Exact Real Calculator for Everyone

Weng Kin Ho[*]
`wengkin.ho@nie.edu.sg`
National Institute of Education
Nanyang Technological University
1 Nanyang Walk, Singapore 637616

August 7, 2013

Abstract

Despite its simplicity and versatility, the well-known Floating Point System (FPS) has a serious shortcoming: the finite nature of a computer makes rounding-off inevitable. Because of this, FPS can sometimes lead to serious computational errors, i.e., a small round-off error due to truncation can cause a large deviation in the output in iterations within chaotic systems. This paper bridges the gap between theory and practice of Exact Real Arithmetic (ERA), and reports on the design and implementation of a user-friendly scientific calculator `ERCE` using HASKELL, capable of ERA. With a functional-programming slant, we use `ERCE` as a channel for the technology of ERA to reach out to a wider community: even a school student can use it.

1 Introduction

Current computers handle real number computations using the well-known floating point system (FPS, for short) – the computer realization of $x = A \times 10^n$, where A stands for the significant part of the numeral x. Within A, the radix point is allowed to 'float', and hence the name. Recognized for its simplicity and versatility, FPS even has its own IEEE standard (IEEE 754 [19]). Instead of base 10, IEEE 754 single-precision floating point is encoded in 32 bits using 1 bit for the sign s, 8 bits for the exponent e and 23 bits for the normalised mantissa m without the leading 1 so that a 'real' number can be written in the form $(-1)^s \cdot 2^{e-127} \cdot (1.m)$ [27, p.1].

Despite its versatility, the FPS has a serious shortcoming: numbers with too many (or infinitely) significant digits can only be represented by nearby rational numbers, and this produces round-off errors. Though such inaccuracies are generally tolerable, there are inevitable exceptions. For example, functions such as the logistic map which are sensitive to initial input can yield behemoth deviations even when seemingly negligible errors occur in the input [24].

[*]Supported by NTU AcRF Project No. 10HWK

For this, the FPS has been blamed for numerous disasters of varying nature and degree of severity [20, 33].

In general, any positive-integral-base real number representation suffers a realizability problem: even simple functions such as scaling, e.g., $t(x) = 9x$, cannot be programmed in such systems. Counterintuitive it may seem, this fact will be explained in the next section. First pointed out by Brouwer [7], this realizability problem can be circumvented by a number of approaches, e.g., alternative real numbers representations (e.g., rational-interval representation [21, 23, 28, 38], computable Cauchy sequence of rational numbers with a fixed rate of convergence [31, 4], signed-digits representation [1, 5, 36, 13], continued fraction representation which admits negative integers [34], linear fractional transformations [35, 9, 10]). For a comparison of these, see [16, p.13–15].

Exact Real Arithmetic (ERA) refers to the science of computing real numbers up to arbitrary precision, and is one of the ways to overcome the above shortfall of FPS. Typically, ERA makes use of ways for representing real numbers (e.g., Dedekind cuts, continued fractions, etc.) and algorithms for realizing real number computations such that the infinite nature of real numbers is kept 'intact'. Implementations of ERA often exploit recursive data structures and associated computations. Thus, the multi-faceted nature of ERA lends itself to an exciting meeting place for computability theory [32, 30, 37], topology in the form of domain theory [16, 11], integration theory [8] and programming theory [12].

By now, theories fundamental to ERA have been well-studied and understood. However, equally important are the practical issues of ERA: how can it be implemented in computers? Major steps to answer this have already been taken by computer scientists. Using functional programs such as HASKELL, D. Plume built a basic calculator that can perform real exact arithmetic operations using signed-digit representation and dyadic representation of real numbers [26]. Continuing this enterprise, A. Scriven further implemented elementary functions of high school calculus (e.g., algebraic, trigonometric, exponential, logarithmic functions) based on the signed-digit representation of Plume. Based on theoretic developments in Simpson [29] and Escardó [14, 15], ERA programs have been created for the Riemann integral, and the supremum function on closed bounded intervals. On another track, A. Bauer and P. Taylor used Dedekind cuts and Abstract Stone Duality to perform efficient ERA in `OCaml` [3]. Recently, more state-of-the-art implementations for ERA emerged such as `iRRAM` ([25]) and `RealLib` ([22]), `xrc` (Exact Reals in C) ([6]), `HERA` (Haskell Exact Real Arithmetic), and RZ ([2]).

The aforementioned systems have a common drawback: users are expected to have background knowledge on functional programming and exact real arithmetic. Most of these systems are not equipped with a user-friendly interface. Though slowly gaining impetus in its development and application, functional programming has yet to attain wide usage from the programming community owing to its relatively abstract mathematical overhead. Founded on the theoretic ideas and algorithms due to the abovementioned researchers, this paper aims to *bridge the existing gap between theory and practice* in ERA. We present the design and implementation of a user-friendly scientific calculator `ERCE` programmed in HASKELL for ERA. In particular, we show how ERA calculators developed using functional programming languages can be made available to users who has no prior knowledge in programming. Congruous to the ATCM theme of harnessing technology for mathematics (and vice versa), we demonstrate how ERA can be easily accessible to a wider community. The features of `ERCE` are: Users (i) need only to know a hand-held scientific calculator is used, (ii) can input one-variable functions

$f : \mathbb{R} \longrightarrow \mathbb{R}$ as primitives, and perform calculations with them, and (iii) can perform ERA.

2 Preliminaries

2.1 Finite character of calculating machines

Real numbers are infinite objects, and so ERA involves computing with infinite objects. Thinking of a machine that performs ERA as a black box that feeds on a real number, say, coded as an infinite stream of a finite number of symbols (e.g., $., 0, 1, \ldots, 9$, for convenience) to print some real number (say, using a stream of the same symbols). The machine reads some finite number of digits from the input, makes some calculations (and perhaps read some more digits) and then prints one (or more) digits for the output. To print the next digit, the preceding process is repeated. Note that every *finite* part of the output depends only on a *finite* part of the input. In other words, a working program can only do finite *look-aheads*. This is the very nature of *any* calculating machine – its finite character; a machine which must read an infinity of digits in the input before outputting anything basically computes nothing.

In light of the finiteness of machines, all real number representation using positive integral base (e.g., FPS) have a major problem:

Proposition 1 *The scaling function* $(x \mapsto 9x) : \mathbb{R} \longrightarrow \mathbb{R}$ *is not realizable.*

Proof. For illustration sake, it suffices to consider base 10. Suppose not, i.e., a machine that realizes $(x \mapsto 9x)$ does exist. Consider what this machine outputs when it is applied to the potential input $\frac{1}{9} = 0.111\ldots := 0.\dot{1}$. There are two ways to output 1: *either* print $1.\dot{0}$ (i.e., '1.' followed by an infinite stream of 0's), *or* $0.\dot{9}$ (i.e., '0.' followed by an infinite stream of 1's). Let's take the first option. For this, it must print '1' as the first digit. A simple reasoning involving inequality quickly reveals that unless the digit '2' appears somewhere after a finite trail of 1's in the input, one can never be sure when to output '1' as the first digit of the output. In other words, it is never enough to scan a finite number of digits before deciding to print '1' as the first digit in the output; if it were so, say, after reading N digits of the input, then it would be wrong to print '1' as the first digit in the output for the input stream $\underbrace{0.111\ldots1}_{N}0\ldots$ because scaling it by 9 times gives a number strictly smaller than 1. A similar argument applies to the second option. In short, this machine cannot output anything. It is easy to see that the above argument can be modified for any positive integral base. ■

The finite character of the machine somewhat prevents it from realizing the scaling function using the usual positive integral base system, and this defect has long been identified, e.g., in [16, p.3].

2.2 Signed bit representation

One method to overcome the aforementioned problem is to use a different representation for real numbers, namely, signed-bit streams. Observe that any $a \in [-1, 1]$ can be represented as $[\![a]\!] := \sum_{i=0}^{\infty} a_i 2^{-i-1}$, where $a_i \in \mathbf{3} := \{\bar{1}, 0, 1\}$. Here $\bar{1}$ denotes the integer -1. By the Heine-Borel Theorem, a real number can be given a sequence of increasingly accurate approximations of its location, i.e., a real number a with initial segment $a_0 a_1 \ldots a_{n-1}$ must lie in the closed interval $\left[\sum_{k=0}^{n} a_k 2^{-k-1} - 2^{-n}, \sum_{k=0}^{n} a_k 2^{-k-1} + 2^{-n}\right]$. Each finite initial segment, known as a *partial real number*, can be identified with its associated closed interval. Ordering the set $\mathcal{I}$ of partial numbers using reverse inclusion one obtains what is called the *interval domain*; elements of $[-1, 1]$ can be seen to be maximal elements of $\mathcal{I}$ (see Figure 2.2). Pictorially, a real number such as $\frac{1}{9}$ can be realized by the path $\dot{0}0011\dot{1}$. Since for every finite string α and infinite string β over $\mathbf{3}$, $\alpha 01\beta$ and $\alpha 1\bar{1}\beta$ represent the same number (likewise for $\alpha\bar{1}1\beta$ and $\alpha 0\bar{1}\beta$), signed-bit representations are far from unique. For more on infinite signed-digit numerals in real exact arithmetic, see [13]. For an arbitrary real number r, one can express it in the form

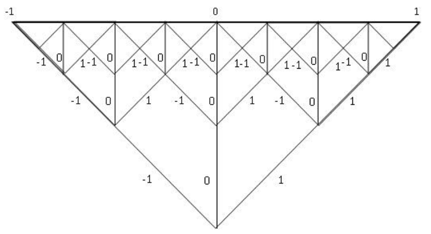

Figure 1: Interval domain $\mathcal{I}$

$(m, e) := m \times 2^e$, where $m \in \mathcal{I}$ is called the *mantissa* and $e \in \mathbb{Z}$ the *exponent.*

To see that the signed-bit representation solves the infinite look-ahead problem, we re-look at the example of realizing the function $(x \mapsto 9x)$. Suppose the machine expects an input stream of $0.\dot{0}0011\dot{1}$ that represents $\frac{1}{9}$. Then it is enough to inspect the first six bits 0.000111 before safely printing the first bit '1' for the mantissa m and '-1' for the exponent e in the output since if the input turns out to be $000111\dot{1} = \frac{1}{8}$, then the output will be $1.000\dot{1} = \frac{9}{8}$; and if the input turns out to be $000111\dot{\bar{1}} = \frac{3}{32}$, then the output will be $1.\bar{1}1010\dot{1} = \frac{27}{32}$.

2.3 Recursive paradigm

Because signed-bit streams are chosen as the medium of representation for reals, the vehicular programming language chosen to handle ERA should be equipped with facilities to handle infinite streams. Streams are a particular instance of *list.* Let σ be a data type (i.e., set of data with same attribute, e.g., integers, booleans, etc.). The list type derived from σ is the data type whose elements are streams over σ. More precisely, s is a datum of type $[\sigma]$ (denoted by $s :: [\sigma]$) if it is either an empty list $[\,]$ or of the form $s_0 s_1 s_2 \ldots$ where each $s_i :: \sigma$. Because each data type σ admits non-terminating elements, labelled collectively by $\perp$ – these are programs entering into infinite looping and running forever. Lists of the form $s_0 s_1 s_2 \ldots s_n \dot{\perp}$ that consist of a finite length of non-terminating elements s_i's followed by an infinite stream of non-terminations can be regarded as *finite* lists. List types are an instance of a more general type-construct called *recursive types.* The list type $[\sigma]$ can be defined in terms of itself as follows: $[\sigma] \cong \mathtt{1} + (\sigma \times [\sigma])$, i.e., a list l can either be an empty list $[\,]$ (i.e., the list that contains nothing and hence the void data type $\mathtt{1}$) or (hence the $+$ sign) $(x : xs)$, where x is of type σ (called the *head*) and (hence the $\times$ sign) $xs :: [\sigma]$ is another list (called the *tail*). A function f on lists can, too, be defined recursively: to execute f on a list l tell it (i) what to do if l is an empty list, and (ii) what operations to perform on x (its head) if l is non-empty before passing the execution of f on the tail xs. For rigorous reasoning principles associated to general recursive types, see [18, 17].

In summary, our programming mindset is a *recursive* paradigm for both data *structure* and *manipulation*. Because of this, we choose the sequential functional programming language HASKELL. Based on our discussion in the preceding section, the exact real calculator we build here relies on the data type `I` for $\mathcal{I} = [-1, 1]$ and `ME` for the mantissa-exponent representation for $\mathbb{R}$, which are defined in HASKELL as follows:

```
type SD    = Int          -- Signed digits. {-1,0,1}
type I     = [SD]
type ME    = (I,Int)      -- Mantissa exponent representation.
```

Notice that the command `type` creates data types built inductively using type constructors from existing ground types such as `Int` (single precision integers), `Integer` (double precision integers), `Bool` (booleans) and `Float` (floating point numbers, which we do not use). Apart from the void type `1`, binary product `(A,B)` and the list constructor `[A]`, one constructor central to functional languages is that of function space `A -> B`. A program `p::A -> B` expects an input `a:: A` and produces an output `p(a) :: B`. Because types are defined inductively via the ground types and type constructors, there is a great deal of flexibility in manufacturing data of sophisticated types, such as higher-order types `Int -> Int -> Int`. The list type nature of `I` (and `ME`) allows coding of real number constants; e.g., 0 and 1 can be coded as

```
i_ZERO, i_ONE :: I
i_ZERO = repeat 0
i_ONE  = repeat 1
```

where the in-built HASKELL function `repeat` is pre-defined recursively by

```
repeat :: a -> [a]
repeat x = (x: repeat x)
```

Programs (such as `repeat` above) written in HASKELL are typed, i.e., the data type of the program has to be first declared before the explicit definition is given.

3 Calculator design and implementation

This section is devoted to the design and implementation of the exact real calculator `ERCE` which is available for public access at `http://math.nie.edu.sg/wkho/ERCE`. The calculator `ERCE` constitutes of four components: (1) a modularized set of *functional programs* that realize real number functions and constants using `ME` types for real numbers, (2) a *parser* for mathematical (and function) expressions, (3) an *evaluator* that evaluates parsed expressions into real numbers to decimal form (up to user-defined precision) and (3) a *user interface* that serves to be the channel of interaction between the user and the calculator.

3.1 Functional programs

3.1.1 Arithmetic functions

To give the reader an idea of how arithmetic in real numbers is performed using signed-bit streams, we demonstrate how the addition operation is realized. The program makes use of the midpoint program `mid_I_I` which we construct in stages. Let $a, b \in [-1, 1]$ be two real numbers whose representation in `I` are the signed-bit streams $\mathtt{a} := a_0 a_1 \ldots$ and $\mathtt{b} := b_0 b_1 \ldots$. Since calculating $a + b$ involves evaluating $\sum_{k=0}^{\infty}(a_i + b_i) \cdot 2^{-i-1}$ and, in addition, each $a_i + b_i \in \mathbf{5} := \{-2, -1, 0, 1, 2\}$, there is a need to create a new data type `I2` whose elements are streams over the set $\mathbf{5}$. The function that `add2` realizes essentially adds the streams $a_0 a_1 \ldots$ and $b_0 b_1 \ldots$ elementwise, i.e., it produces $a_0 + b_0, a_1 + b_1, \ldots$. Using the prefix notation of `(+)`, this sum may be denoted by $(+)a_0\ b_0, (+)a_1\ b_1, \ldots$. The in-built program `zipWith` applies the binary operation `(+)` to the pair of corresponding elements in the input streams; taking the imagery of the two streams as the two sides of a zip and their elements as the teeth, elementwise operation to produce a new stream is then seen as a *zipping* action. Hence the program `add2` can be implemented by the codes below:

```
add2 :: I -> I -> I2
add2 = zipWith (+)
```

Notice that `add2` is not a binary operation on `I` since closure is not achieved. The best we can get is the midpoint operation $\oplus$; given $a, b \in [-1, 1]$, $a \oplus b := \frac{a+b}{2}$. Given that we already have the sum of two real numbers a and b as $\sum_{k=0}^{\infty}(a_i + b_i)2^{-i-1}$, to obtain their midpoint one only needs to divide this sum by 2. Because for each i the element $a_i + b_i \in \{-2, -1, 0, 1, 2\}$ it may not be the case that one can divide $a_i + b_i$ by 2 directly to obtain an element in $\{-1, 0, 1\}$. To circumvent this problem, one must rely on the program `div_I_2` which divides a real number (represented by the $\mathbf{5}$-digits streams in base 2) by 2. Like many programs defined in HASKELL, `div_I_2` can be defined by *pattern matching*. The first two cases of the matching appear below:

```
div_I_2 :: I2 -> I
div_I_2 (-2:      x) = -1 : div_I_2      x
div_I_2 (-1:(-2):x) = -1 : div_I_2 ( 0:x)
```

To see that the way `div__2` is defined is forced upon by the meaning of dividing by 2, we examine the action of `div_I_2` on the second case, i.e., $\mathtt{s} := (-1 : (-2) : \mathtt{x})$. Denoting by `f` the above program `div_I_2` and suppose $[\![\mathtt{f}]\!] = \frac{(-)}{2}$, then

$$\begin{aligned}
[\![\mathtt{f(s)}]\!] &= \frac{([\![\mathtt{s}]\!])}{2} \\
&= \frac{([\![(-1{:}(-2){:}\mathtt{x})]\!])}{2} \\
&= \frac{(-1{\cdot}2^{-1}+(-2){\cdot}2^{-2}+\frac{1}{2^2}[\![\mathtt{x}]\!])}{2} \\
&= \frac{(-2{\cdot}2^{-1}+\frac{1}{2^2}[\![\mathtt{x}]\!])}{2} \\
&= -1 \cdot 2^{-1} + \frac{1}{2}\frac{(0{\cdot}2^{-1}+\frac{1}{2}[\![\mathtt{x}]\!])}{2} \\
&= [\![(-1 : \mathtt{f}(0 : \mathtt{x}))]\!]
\end{aligned}$$

Composing `add2` and `div_I_2` with the functional composition operator `.`, we derive the midpoint operator `mid_I_I`:

```
mid_I_I :: I -> I -> I
mid_I_I = div_I_2.add2
```

Suppose $x = m_0 \times 2^{e_0}$ and $y = m_1 \times 2^{e_1}$. Without loss of generality, we may assume that $e_0 \leq e_1$ so that $e := \max\{e_0, e_1\} = e_1$. we have:

$$\begin{array}{rcl} x + y & = & m_0 \times 2^{e_0} + m_1 \times 2^{e_1} \\ & = & \frac{(m_0 \times 2^{e_0 - e} + m_1 \times 2^{e_1 - e})}{2} \times 2^{e+1} \\ & = & \frac{(m_0 \times 2^{-(e-e_0)}) + m_1}{2} \times 2^{e+1} \end{array}$$

Given `(m2,e2)` in `I` with `e2` a non-positive integer, we require an auxiliary function `f` that 'shifts' the radix point '.' backwards by `e2` number of places, or equivalently prefix the mantissa `m2` by `abs(e2)` number of 0's. Thus `f` converts the mantissa-exponent $m_0 \times 2^{-(e-e_0)}$ into a signed-bit stream in `I`. This is followed by an application of the midpoint operator `mid_I_I` to the pair of signed-bit streams `f (m0,e0-e)` and `f (m1,e1-e)` which calculates $\frac{(m_0 \times 2^{e_0 - e} + m_1 \times 2^{e_1 - e})}{2}$ as the mantissa of the output. The exponent of the output is accounted by `e+1`. All these justify the following script for `add_ME_ME`:

```
add_ME_ME :: ME -> ME -> ME
add_ME_ME (m0,e0) (m1,e1) = ( mid_I_I (f (m0,e0-e)) (f (m1,e1-e)) , e + 1 )
 where e          = max e0 e1;
       f (m2,e2) = (take (abs (e2)) i_ZERO) ++ m2;
```

Like addition, the rest of the arithmetic operations (`sub_ME_ME`, `mul_ME_ME`, `div_ME_ME`) manipulate signed-digit streams to achieve the desired representation that corresponds to the final intended output. Also, raising a real number by a non-negative integral exponent can be realized by `me_power_INT` making use of `mul_ME_ME`:

```
me_power_INT :: ME -> INT -> ME
me_power_INT (m,e) 0 = me_ONE
me_power_INT (m,e) n = mul_ME_ME (me_power_INT (m,e) (n-1)) (m,e)
```

3.1.2 Elementary functions

The elementary functions our exact real calculator is able to handle are restricted to those expressible in some convergent power series. Our work is based on Adam Scriven's work on realizing infinite series of the form $\sum_{k=0}^{\infty} \frac{x_k}{2^{k+1}}$, where $(x_k)_{k=1}^{\infty}$ is a sequence of real numbers in $[-1, 1]$. The main idea that justifies how our programs are to be written is this seemingly innocent-looking identity: $x \equiv \frac{4x}{4}$.

Suppose $x \in [-1, 1]$. Then $4x \in [-4, 4]$, and thus, we need to create a new data type `I4` for the signed-digit ($\{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$) representation of a number in $[-4, 4]$.

```
type I4     = [SD4]         -- Signed-4 digit streams.
```

Working out an infinite series requires nothing but adding all the terms in an infinite sequence. So, in our setting of infinite streams, we need to work with list of streams, e.g., `[I]`. The program that multiplies a series $\sum_{k=0}^{\infty} \frac{x_k}{2^{k+1}}$ by 4 can be realized as follows:

```
mul_SumOfIListBy4 :: [I] -> I4
mul_SumOfIListBy4 ((a:b:x):(c:y):s)
= (2*a + b + c): mul_SumOfIListBy4 ((mid_I_I x y):s)
```

To see how this program works, let $(x_0 : x_1 : x')$ be a stream of real numbers in $[-1,1]$, represented by streams `(x0:x1:x')::[I]`. Note that `x0, x1` :: `I` while `x'` :: `[I]`. Further let x_0 and x_1 be represented by the signed-bit streams `(a:b:x)` and `(c:y)` respectively. We require the denotation of the program `mul_SumOfIListBy4` be multiplication by 4 so that

$$[\![\texttt{mul\_SumOfIListBy4}]\!] = 4\left(\frac{x_0}{2}+\frac{x_1}{4}+\frac{x'}{4}\right) = 4\cdot\sum_{k=0}^{\infty}\frac{x_k}{2^{k+1}}.$$

Thus, we have:

$$\begin{array}{rcl}
[\![\texttt{mul\_SumOfIListBy4}]\!] & = & 4\left(\frac{x_0}{2}+\frac{x_1}{4}+\frac{x'}{4}\right)\\
& = & 2x_0 + x_1 + x'\\
& = & 2[\![(\mathsf{a} : \mathsf{b} : \mathsf{x})]\!] + [\![(\mathsf{c} : \mathsf{y})]\!] + x'\\
& = & 2\left(\frac{a}{2}+\frac{b}{4}+\frac{x}{4}\right)+\frac{c}{2}+\frac{y}{2}+x'\\
& = & (2a+b+c)\cdot 2^{-1} + x\cdot 2^{-1} + y\cdot 2^{-1} + x'\\
& = & (2a+b+c)\cdot 2^{-1} + (x\oplus y) + x'\\
& = & (2a+b+c)\cdot 2^{-1} + 2^{-1}\cdot 4\left(\frac{x\oplus y}{2}+\frac{x'}{2}\right)\\
& = & (2a+b+c)\oplus [\![\texttt{mul\_SumOfIListBy4}(\texttt{mid\_I\_I x y} : \texttt{x}')]\!]\\
& = & [\![(2*\mathsf{a}+\mathsf{b}+\mathsf{c}) : \texttt{mul\_SumOfIListBy4}(\texttt{mid\_I\_I x y} : \texttt{x}')]\!]
\end{array}$$

Mimicking the construction of `div_I_2`, it is easy to write the program `div_I_4 ::  I4 -> I` for division by 4 and we omit its code here. Thus, $\sum_{k=0}^{\infty}\frac{x_k}{2^{k+1}}$ can be realized by composing the above two programs, i.e.,

```
sum_ILIST :: [I] -> I
sum_ILIST = div_I_4.mul_SumOfIListBy4
```

An elementary function which is expressible in power series (whose summands are all bounded by unity) can be realized by the preceding consideration. We use $\exp(x) = e^x$ as an example by proceeding in two stages: (1) realize its restriction on $[-1,1]$, and (2) then extend it to $\mathbb{R}$. First consider $\frac{1}{2}\exp(\frac{x}{2})$ whose series expansion is $\sum_{k=0}^{\infty}\frac{1}{2^{k+1}}\cdot\frac{x^k}{k!}$. To exploit the `sum_ILIST` program, we supply the sequence $a_k = \frac{x^k}{k!}$, $k = 0,1,\ldots$. Note that the sequence $(a_k)_{k=0}^{\infty}$ may be defined recursively as follows: $a_0 = 1$, $a_{k+1} = \frac{x}{k+1}$, $(k = 0,1,\ldots)$. For code uniformity, we program the sequence $(a_k)_{k=0}^{\infty}$ as follows:

```
pgmtofind_ALLTERMSofmy_e :: I -> I -> Q -> [I]
pgmtofind_ALLTERMSofmy_e y x (m,n)
= y : pgmtofind_ALLTERMSofmy_e (mul_I_PPF (mul_I_I y x) (m,n)) x (m,n+1)

find_ALLTERMSofmy_e :: I -> [I]
find_ALLTERMSofmy_e x = pgmtofind_ALLTERMSofmy_e i_ONE x (1,1)
```

It then follows that the function $\frac{1}{2}e^{\frac{x}{2}}$ is realized by the program below:

```
my_e :: I -> I
my_e x = sum_ILIST (find_ALLTERMSofmy_e x)
```

Since $e^x = 4\cdot\left(\frac{1}{2}e^{\frac{x}{2}}\right)^2$, the exponential function $\exp(x)$ may be realized by the following program:

```
e_power_I :: I -> ME
e_power_I x = mul_ME_ME me_FOUR (mul_I_I (my_e x) (my_e x),0)
```

Secondly, we extend the above program to cope with the entire real line. Given a real number $x = m \times 2^{e_1}$ in mantissa-exponent form, where m and e are realized by `m` and `e1`. Since $e^x = e^{m\times 2^{e_1}} = (e^m)^{2^{e_1}}$, the desired program for calculating $\exp : \mathbb{R} \longrightarrow \mathbb{R}$ is given by:

```
e_power_ME :: ME -> ME
e_power_ME (m,e1) = me_power_INT (e_power_I m) (2^e1)
```

3.2 Parser function

The parser is executed through a functional program `parse :: String -> Expression` which receives an input of a mathematical expressions and produces a tree that is typed as `Expression`.

Example 2 *We run the program* `parse` *on the input* `"1+2/sin(PI/3)*7.13-0"` *below:*

```
*Parse> parse "1+2/sin(PI/3)*7.13-0"
Op Add (Value "1")
       (Op Add (Op Mul (Op Div (Value "2")
                       (Fun Sin (Op Div (Value "PI")
                                        (Value "3"))))
                       (Value "7.13"))
       (Op Mul (Value "-1") (Value "0")))
```

Instead of giving all the explicit script for `parse`, it suffices to make a qualitative description of it. The program `parse` anticipates all usual mathematical expression, except that multiplication `*` must be spelt out explicitly. For instance, the parser does not accept the string `"3(1+2)"`. This restriction must be in place so that function application is possible in our calculator.

Values and variables. Using the program `chkvalue`, the parser detects if a mathematical expression is a numerical constant, e.g., `PI` or a numerical value in decimal form, i.e., either an integer such as `-4` or a radix form `2.34`. It also admits the variable `X`. Numerical constants and decimal values are parsed as a leaf in the expression tree, i.e., marked by a tag `Value` that prefixes this numeral string. The variable `X` is parsed as the leaf marked by a tag `Var` that prefixes the character `X`. Numerical constants and decimal values can be thought of as ground values to be evaluated as a real number, while variables can be thought of as a 'hole' waiting for a value or a mathematical expression to be filled in. The tag `Var` flags the formation of a function of type `ME -> ME`, e.g., a mathematical expression `4*X*(1-X))` will be parsed as the logistic function $L(x) = 4x(1-x)$.

Example 3 *We run* `parse` *respectively on* `-2.4`*,* `PI` *and* `X` *below:*

```
*Parse> parse "-2.4"
Value "-2.4"
*Parse> parse "PI"
Value "PI"
*Parse> parse "X"
Var "X"
```

Binary operations. An instance of a binary operation is parsed as an expression tree tagged with the prefix `Op`, followed by a label for this operation, together with its two arguments.

Example 4 `*Parse> parse "3/7"`
```
Op Div (Value "3") (Value "7")
```

Also, the parser is sophisticated enough to take care of the order of arithmetic operations; for instance:

```
*Parse> parse "1*(2-3*4/5+6)"
Op Mul (Value "1")
       (Op Add (Value "2")
               (Op Add (Op Div (Op Mul (Op Mul (Value "-1") (Value "3"))
                                       (Value "4"))
                               (Value "5"))
               (Value "6")))
```

Notice that a substraction $x - y$ is always rewritten as $x + (-1) \cdot y$ before parsing. This is made possible by applying the negtoadd function to the incoming string of mathematical expression which replaces the "-" by "`+(-1)*`"

Function application. An instance of a function application is raised when the user keys in a string of the form `(exp1)(exp2)`, where `exp1` is a function of a single variable `X` and `exp2` intended to be substituted into `X`. Making use of a program `posclosebr` which returns the position of the close parenthesis that corresponds to the open one at the head of the string, the program `testappl` below

```
testappl :: String -> Bool
testappl s = (head(s) == '(') &&
             (posclosebr(s) /= length(s)) && (s!!(posclosebr (s)) == '(')
```

detects any instances of function application by the presence of the sub-string "`)(`". This explains why we make the multiplication operation explicit to avoid potential ambiguity. More precisely, in our syntax, `(1+2)(3-4)` denotes the application of the constant function "`(1+2)`" (which contains no variable `X`) to the input `(3-4)` rather than the multiplication of `(1+2)` to `(3-4)`.

Example 5 *We run* `parse` *on* `(4*X*(1-X))(0.5)` *below:*

```
*Parse> parse "(4*X*(1-X))(0.5)"
App (Op Mul (Op Mul (Value "4") (Var "X"))
            (Op Add (Value "1") (Op Mul (Value "-1") (Var "X"))))
    (Value "0.5")
```

The tag `App` raises an instance of function application.

The parser also admits a finite iteration n of a function f on an input x_0, an instance of which takes the form `(exp1 @ exp2)(exp3)`. Here, `exp1` is a function f in `X`, `exp2` is a non-integer n (standing for the number of iterations) and `exp3` is the seed of the iteration x_0.

Example 6 *We run* `parse` *on* `((4*X*(1-X))@20)(0.5)` *below:*

```
*Parse> parse "((4*X*(1-X))@20)(0.5)"
App (Itn (Op Mul (Op Mul (Value "4")(Var "X"))
                 (Op Add (Value "1")(Op Mul (Value "-1") (Var "X"))))
          "20")
    (Value "0.5")
```

The tag `Itn` raises an instance of a finite iteration of a function.

Elementary functions. If the string is headed by an elementary function, then the name of the function is matched and the string is parsed into an expression tagged by `Fun` along with the corresponding function label.

Example 7 *We run* `parse` *on* `cos(Pi/3)` *as follows:*

```
*Parse> parse "cos(PI/3)"
Fun Cos (Op Div (Value "PI") (Value "3"))
```

3.3 Evaluator

The evaluator unit assigns a real number in decimal representation (that can be read intelligibly by a human) to the tree expression parsed out of the `parser` program. Because a tree expression can either be a function or not, the evaluator *evaluates* its input based on this dichotomy. If the input is a function, the evaluator interprets it into a function of type `ME -> ME`; otherwise, it interprets it into an element of `ME`. Below are some cases of pattern matching for the evaluator unit (which consists of the programs `evalme` and `evaltofn`):

```
-- Evaluating parsed trees in mantissa exponent form
evalme :: Expression -> ME
evalme (Value x)      = if x == "PI" then myPI else cvt_DECSTRINGtoME x
evalme (App e1 e2)    = (evaltofn e1)(evalme e2)
evalme (Op Add e1 e2) = add_ME_ME (evalme e1) (evalme e2)
evalme (Fun Exp x)    = e_power_ME (evalme x)

-- "Evaluating" an application to a function
evaltofn :: Expression -> ME -> ME
evaltofn (Var x) r = r
evaltofn (Value x) r = if x == "PI" then myPI else cvt_DECSTRINGtoME x
evaltofn (Itn e1 e2) r = (repeat_MEfunction (evaltofn e1) (read e2 :: Int)) r
```

Any mathematical expression is, by default, given a functional citizenship. In particular, any mathematical value or constant (which does not contain a variable `X`) is regarded as a constant function. This functional feature of our calculator stands in stark contrast to other existing scientific calculators, and is made possible because HASKELL, the language with which we build our exact real calculator, is a functional one. Finally, for human consumption, the signed-bit streams are then converted into readable form, i.e., decimal representation correct to a default of 20 decimal places.

3.4 User-interface

The module `Main` is the standard platform by which *actions-sequence* can be scripted. To link up the interface with the underlying programs written thus far, we first import the `Eval` and the `Parse` modules, and next the graphic user interface module `Graphics.UI.Gtk` (supported by WXHASKELL). Actions are of an abstract data type `IO a`, where `IO` is a fixed type constructor that flags an I/O action and the data type `a` is the type of data which this action returns. The action `main` records a sequence of actions to be performed, one after another, which appears after the syntax `do`.

After initialising GUI, the window frame fixes the dimension of a table, which forms the panel for the buttons. We have designed two display panels whose output are supplied by the variables `label1` and `label2`. The panel `label2` displays the mathematical expression entered as a string by the user, while `label1` displays the evaluation of the string keyed in as `label2` (up to the user-declared precision) after the key `=` is pressed.

There are two kinds of buttons: *symbolic* and *executive*. A symbolic button when clicked inserts a single mathematical symbol (except `=`) indicated by the button label (e.g. `button1`) at the end of the existing string of symbols already keyed in. An executive button when clicked executes a non-symbolic action. There are a few non-symbolic actions: (i) deleting the last symbol keyed in, (ii) clearing all panel display, and (iii) copying the last displayed answer from `label1` to `label2`, inserting it as the beginning part of a fresh string entry. When a symbolic button is clicked, the `buttonSwitch` function below is invoked:

```
buttonSwitch :: Button -> Label -> Label -> IO()
buttonSwitch b l z= do
  txt <- get b buttonLabel
  lb <- get l labelText
  lbs <- get z labelText
  if txt == "="
     then labelSetText l( show(eval (parse lbs)))
     else labelSetText z(lbs ++txt)
```

A screen-shot of the `ERCE` user-interface is shown in the figure below:

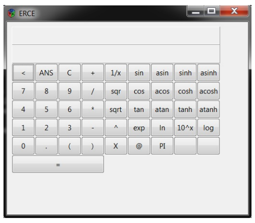

Figure 2: A screen-capture of the calculator user-interface

4 Sample application

In this section, we use `ERCE` to calculate the iterates of the logistic function $Q(x) = 4x(1-x)$ on $x_0 = 0.125$. Because FPS eventually admits an error of negligible magnitude after some small finite iterations, the chaotic nature of the above iterative system then forces upon us a large error in some subsequent iterate. We compare, below, the first 20 decimal places of the iterates obtained by `ERCE` and MATLAB (which relies on FPS) respectively. The underlined part of the MATLAB output indicates discrepancy of each iterate from the correct `ERCE` output.

n	`ERCE`	MATLAB
0	0.12500000000000000000	0.12500000000000000000
10	0.38367583854736609603	0.38367583854736$\underline{526000}$
20	0.55150781744159181178	0.55150781744$\underline{258315000}$
30	0.29059706649102177619	0.29059706$\underline{556439530000}$
40	0.94723756671816869896	0.94723$\underline{803391889350000}$
50	0.97984857115056995132	0.9$\underline{8014818012213323000}$
100	0.99971849434213872830	0.$\underline{78273071414107154000}$

Table 1: Orbits of the logistic map produced by ERCE and MALTAB

5 Concluding remarks

Our calculator, `ERCE`, is a first prototype of a common scientific calculator that performs ERA with an emphasis of functions being first class citizens. Our aim is to promote a wide usage of exact real arithmetic, e.g., even school students can use it. `ERCE` is far from perfect. Indeed speed is an area of concern. Also, we have not included complex numbers arithmetic and calculus-related algorithms in `ERCE`, e.g., for supremum function, Riemann integrals, etc. As the system develops, we hope to address these issues. Lastly, I wish to acknowledge my project team members: the co-PI's (K. C. Ang & D. Zhao), and the Research Assistants (M. Fadzli & S. C. Khoo).

References

[1] A. Avizienis. Binary-computable signed-digit arithmetic. In *AFIPS Conference Proceedings*, volume 26, pages 663–672, 1964.

[2] A. Bauer and I. Kavkler. Implementing real numbers with RZ. In Ruth Dillhage, Tanja Grubba, Andrea Sorbi, Klaus Weihrauch, and Ning Zhong, editors, *Proceedings of the Fourth International Conference on Computability and Complexity in Analysis*, volume 202 of *Electronic Notes in Theoretical Computer Science*, pages 365–384, Siena, Italy, June 2007. CCA 2007, Elsevier.

[3] A. Bauer and P. Taylor. The Dedekind reals in abstract Stone duality. *Mathematical Structures in Computer Science*, 19(4):757–838, July 2009.

[4] E. Bishop. *Foundation of constructive analysis.* McGraw-Hill, New York, 1967.

[5] H. J. Boehm, R. Cartwright, M. Riggle, and M .J. O'Donell. Exact real arithmetic: a case study in higher programming. In *ACM Symposium on lisp and functional programming*, 1986.

[6] K. Briggs. xrc (exact reals in c). Available at the website http://keithbriggs.info/xrc.html, 2013.

[7] L. E. J. Brouwer. Besitzt jede reelle zahl eine dezimalbruchentwicklung? *Math. Ann.*, 83(3-4):201–210, 1921.

[8] A. Edalat and M. H. Escardó. Integration in Real PCF. *Information and Computation*, 160(1):128–166, July 2000.

[9] A. Edalat and P. J. Potts. A new representation for exact real numbers. *Electronic Notes in Theoretical Computer Science*, 6:119–132, 1997.

[10] Abbas Edalat and Reinhold Heckmann. Computing with real numbers - I. The LFT Approach to Real Number Computation - II. A Domain Framework for Computational Geometry. In *PROC APPSEM SUMMER SCHOOL IN PORTUGAL*, pages 193–267. Springer Verlag, 2002.

[11] Abbas Edalat and Philipp Sünderhauf. A domain-theoretic approach to computability on the real line. *Theor. Comput. Sci.*, 210(1):73–98, January 1999.

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

[13] M. H. Escardó. Effective and sequential definition by cases on the reals via infinite signed-digit numerals. *Electr. Notes Theor. Comput. Sci.*, 13:53–68, 1998.

[14] M.H. Escardó. Synthetic topology of data types and classical spaces. *Electronic Notes in Theoretic Computer Science*, 87, 2004.

[15] M.H. Escardo and W.K. Ho. Operational domain theory and topology of a sequential language. In *Proceedings of the 20th Annual IEEE Symposium on Logic In Computer Science*, pages 427 – 436. IEEE Computer Society Press, 2005.

[16] P. Di Gianantonio. Real number computability and domain theory. *Information and Computation*, 127(1):11–25, 1996.

[17] J. Gibbons and G. Hutton. Proof Methods for Corecursive Programs. *Fundamentae Informaticae*, 20:1 – 14, 2005.

[18] W. K. Ho. An operational domain-theoretic treatment of recursive types. *Mathematical Structures in Computer Science*, FirstView:1–59, 6 2013.

[19] IEEE. IEEE standard 754 for binary floating-point arithmetic. *SIGPLAN*, 22(2):9–25, 1985.

[20] W. Kramer. A priori worst-case error bounds for floating-point computations. In Tomas Lang, Jean-Michel Muller, and Naofumi Takagi, editors, *13th IEEE Symposium on Computer Arithmetic*, volume 13, pages 64–73, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, July 1997. IEEE Computer Society Press.

[21] D. Lacombe. *Constructivity in mathematics*, chapter Quelques procédés de définitions en topologie recursif, pages 129–158. North-Holland, 1959.

[22] Branimir Lambov. Reallib: An efficient implementation of exact real arithmetic. *Mathematical. Structures in Comp. Sci.*, 17(1):81–98, February 2007.

[23] P. Martin-Löf. *Note on Constructive Mathematics.* Almqvist and Wiksell, 1970.

[24] R. M. May. Simple mathematical models with very complicated dynamics. *Nature*, 261(5560):459–467, 1976.

[25] N. T. Müller. The iRRAM: Exact arithmetic in C++. *Lecture Notes in Computer Science*, 2064:222–252, 2001.

[26] D. Plume. A Calculator for Exact Real Number Computation. 4th year project report, University of Edinburgh, 1998.

[27] P. J. Potts and A. Edalat. Exact real computer arithmetic. Draft available from www.doc.ic.ac.uk/ ae/papers/computerarithmetic.ps.gz, March 1997.

[28] D. S. Scott. Outline of the mathematical theory of computation. In *Proceedings of the 4th Princeton Conference on Information Science*, 1970.

[29] A.K. Simpson. Lazy functional algorithms for exact real functionals. *Lecture Notes in Computer Science*, (1450):323–342, 1998.

[30] E. Specker. Nicht konstruktiv beweisbare sätze der analysis. *J. Symbolic Logic*, 14:145–158, 1949.

[31] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics.* North-Holland, Amsterdam, 1988.

[32] A. M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. Lond. Math. Soc., Ser. 2*, 42:230–265, 1937.

[33] C. Vuik. Some disasters caused by numerical errors. Website at `http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html`.

[34] J. Vuillemin. Exact real computer arithmetic with continued fractions. In *Proc. A.C.M. conference on Lisp and functional Programming*, pages 14–27, 1988.

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

[36] K. Weihrauch. A simple introduction to computable analysis. Technical Report 171-7/1995, FernUniversität, 1995.

[37] K. Weihrauch and C. Kreitz. Representations of the real numbers and of the open subsets of the set of real numbers. *Annals of Pire and Applied Logic*, 35:247–260, 1987.

[38] K. Weihrauch and U. Schreiber. Embedding metric spaces into cpo's. *Theoret. Comp. Sci.*, 16:5–34, 1981.