if\ rtest\ 0\ 1/2\ (x)$$

$$then\ if\ rtest\ -\frac{1}{2}\ 0\ (y)$$

$$then\ cons\left(-\frac{3}{4}, \frac{1}{4}\right)\left(faverage\left(tail\left(-\frac{1}{2}, \frac{1}{2}\right)x, tail\left(-\frac{1}{2}, \frac{1}{2}\right)y\right)\right)$$

$$else\ if\ rtest\ 0\ \frac{1}{2}\ (y)$$

$$then\ cons\left(-\frac{1}{2}, \frac{1}{2}\right)\left(faverage\left(tail\left(-\frac{1}{2}, \frac{1}{2}\right)x, tail\left(-\frac{1}{2}, \frac{1}{2}\right)y\right)\right)$$

$$else\ cons\left(-\frac{1}{4}, \frac{3}{4}\right)\left(faverage\left(tail\left(-\frac{1}{2}, \frac{1}{2}\right)x, tail(0, 1)y\right)\right)$$

$$else\ if\ rtest\ -\frac{1}{2}\ 0\ (y)$$

3. See Marcial-Romero and Escardó (2004) for a discussion.

$$\textit{then conss}\left(-\frac{1}{2}, \frac{1}{2}\right)\left(\textit{faverage}\left(\textit{tail}\ (0, 1)x, \textit{tail}(-1, 0)y\right)\right)$$

$$\textit{else if rtest}\ 0\ \frac{1}{2}\ (y)$$

$$\textit{then cons}\left(-\frac{1}{4}, \frac{3}{4}\right)\left(\textit{faverage}\left(\textit{tail}\ (0, 1)x, \textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right)y\right)\right)$$

$$\textit{else cons}(0, 1)(\textit{faverage}(\textit{tail}(0, 1)x, \textit{tail}(0, 1)y))$$

Program *Average* divides the interval [−1, 1] in two overlapping intervals ($\left[-1, \frac{3}{4}\right]$) and ($\left[-\frac{3}{4}, 1\right]$) resulting in four cases in the program. Program *faverage* divides the interval [−1, 1] it three overlapping intervals ([−1, 0], [−1/2, 1/2] and [0, 1]) resulting in nine cases in the program. Table 1 presents the time reported by the Glasgow Haskell compiler doing *n* different average operations in both programs at precision $1E-11$. For example, if ⊕ denotes any of the *average* functions, the result of 1/11 ⊕ 2/13 ⊕ 3/15 ⊕ 2/13⋯21/51 is reported in the 20 operation`s row. Although program *average* has less code lines than program *faverage*, the rate of convergence in program *faverage* is better.

## 4. The Calculator

In this section we just present the implementations of addition and division in our calculator, the other basic operations (subtraction and multiplication) are implemented similarly, the reader can download the implementation from http://fi.uaemex.mx/rmarcial/LRT. The domain and codomain of the implementations are the whole real line, hence we use the *CREAL* datatype defined in section 4.

### 4.1 The addition function

The addition function is defined from the *faverage* function and a pair of auxiliary function.

$$\textit{addition} :: \textit{CREAL} \to \textit{CREAL} \to \textit{CREAL}$$
$$\textit{addition}\ (xs, a)(ys, b) = (\textit{fst}(aux), \textit{snd}(aux * 2))$$
$$\textit{where aux} = \textit{auxaverage}\ (xs, a)(ys, b)$$

**Table 1.** Time reported by the Haskell Glasgow compiler at doing *n* average operations in both programs with precision $1E-11$

| Number of Operations | Time Reported | |
|---|---|---|
| | Program *Average* | Program *faverage* |
| 20 | 47.50 sec | 0.032 sec |
| 30 | 73.49 sec | 0.112 sec |
| 40 | 99.56 sec | 0.136 sec |
| 50 | 129.51 sec | 0.144 sec |
| 100 | 289.49 sec | 0.344 sec |

$$\textit{auxaverage} :: \textit{CREAL} \to \textit{CREAL} \to \textit{CREAL}$$
$$\textit{auxaverage}\ (xs, i)(ys, j)$$
$$\mid i > j = (\textit{faverage}(xs, \textit{recorre}(ys, \textit{logBase}\ 2\ i) - (\textit{logBase}\ 2\ j))), i)$$
$$\mid i < j = (\textit{faverage}(\textit{recorre}(xs, \textit{logBase}\ 2\ j) - (\textit{logBase}\ 2\ i))), ys)$$
$$\mid i == j = (\textit{faverage}(xs, ys), i)$$

In order to add two real numbers, firstly translate them to the interval [−1, 1] using the mantissa-exponent datatype presented in section 4. Once the translation is done, the *faverage* operation is applied. Because the *faverage* operation divides the sum of the two numbers by two, the exponent of the result is multiply by two, to obtain the required result.

### 4.2. The division implementation

An implementation of division of two real numbers is presented. Plume (1998) algorithm for defining the division is used. To simplify discussion, Plume defines division on the intervals

$$\textit{division}: [-1, 1] \times \left[\frac{1}{4}, 1\right] \to [-4, 4]$$

to keep the result in a bounded interval, because taking inputs from the intervals [−1, 1] results on an output on the interval (−∞, ∞). In that sense we give a definition for ***division***$(x, 4y)$ to keep the result in [−1, 1]. For example ***division***$(1/4, 3/8)$ should produce as a result $(1/4)/(4\ (3/8)) = (1/4)/(3/2) = 1/6$. We multiply the exponent of the final result by four to obtain the required result.

$$\textit{division} :: \textit{CREAL} \to \textit{CREAL} \to \textit{CREAL}$$
$$\textit{division}(xs, a)(ys, b) = (\textit{fdiv}(xs, ys), (a/b)*4)$$
$$\textit{fdiv}(x: xs)(y: ys) = \textit{if}\ (\textit{fst}(\textit{aux}1))$$
$$\textit{then}$$
$$\textit{if}(\textit{fst}(\textit{aux}2))$$
$$\textit{then if}\ (\textit{fst}(\textit{aux}3))$$
$$\textit{then if}\ (\textit{fst}(\textit{aux}4))$$
$$\textit{then cons}(-1, 0): (\textit{fdiv}\left(\left(\textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{aux}10\right), (y: ys)\right)$$
$$\textit{else cons}\left(-\frac{3}{4}, \frac{1}{4}\right): (\textit{fdiv}\ ((\textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{aux}4), (y: ys)))$$
$$\textit{else if}\ (\textit{fst}(\textit{aux}5))$$
$$\textit{then cons}\left(-\frac{3}{4}, \frac{1}{4}\right): (\textit{fdiv}((\textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{aux}5), (y: ys)))$$
$$\textit{else cons}\left(-\frac{5}{8}, \frac{3}{8}\right): (\textit{fdiv}((\textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{aux}13), (y: ys)))$$
$$\textit{else cons}\left(-\frac{1}{2}, \frac{1}{2}\right): (\textit{fdiv}((\textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{aux}2), (y: ys)))$$

*else*
*if* ( *fst*(*aux*6))

$$\textit{then cons}\left(-\frac{1}{2}, \frac{1}{2}\right): (\textit{fdiv}((\textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{aux}2), (y: ys)))$$

*else if* ( *fst*(*aux*7))

*then if* ( *fst*(*aux*8))

$$\textit{then cons}\left(-\frac{3}{8}, \frac{5}{8}\right): (\textit{fdiv}\ ((\textit{aux}11),(y: ys)))$$

$$\textit{else cons}\left(-\frac{1}{4}, \frac{3}{4}\right): (\textit{fdiv}\left(\left(\textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{aux}8\right), (y: ys)\right)\right)$$

*else if* ( *fst*(*aux*9))

$$\textit{then cons}\left(-\frac{1}{4}, \frac{4}{4}\right): (\textit{fdiv}\left(\left(\textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{aux}9\right), (y: ys)\right)\right)$$

$$\textit{else cons}(0, 1): (\textit{fdiv}\left(\left(\textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{aux}12\right), (y: ys)\right)\right)$$

$$\textit{where aux}1 = \textit{rtest}\left(-\frac{1}{2}\right)\left(\frac{1}{2}\right)(x: xs)$$

$$\textit{aux}2 = \textit{rtest}\left(-\frac{1}{2}\right)\left(\frac{1}{4}\right)(\textit{aux}1)$$

$$\textit{aux}3 = \textit{rtest}\left(-\frac{1}{2}\right)\left(\frac{1}{2}\right)\textit{tail}(\textit{farest}(\text{snd}(\textit{aux}2))(y: ys))$$

$$\textit{aux}4 = \textit{rtest}\left(-\frac{1}{2}\right)\left(\frac{1}{4}\right)(\textit{aux}3)$$

$$\textit{aux}5 = \textit{rtest}\left(\frac{1}{4}\right)\left(\frac{1}{2}\right)(\textit{aux}3)$$

$$\textit{aux}6 = \textit{rtest}\left(\frac{1}{4}\right)\left(\frac{1}{2}\right)(\textit{aux}1)$$

$$\textit{aux}7 = \textit{rtest}\left(-\frac{1}{2}\right)\left(\frac{1}{2}\right)\textit{tail}(\textit{farest}(\text{snd}(\textit{aux}6))\ (y: ys))$$

$$\textit{aux}8 = \textit{rtest}\left(-\frac{1}{2}\right)\left(\frac{1}{4}\right)(\textit{aux}7)$$

$$\textit{aux}9 = \textit{rtest}\left(\frac{1}{4}\right)\left(\frac{1}{2}\right)(\textit{aux}7)$$

$$\textit{aux}10 = \textit{tail}(\textit{faverage}(\textit{snd}(\textit{aux}4))(y: ys))$$

$$\textit{aux}11 = \textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{tail}(\textit{farest}(\textit{snd}(\textit{aux}6)), \textit{cons}\left(-\frac{1}{2}, \frac{1}{2}\right)(y: ys))$$

$$\textit{aux}12 = \textit{tail}(\textit{frest}(\text{snd}(\textit{aux}9))(y: ys))$$

$$\textit{aux}13 = \textit{tail}\left(-\frac{1}{2}, \frac{1}{2}\right): \textit{tail}(\textit{faverage}(\textit{snd}(\textit{aux}2)), \textit{cons}\left(-\frac{1}{2}, \frac{1}{2}\right)(y: ys))$$

Trigonometric operations like *sin*, *cosine*, *tangent* among others were also programmed in LRT using Plume's algorithms. The reader can download either the text modules or the graphical interface from http://fi.uaemex.mx/rmarcial/LRT.

## 5. Comparing our calculator

Two different comparisons are performed to test our implementation. The first one compares the common use of real numbers in the *C* programming language against our exact real number implementation. The second one compares a three digit bit implementation of exact real numbers again our implementation. All the comparisons were performed on a MacBook with processor of 2.4 GHz Intel Core 2 Duo and memory of 2 GB.

### *5.1. The logistic Map*

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

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

for a given constant $a$. Devaney (Devaney, R. L. 1989) 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 is, given an initial value $x_0$, to compute the orbit

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

which collects 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 2. Also, table 2 shows the exact result and the value obtained using our calculator. As it can be noticed the tables are equal up to $n = 7$. From row 8th up to 39th the double, exact and LRT column report equal results. From row 40th the *C* double precision shows a small deviation from the exact result and at the last 63rd row this deviation is evident enough. It is worth to mention that every exact real number computation implemented must produce the correct result as is the case in our calculator. The main drawback is the execution time that our implementation takes to compute the orbits. However, in this first version of our implementation, the goal is not to look for the most efficient algorithms for

**Table 2.** Results of computing the logistic map for simple and double precision in the *C* programming language, and our implementation and the exact result. From values $n = 8$ and $n = 40$ the simple and double precision respectively deviate from the exact result.

| $n$ | Simple precision | Double precision | LTR result | Exact result |
|---|---|---|---|---|
| 0 | 0.671875 | 0.671875 | 0.671875 | 0.671875 |
| 1 | 0.881836 | 0.881836 | 0.881836 | 0.881836 |
| 2 | 0.416805 | 0.416805 | 0.416805 | 0.416805 |
| 3 | 0.972315 | 0.972315 | 0.972315 | 0.972315 |
| 4 | 0.107676 | 0.107676 | 0.107676 | 0.107676 |
| 5 | 0.384327 | 0.384327 | 0.384327 | 0.384327 |
| 6 | 0.946479 | 0.946479 | 0.946479 | 0.946479 |
| 7 | 0.202625 | 0.202625 | 0.202625 | 0.202625 |
| 8 | 0.646272 | 0.646273 | 0.646273 | 0.646273 |
| 9 | 0.914417 | 0.914416 | 0.914416 | 0.914416 |
| 10 | 0.313033 | 0.313037 | 0.313037 | 0.313037 |
| 11 | 0.860174 | 0.860179 | 0.860179 | 0.860179 |
| 12 | 0.481098 | 0.481084 | 0.481084 | 0.481084 |
| 13 | 0.998570 | 0.998569 | 0.998569 | 0.998569 |
| 14 | 0.005708 | 0.005716 | 0.005716 | 0.005716 |
| 15 | 0.022702 | 0.022735 | 0.022735 | 0.022735 |
| 16 | 0.088747 | 0.088875 | 0.088875 | 0.088875 |
| 17 | 0.323485 | 0.323907 | 0.323907 | 0.323907 |
| 18 | 0.875370 | 0.875965 | 0.875965 | 0.875965 |
| 19 | 0.436386 | 0.434601 | 0.434601 | 0.434601 |
| 20 | 0.983813 | 0.982892 | 0.982892 | 0.982892 |
| 25 | 0.652836 | 0.757549 | 0.757549 | 0.757549 |
| 30 | 0.934926 | 0.481445 | 0.481445 | 0.481445 |
| 35 | 0.848152 | 0.313159 | 0.313159 | 0.313159 |
| 39 | 0.014638 | 0.006038 | 0.006038 | 0.006038 |
| 40 | 0.057695 | 0.024007 | 0.024009 | 0.024009 |
| 45 | 0.991612 | 0.930952 | 0.930881 | 0.930881 |
| 50 | 0.042173 | 0.629401 | 0.625028 | 0.625028 |
| 55 | 0.108415 | 0.749775 | 0.615752 | 0.615752 |
| 60 | 0.934518 | 0.757153 | 0.315445 | 0.315445 |
| 63 | 0.770667 | 0.690457 | 0.996571 | 0.996571 |

**Table 3.** Time reported by the Haskell Glasgow compiler. All operations were calculated at precision 1*E*-11

| Operation | Time reported in seconds | |
|---|---|---|
| | Plume | LRT |
| $\sum_{i=1}^{20} \frac{1}{2_i+1}$ | 1.599987 | 0.328326 |
| $\sum_{i=1}^{50} \frac{i}{3_i+1}$ | 1.808238 | 0.754179 |
| $\sum_{i=1}^{20} \frac{1}{2_i+1}$ | 0.664056 | 11.351002 |
| $\sum_{i=1}^{50} \frac{i}{2_i+1}$ | 1.362235 | 27.573404 |
| $j = \frac{1}{3}$ *for* $i = 2$ *to* $20$ $j = j$ *div* $\frac{i}{2_i+1}$ | 1583.99 | 1786.81 |
| $sin(\frac{1}{3} + cos(\frac{7}{9})) \,/\, cos(\frac{1}{3} + sin(\frac{7}{11}))$ | 0.716 | 12.54 |
| $e^{(tan(3/11)) - tan(2/13))}$ | 1.048 | 9.27 |
| $\pi * arctan(1/3) + (cos(2/3) * tan(13/15)$ | 0.556 | 23.97 |
| $sin(3/11 + e^{(1/3)}) * sin(4/13 - e^{(2/3)})$ | 0.564 | 27.109 |

exact real number computation. Instead, we wanted to show that it is possible to transit from the basic LRT theory to actual practice in a smooth way.

### *5.2. Comparing with the three digits representation*

The three digit implementation used in our comparison was developed by Plume (1998).[4] We can say, however, that our implementation is better at performing additions and subtractions but is less good at multiplications and divisions. These results affect in general the performance of trigonometric functions and other operations as can be seen in table 3. However, our implementation is based on a formal specification in which our programs are shown to be correct. Therefore, we believe it is easier to develop new algorithms for our implementation than build a theory for the three digit operational semantics representation.

## 6. Prospective Analysis

Several rounding off errors have occurred during the last years due to floating point arithmetic. These errors have caused disasters like the following.

During the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soldiers. It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors.

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after lift-off. The rocket was on its first voyage, after a decade of development costing $7 billion. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,768, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed.

The Germanas experienced a shattering computer error during a election (5 April, 1992). The elections to the parliament for the state of Schleswig-Holstein were affected. After midnight (and after the election results were published) someone discovered that the Greens actually only had 4,97% of the vote. The program that prints out the percentages only uses one place after the decimal, and had *rounded the count up* to 5%! This software had been used for *years*, and no one had thought to turn off the rounding at this very critical (and IMHO very undemocratic) region.

4. We will not discuss Plume's implementation in this paper; instead we refer the interested reader to Plume (1998).

Although it can be argued that the previous examples were in the nineteen`s, the solutions taken to prevent them in the coming years have not been soundly and correctly verified. So, as soon as the precisions for the calculations reach a maximum fixed value, the same errors will occur. If the implementation proposed in this paper ends in a faster library for exact real number computation, *then* the development of accurate software will be made and so none of the previous errors will occurs. However, this is still the first step towards this goal.

## Conclusion

We have described an implementation of LRT in the Glasgow Haskell compiler and a basic calculator using Plume's algorithms. Although these algorithms have the same range of convergence, our implementation is in most of the cases slower as table 3 shows. We consider that the growth of the rational intervals during a calculation decreases the efficiency of our implementation due to the number of operations required. Also the multiplication algorithm contributes to the efficiency. We believe that a new algorithm for multiplication, the use of dyadic rational (as used by other implementation (GNU; Bauer and Kavkler, 2008; Hanrot, Lefévre and Zimmermann; Lambov B.; Muller N.) and a faster library to compute operation with the rational numbers will improve the efficiency of our implementation.

Being the language representation independent, other algorithms proposed for first order computable functions can be programmed in LRT. A further work is to translate the best of those algorithms to LRT. In this paper we do not present any efficiency results of the implementation, which similar to others, is still the main gap between what is needed and what has been achieved in exact real number computation.

## Bibliografía


Abramsky, S. and A. Jung (1994). "Domain Theory", in S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, (ed.). *Handbook of Logic in Computer Science*. Volume 3, 1: 168. Clarendon Press.

Bauer, A. and I. Kavkler (2008). "Implementing Real Numbers with Rz", in *Proceedings of the Fourth International. Conference on Computability and Complexity in Analysis*. ENTCS.

Boehm, H. J. and R. Cartwright (1990). "Exact Real Arithmetic: Formulating Real Numbers as Functions", in Turner. D. (ed.). *Research Topics in Functional Programming*. Addison-Wesley.

Brattka, V. (1996). Recursive Characterization of Computable Real-Valued Functions and Relations. *Theoretical Computer Science*, Vol. 162: 45-77.

Devaney R. L. *An Introduction to Chaotical Dynamical Systems*. Addison-Wesley, California, 2do edition, 1989.

Escardó, M. H. (1996). "PCF Extended with Real Numbers", *Theoretical Computer Science*. Vol. 162, Núm. 1: 79-115, August.

Hanrot, P. P. G. V. Lefévre and P. Zimmermann (2007). *The MPFR library*. INRIA. <http://mpfr.org> (julio de 2011)

GNU (2000). *Multiple precision arithmetic library*. <http://gmplib.org> (julio de 2011).

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

Lambov, B. (2001). *The Reallib Project*. BRICS, University of Aarhus. <http://brics.dk/~barnie/RealLib> ( julio de 2011).

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

Marcial-Romero, J. R. and M. H. Escardó (2004). "Semantics of a Sequential Language for Exact Real-Number Computation", in Ganzinger, H. (ed.). *Proceedings of the Nineteenth Annual IEEE Symp. on Logic in Computer Science*. LICS IEEE Computer Society Press.

Marcial-Romero, J. R. and M. H. Escardó (2007). "Semantics of a Sequential Language for Exact Real-Number Computation", *Theoretical Computer Science*, Vol. 379, Núm. 1-2: 120-141.

Marcial-Romero, J. R. and A. Moshier (2008a). "Sequential Real Number Computation and Recursive Relations", in *Proceedings of the Fourth International Conference on Computability and Complexity in Analysis*, CCA. ENTCS.

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

Muller, N. (1996). *iRRAM - Exact Arithmetic in C++. Universit at Trier*. <http://www.informatik.unitrier.de/iRRAM> (julio 2011).

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

Plume, D. (1998). *A Calculator for Exact Real Number Computation*. 4th Year Project Report, Department of Computer Science and Artificial Intelligence, University of Edinburgh.

Potts, P. J.; A. Edalat and M. Escardó (1997). *Semantics of exact real arithmetic*. Proceedings of the Twelveth Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society Press.

Weihrauch, K. (2000). *Computable Analysis*. Springer.