An Exact Real Package in ML

Shi Shu

Fitzwilliam College

May 14, 2015

Proforma

Name:	**Shi Shu**
College:	**Fitzwilliam College**
Project Title:	**An Exreal Package in ML**
Word Count:	**9371**
Project Originator:	Dr. Arno Pauly
Supervisor:	Dr. Arno Pauly

Original Aims of the Project

The project mainly focuses on building an exact real package and basic arithmetic on the datatype in ML. And then evaluate its performance and comparing it with other exact real implementations such as iRRAM. Then implement mathematical concept such as open/closed set on real numbers and real intervals on developed data structure. Then completing the algorithm of the inverse function calculator using the package will be an extension to the project.

Work Completed

I did research on many of the well developed ways of representing real numbers and choose one representation to implement the exact real number datatype. Basic arithmetic operation is implemented for the developed datatype. And also an inverse function calculator is implemented using the Exreal datatype constructed. The performance of the constructed datatypes and functions is evaluated both theoretically and empirically.

Special Difficulties

None

Declaration

I, Shi Shu of Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

Acknowledgements

Thanks to my supervisor Arno Pauly for useful supervisions and helpful advices on the dissertation.

Chapter 1

Introduction

1.1 Motivation

Many applications of computers require the use of real arithmetic. Unfortunately, there are a number of significant problems associated with performing real arithmetic using floating point approximation, the method most commonly used by programmers, and which forms part of most computer languages and the instruction set of many modern CPUs. These problems stem from the fact that only a finite set of reals are represented exactly using this approach, and rounding occurs after each floating point operation. And the accumulation of rounding errors can lead to highly inaccurate results. They can result in complete loss of accuracy in even relatively simple computations. The exact real datatype would allow us to give a precise representation for any real number and do calculation without rounding error.

1.2 Related Work

1.2.1 Exact Real Arithmetic Systems Competition

A competition between systems for doing exact real number computations was held in September 2000 [4]. They tested the ability of three major exact real number systems at the time in term of performing calculatation like the square

root of π. The three major system that were tested are :

- **MAP** Manchester Arithmetic Package
- **iRRAM** Iterative Real RAM package in C++
- **MPFR** Multiple-precision floating-point computations with correct rounding Library in C[1].

The test result indicates that in most scenorios, the iRRAM package in C++ had the best performance.

1.2.2 iRRAM

The iRRAM is a C++ package for error-free real arithmetic based on the concept of a Real-RAM[11]. Its capabilities range from ordinary arithmetic over trigonometric functions to linear algebra and even differential equations.
A program for the iRRAM is coded in ordinary C++, but may use a special class REAL, that behaves like real numbers without any error. A quite small set of operations is allowed to be used directly with variables of this type: usual arithmetic operations, tests and conversion to/from integer or other types. Programmer can use other programming methods from C++, like defining own data types.

1.2.3 Different focus of this project

Most of the previous exact real packages developed in different languages focus on developing arithmetic functions over the data structure. Many of these packages focus on implementing effective ways of calculating functions such as logarithms and trigonometry etc. While this project focus on developing a package of exact real number using a fundamentally functional approach. Along with basic arithmetic, many other data structures based on the real representation are introduced here as well, such as *open set*, *closed set*, *compact set* on reals and real intervals etc. Also by utilising the admissibility of the chosen real representation and these developed data structure, a function inverse calculator is implemented.

1.3 Project Goals

- Find a proper representation of real number and implement the data structure in ML.
- Implement the basic arithmetic using the developed data structure.
- Implement related mathematical concepts such as *open/closed set, compact set* on real number and real intervals.
- Implement an inverse function calculator using the developed exact real number data type.

Chapter 2

Preparation

2.1 Choice of Real Representation

Since floating point has been proved to be not sufficient under many situations, Exact real representations are mostly based on infinite data streams. Here I introduce three of the commonly used exact real representations.

2.1.1 Real Representation and real number function

Definition 2.1 (Representation) [14]
A representation of the real numbers $\mathbb{R}$ is a surjective function:

$$\delta :\subseteq \Sigma^{\omega} \to \mathbb{R} \tag{2.1}$$

Here Σ^{ω} denotes the set of infinite sequences over the alphabet Σ, the inclusion symbol $\subseteq$ indicates a potentially partial functions. An example of a representation is the ordinary decimal representation.
Definition 2.2 (Computable real number function) [14]
A function $f :\subseteq \mathbb{R} \to \mathbb{R}$ is called computable with respect to some representation $\delta :\subseteq \Sigma^{\omega} \to \mathbb{R}$, if there exists some Turing machine M with one-way output tape which computes infinitely long and which in the long run transforms each sequence $p \in \Sigma^{\omega}$ which represents some $x := \delta(p) \in \mathbb{R}$ into some sequence $q \in \Sigma^{\omega}$

which represents $f(x)$, i.e. $f(x) = \delta(q)$.

With these definitions, we can see why ordinary decimal representation is not sufficient in some cases. For example, multiplication by 3, *i.e* . the real function $f : \mathbb{R} \to \mathbb{R}$, $x \to 3x$ is not computable with respect to the decimal representation [7].
To show $3x$ is not computable is quite easy: each machine which would compute multi- plication by 3 w.r.t. the decimal representation has to transform the input $p = 0.33333\ldots$ or to either to $q = 0.9999....$ or to $q^{'} = 1.0000....$ Especially, the machine has to write 0.9 or 1.0 on the output tape after some finite time, but no finite prefix of the input sequence p suffices to decide whether the correct output sequence has to start with 0.9 or with 1.0. Consequently, such a machine cannot exist.
Thus other representations of real numbers are introduced.

2.1.2 Sequence of Interval Representation

Definition 2.3 (Sequence of Interval Representation)[14]
Define $\rho_I :\subseteq \Sigma^{\omega} \to \mathbb{R}$ as follows :

$$\begin{aligned} \rho_I(p) = x \text{ , if and only if there are } u_0, v_0, u_1 v_1 .... \in \mathbb{Q} \\ \text{with } p = [u_0, v_0], [u_1, v_1], ... \text{ and } x = sup(u_i) = inf(v_i) \end{aligned} \tag{2.2}$$

With this representation, basic arithmetic can be easily implemented, basically doing rational arithmetic on all the end point of all the intervals. Thus initially it would seem to be a nice representation to implement, but this implementation would have a very high memory requirement for every single exreal. Even utilising the laziness of ML, each real number would require at least 2 rational numbers to a rather high precision. Hence this representation is far from ideal for computer implementation.

2.1.3 Cauchy Representation

Definition 2.4 (Cauchy Representation)[14]
Define $\rho_C :\subseteq \Sigma^\omega \to \mathbb{R}$ as follows :

$$\begin{aligned} \rho_C(p) = x \text{ , if and only if there are } u_0, u_1, u_2.... \in \mathbb{Q} \\ \text{with } p = u_0|u_1|u_2.... \text{ and} \\ \forall k\ \forall i > k\ |u_i - u_k| < 2^{-k} \text{ and } x = \lim_{i\to\infty} u_i \end{aligned} \tag{2.3}$$

This representation is equivalent to the sequence of interval representation (proof can be found here [14]) and with a minimal guaranteed converging speed. Thus the space usage problem remains for computer implementation.

2.1.4 Signed Digit Representation

Definition 2.5 (Signed Digit Representation)[14]
Define $\rho_S :\subseteq \Sigma^\omega \to \mathbb{R}$ as follows :

$$\begin{aligned} \rho_S(p) = x \\ \text{if and only if there are } u_1, u_2, u_3....\text{and } v_1, v_2, v_3...v_m \in \{1, 0, -1\} \\ \text{with } p = v_m v_{m-1}...v_2 v_1.u_1 u_2 u_3.... \text{ and} \\ x = \sum_{i=1}^{m} 2^i * v_i + + \sum_{i=1}^{\infty} 2^{-i} * u_i \end{aligned} \tag{2.4}$$

Signed digit representation is defined like the decimal representation but which also allows negative digits. Most concrete continuous functions which are used in analysis are computable in this sense. The actual implementation can work as a proof for addition and multiplication.
Theorem 2.1 (**Undecidability of the equality**)

The equality test on the real numbers is not decidable, *i.e* . the function

$$s : \mathbb{R}^2 \to \mathbb{R}, (x, y) \mapsto \begin{cases} 0 & \text{if } x = y \\ 1 & \text{else} \end{cases} \tag{2.5}$$

is not computable[7]. The undecidability of the equality is not a bad property of the signed-digit representation (in the sense as the non-computability of the multiplication by 3 is a bad property of the decimal representation). Unfortunately equality is undecidable with respect to all representations of the real numbers. In other words: there is no way to represent real numbers which could enable physical computers to decide equality and that make the next data type introduced crucial in our implementation.

2.2 Sierpinski Space

In mathematics, the Sierpinski space (or the connected two-point set) is a finite topological space with two points, only one of which is closed.
The Sierpinski space is a topological space defined as

- The underlying set is a two-point set $S = \{ \top, \bot \}$.
- The open subsets are {} , {$\bot$}, {$\top$,$\bot$} , the closed subset are {}, {$\top$}, {$\top$, $\bot$}.

Based on the definition we can consider $\top$ (or true) as the "observable true" and $\bot$ as "unobservable false". Sierpinski Space is then used in place of boolean in the developed real number systems. Due to the undecidability of inequality between real numbers, we use this semi decidable data structure to implement the undecidable not-equality of real numbers.

2.3 Topology

2.3.1 Open Sets

Definition 2.6 [9]
A set $G \subset \mathbb{R}$ is open if for every $x \in G$ there exist a $\delta > 0$ such that $G \supset (x\text{-}\delta,\ x\text{+}\delta)$. For every open set, a characteristic function can defined as :

$$C : \mathbb{R} \to \text{Boolean} \ , x \mapsto \begin{cases} \text{true} & \text{if } x \text{ is in the open set} \\ \text{false} & \text{else} \end{cases} \tag{2.6}$$

Theorem 2.2 An arbitrary union of open sets is open, and a finite intersection of open sets is open.
Here we introduced the idea of *synthetic topology* [8] for implementation purpose. In synthetic topology we define the open set as,
Definition 2.7 [8] A subset U of a data type a is called open if its characteristic function $f_U : a \to$ Sierpinski Space defined by $f_U(x) = \top$ if and only if $x \in U$ is continuous.

2.3.2 Closed Sets

Definition 2.8 A set $F \subset \mathbb{R}$ is closed if $F^c = \{\ x \in \mathbb{R} : x \notin F\ \}$ is open.
Theorem 2.3 An arbitrary intersection of closed sets is closed, and a finite union of closed sets is closed.
The implementation of open/closed set on exact real number in **Section 3.3** can show that any countable unions/intersections of open sets is an open set and any countable unions/intersections of closed sets is a closed set.

2.3.3 Compact Sets

There are different ways of defining *compact set*.
Definition 2.9 Sequential Definition [9]
A set $K \subset \mathbb{R}$ is sequentially compact if every sequence in K has a convergent

subsequence whose limit belongs to K.

Intuitively, a compact set confines every sequence of points in the set so much that the sequence must accumulate at some point of the set.

Definition 2.10 Topological Definition [9]

Let A $\subset$ R. A cover of A is a collection of sets $\{A_i \subset \mathbb{R} : i \in I\}$ whose union contains A,

$$\bigcup_{i \in I} A_i \supset A \tag{2.7}$$

An open cover of A is a cover such that A_i is open for every $i \in I$.

Suppose that $C = \{A_i \subset R : i \in I\}$ is a cover of $A \subset R$.

A subcover S of C is a sub-collection $S \subset C$ that covers A, meaning that

$$S = \{A_{i_k} \in C : k \in J\}, \bigcup_{k \in J} A_{i_k} \supset A \tag{2.8}$$

A finite subcover is a subcover $\{A_{i_1}, A_{i_2}, ..., A_{i_n}\}$ that consists of finitely many sets.

Then we define a set $K \subset R$ to be compact if every open cover of K has a finite subcover.

Also for implementation purpose, a synthetic approach to define *compact sets* is introduced as following [8],

Definition 2.11 [8]

We call a subspace Q of a data type a compact if its universal quantification function $\forall_Q : (a \to SierpinskiSpace) \to SierpinskiSpace$ defined by
$\forall_Q(p) = \top$ if and only if $p(x) = \top$ for all $x \in Q$ is continuous. Here p is an arbitrary predicate function that has the type $p : a \to SierpinskiSpace$.

Definition 2.11 is equivalent to Definition 2.10. The reason that the sequential compactness can not be used here is that it does not give rise to a computable type [5].

Theorem 2.4 The intersection of a compact set and a closed set is a compact set.
It is proven in *On the topological aspects of the theory of represented spaces* Proposition 11.4 by Arno Pauly [12].

2.4 Admissibility

Definition 2.12 Admissimility
Let $\delta_{\mathbf{X}} : \Sigma^\omega \to \mathbf{X}$ and $\delta_{\mathbf{Y}} : \Sigma^\omega \to \mathbf{Y}$ be representations on $\mathbf{X}$ and $\mathbf{Y}$. Here $\mathbf{X}$ and $\mathbf{Y}$ are topological spaces. $\delta_{\mathbf{Y}}$ is an admissible representation if and only if for any continuous function $f : \mathbf{X} \to \mathbf{Y}$, there exist a continuous function $F : \Sigma^\omega \to \Sigma^\omega$ such that it makes the following diagram commutes.

$$\begin{array}{ccc} \Sigma^\omega & \xrightarrow{F} & \Sigma^\omega \\ \downarrow{\scriptstyle \delta_{\mathbf{X}}} & & \downarrow{\scriptstyle \delta_{\mathbf{Y}}} \\ \mathbf{X} & \xrightarrow{f} & \mathbf{Y} \end{array}$$

One equivalent definition is for any singleton compact set under $\delta_{\mathbf{X}}$ representation, the function $f = \{\mathbf{x}\} \to \mathbf{x} : Compact(\mathbf{X}) \to \mathbf{X}$ is computable. Then $\delta_{\mathbf{X}}$ is an admissible representation.

The signed digit representation of real number is an admissible representation of real numbers [14]. An algorithm of how to use admissibility can be described as below [12],

- Conclude that the solution set S is closed based on its definition.
- Obtain some compact candidate set K (either from the situation, or by assumption - this often involves some bounds).
- Compute $Intersect(S, K)$ as a compact set via **Theorem 2.4**.
- Use domain-specific reasoning or assumption to conclude that the solution s is unique.
- As we have singleton {s} available as a compact set, if our target space is admissible, we can compute s.

This work as our key algorithm to implement the inverse function calculator.

2.5 Software Development Approach

2.5.1 Development Environment

The development and testing environment is in Mac OS X 10.10 with 2GHz Intel Core i7 CPU and 8GB 1600 MHz DDR3 memory.

2.5.2 Programming Language and Software

ML is the choice of language here since many infinite data structure such as Exreal and Sierpinski Space will be implemented and the laziness that ML provide us with easy evaluation of these data structure.
Among different versions of ML available, Poly/ML is used here. Poly/ML was originally written by David Matthews at the Computer Laboratory at Cambridge University. It was written in an experimental language, Poly, similar to ML but with a different type system. The current version 4.0 release Poly/ML supports the full version of the language as given in the "Definition of Standard ML (Revised)", generally known as ML97. [2]
During the testing phase, New Jersey ML [3] is also tried for performance comparison and no significant difference is noticed.

2.5.3 Back Up

Code and Dissertation are backed up using both Dropbox and physical external hard drive.

2.5.4 Extendability of the code

For easier later expansion and possible comparison, most functions that work on exreal (except basic arithmetic) are independent of the representation of the exact real numbers. Thus if another representation of real number is implemented then most of the code would still be usable on the new representation with minimal modification.

Chapter 3

Implementation

3.1 Real Representation

3.1.1 Exreal Datatype

Among many representations of real numbers, signed binary representation is used here. Real number are resented by a regular integer and a lazy list of signed binaries, which consist of {1,0,-1}.

```
datatype 'a seq = Cons of 'a * (unit -> 'a seq);
datatype exreal = R of int * int seq;
```

Here we assume that all real numbers have a infinitely long fraction part. We write finite length real number (rationals) such as 0.75 as :

```
> fun zeros () = Cons(0,fn ()=>zeros ());
val zeros = fn: unit -> int seq
> val rational = (R(0,Cons(1,fn ()=>(Cons(1,fn ()=>zeros())))));
val rational = R (0, Cons (1, fn)): exreal
```

3.1.2 Conversion Between real and Exreal

To allow the easy conversion, conversion function between reals and exreals are introduced. Thus user can built in real as input, convert it into exreal and after certain operations, convert back to real and return a real number result as expected.

From exreal to real, we need user to input a exreal and an integer which indicates how many bits in the lazy list of fractions it requires.

```
fun ll2r _ 0 _ = 0.0
|   ll2r (Cons(x,xq)) n i =
(Real.fromInt(x)/(Math.pow(2.0,i))) + (ll2r (xq()) (n-1) (i+1.0));
val ll2r = fn: int seq -> int -> real -> real
fun exr2r (R(i,f)) n = Real.fromInt(i)+(ll2r f n 1.0);
val exr2r = fn: exreal -> int -> real
```

On the other hand, from real number to exreal, first change the real into binary to the desired precision. Then an integer sequence filled with 0s is attached to the tail to make the returned result exreal.

```
fun r2ll x 0 i = zeros()
|   r2ll x n i = if Real.signBit(x - 1.0/Math.pow(2.0,i))
      then Cons(0,fn ()=> r2ll x (n-1) (i+1.0))
      else Cons(1, fn ()=> r2ll (x- 1.0/Math.pow(2.0,i)) (n-1) (i+1.0));
val r2ll = fn: real -> int -> real -> int seq
fun r2exr r n = (R(floor r, r2ll (r - Real.fromInt(floor r)) (n) (1.0)));
val r2exr = fn: real -> int -> exreal
```

A simple conversion example as shown below :

```
> val test = r2exr 2.3 100;
val test = R (2, Cons (0, fn)): exreal
> exr2r test 5;
val it = 2.28125: real
> exr2r test 10;
val it = 2.299804688: real
> exr2r test 100;
val it = 2.3: real
```

3.1.3 Addition

In exreal addition, the importance of having the signed binary representation instead of standard binary can be shown. In exact real arithmetic, the major difficulty would be calculation needs to be done from left to right. And bits needs to be determined without allowing the carryovers from following bits to affect this bit.

To add two exreal numbers, first we add the two integer and the two lazy list correspondingly. The bitwise addition of two fraction part of exreals has a result of a lazy list consist of {2,1,0,-1,-2}.

```
> fun frac_sum1 (Cons(x,xq),Cons(y,yq)) =
      Cons(x+y,fn ()=>(frac_sum1(xq(),yq())));
val frac_sum1 = fn: int seq * int seq -> int seq
```

Then evaluation of the result lazy list is required.
To determined the current bit, two further bits would be required to be evaluated. By examining the third bit, possible carryovers from the third bit to the second bit can be determined. Then with the possible carryovers from the third bit and current second bit, the possible carryovers from the second bit to current bit can be determined. And then the signed binary representation will help to eliminate the possible carryover to the current bit. For example, we have a sequence of 011... then the third bit is 1 and thus the possible carryover to the second bit is [0,1]. Since the second bit is currently one and with the possible carryover [0,1], thus second bit has a possible carry over of [0,1] to the current bit. Then we can set second bit to -1 and set the current bit to 1 to avoid any possible carryovers to the current bit. Then recursively evaluate the sequence -11.... It seems to have $5^3 = 125$ possibilities. But in reality many of can never occur due to the pre-evaluation. All possible different situation can be listed as below.

Current Bit	Next 2 Bits	Possible Carryover	Evaluated Result
1	2, _	1	0, 0, ...
1	1,0	0	1, 1, ...
1	1,-1	0	1, 1, ...
1	1,-2	0	1, 1, ...
1	1,1	0,1	0, -1, ...
1	1,2	1	0, -1, ...
1	0,_	0	1, 0, ...
1	-1,-1	0,-1	0, 1, ...
1	-1,-2	0,-1	0, 1, ...
1	-1,0	0	1, -1, ...
1	-1,1	0	1, -1, ...
1	-1,2	0	1, -1, ...
1	-2,_	-1	0, 0, ...
0	2,_	1	1, 0, ...
0	1,2	1	1, -1, ...

0	1,1	0,1	1, -1, ...
0	1,-2	0	0, 1, ...
0	1,-1	0	0, 1, ...
0	1,0	0	0, 1, ...
0	0,_	0	0, 0, ...
0	-1,-2	-1	-1, 1, ...
0	-1,-1	0,-1	-1, 1, ...
0	-1,2	0	0, -1, ...
0	-1,1	0	0, -1, ...
0	-1,0	0	0, -1, ...
0	-2,_	-1	-1, 0, ...
-1	2,_	-1	0, 0, ...
-1	2,_	1	0, 0, ...
-1	1,1	0,1	0, -1, ...
-1	1,2	1	0, -1, ...
-1	1,0	0	-1, 1, ...
-1	1,1	0	-1, 1, ...
-1	1,2	0	-1, 1, ...
-1	0,_	0	-1, 0, ...
-1	-1,-1	0,-1	0, 1, ...
-1	-1,-2	-1	0, 1, ...
-1	-1,0	0	-1, -1, ...
-1	-1,1	0	-1, -1, ...
-1	-1,2	0	-1, -1, ...

Then combining the result of the evaluation and the integer addition by adding/subtracting the carryover from the first bit to the integer part. Exreal addition can be integrated as,

```
> fun rplus (R(x1,y1)) (R(x2,y2)) =
     let val temp = frac_sum1(y1,y2) in
        if head(temp) = 2
        orelse (take(2,temp)=[1,2]
        orelse (take(3,temp)=[1,1,1]
        orelse take(3,temp)=[1,1,2]))
```

```
        then (R(x1+x2+1,eval_result (temp)))

        else if head(temp) = ~2
        orelse (take(2,temp)=[~1,~2]
        orelse (take(3,temp)=[~1,~1,~1]
        orelse take(3,temp)=[~1,~1,~2]))
        then (R(x1+x2-1,eval_result (temp)))

        else (R(x1+x2,eval_result(temp)))
     end;
val rplus = fn: exreal -> exreal -> exreal
```

Also addition of a list of exreals is also implemented for the ease of implementing multiplication.

3.1.4 Multiplication

Multiplication can be divided into addition of 4 components. Let's say we want to multiply A and B. Let $A = (R(int1, frac1))$ and $B = (R(int2, frac2))$. Then we have

$$\begin{aligned} A * B &= (int1 + frac1) * (int2 + frac2) \\ &= int1 * int2 + int1 * frac2 + int2 * frac1 + frac1 * frac2 \end{aligned} \tag{3.1}$$

The first part of the result is trivial. Two functions need to be implemented is integer multiply with a signed binary lazy list and multiplication between two lazy lists. To multiply a integer with a lazy list. First convert the integer into a binary form. Let's say that the integer is converted into a n bit binary integer then the multiplication become into the addition of a list of n Exreals.
For example, multiplying 10 and $[1, 1, 1, 1, 1, ...]$. First , 10 needs to be converted into $[1, 0, 1, 0]$, which is the binary representation of 10. Then reverse the result we have $[0, 1, 0, 1]$. Then we have the list of exreals as $[0 * R(0, [1, 1, 1, ...]), 1 * R(1, [1, 1, 1, ...]), 0 * R(3, [1, 1, 1, ...]), 1 * R(7, [1, 1, 1, ...])]$. Using Exreal addition implemented in the previous section to add them together, we have the result as $R(9, [1, 1, 1, ...])$.
A similar idea is also used for lazy list multiplied with lazy list. First, since the fraction part of an Exreal is within the interval $[-1, 1]$ and thus multiply two fraction parts of two Exreal, the result is still in the interval $[-1, 1]$. Thus the

result still can be represented by a lazy list.
Multiply a lazy list $[x_1, x_2, x_3, ...]$ with another lazy list y, we have $x_1 * [0, y] + x_2 * [0, 0, y] + x_3 * [0, 0, 0, y] + ...$ So the problem become an addition of a lazy list of lazy lists. The difficulty here is to determine how many addition needs to be done to determine the n^{th} bit. From the addition implementation, it can be learned that $(n+1)^{th}$ bit and $(n+2)^{th}$ bit are needed to be examined to determine the n^{th} bit. Thus the n^{th} bit in the result is the n^{th} bit of the result of adding the first $n + 2$ elements in generated lazy list.
Using (3.1), the Exreal multiplication can be implemented as,

```
> fun rmult (R(i1,f1)) (R(i2,f2)) =
     rplus (R(i1*i2, mult_frac (f1) (f2)))
          (rplus (mult_frac_int i1 (f2)) (mult_frac_int i2 (f1)));
val rmult = fn: exreal -> exreal -> exreal
```

3.2 Sierpinski Space

The Sierpinski Space is implemented using a lazy list of booleans.

```
> datatype Sierpinski = Sierp of bool seq;
```

By using this implementation, the Sierpinski space is $\top$ if and only if there is at least one true in the lazy list.

```
> fun sierp_is_top (Sierp(Cons(x,xq))) =
     if x = true
     then true
     else sierp_is_top (Sierp(xq()));
val sierp_is_top = fn: Sierpinski -> bool
```

This function will not terminate if the Sierpinski Space is bottom, which is false all the time. Thus for testing purposes, the following functions are used for evaluating Sierpinski space, which evaluate the first n entries in the lazy list.

```
> fun eval_sierp_aux _ 0 = false
|     eval_sierp_aux (Cons(true,xq)) n = true
|     eval_sierp_aux (Cons(false,xq)) n = eval_sierp_aux (xq()) (n-1);
val eval_sierp_aux = fn: bool seq -> int -> bool

> fun eval_sierp x n = eval_sierp_aux (sierp_extract_seq(x)) n;
val eval_sierp = fn: Sierpinski -> int -> bool
```

Using the lazy list representation, the OR operation of two Sierpinski Space can be done by interleaving the two lazy list.

```
> fun interleave (Cons(x,xf), yq) = Cons(x, fn()=> interleave(yq, xf()));
val interleave = fn: 'a seq * 'a seq -> 'a seq

> fun sierp_or (Sierp(a)) (Sierp(b)) = Sierp(interleave(a,b));
val sierp_or = fn: Sierpinski -> Sierpinski -> Sierpinski
```

The AND operation of two Sierpinski Space A and B can be done by first check the first element of A if it is true then the result Sierpinski Space is B, if not then add a false to the result Sierpinski Space and keep evaluating B and tail(A). Here an easy mistake is to forget to output a false to result at every step, since if both inputs are $\perp$ then the function would keep interleaving between the inputs and give no return. Thus a key thing to keep in mind when implementing functions that return Sierpinski Space as a result, is to make sure the function would make progress at every iteration regardless of the inputs.

```
> fun sierp_extract_seq (Sierp(x)) = x;
val sierp_extract_seq = fn: Sierpinski -> bool seq

> fun sierp_and (Sierp(Cons(x,xf))) (Sierp(Cons(y,yf))) =
     if x=true then (Sierp(Cons(y,yf)))
     else (Sierp(Cons(false,fn ()=>
          sierp_extract_seq (sierp_and (Sierp(Cons(y,yf))) (Sierp(xf()))))));
val sierp_and = fn: Sierpinski -> Sierpinski -> Sierpinski
```

The multiple OR of a lazy list of Sierpinski Space can be done by first checking the first boolean of the first Sierpinski Space of the lazy list if it is true then the result is top or otherwise adding a false to the result and recursively calling it self on the rest of the lazy list and interleaving the result with the tail of the first Sierpinski space in the lazy list.

```
> fun c_true () = Cons(true, fn ()=>c_true());
val c_true = fn: unit -> bool seq

> fun multi_sierp_or (Cons((Sierp(Cons(x,xq))),sf)) =
     if x = true
     then (Sierp(c_true()))
     else (Sierp(Cons(false, fn()=>
          interleave (sierp_extract_seq(multi_sierp_or (sf())),(xq())))));
val multi_sierp_or = fn: Sierpinski seq -> Sierpinski
```

3.3 Open and Closed set on Exreal

Both *open set* and *closed set* are defined by their characteristic functions. Here a function that takes an input of type a' and returns a Sierpinski Space can be considered as a characteristic function of an open/closed set on type a'. Thus the datatype open and closed set can be defined as,

```
> datatype 'a opens = cfo of ('a -> Sierpinski);

> datatype 'a closed = cfc of ('a -> Sierpinski);
```

This implementation allows finding the compliment of an open/closed set to become trivial.

```
> fun complementc (cfc(x)) = (cfo(x));
val complementc = fn: 'a closed -> 'a opens

> fun complemento (cfo(x)) = (cfc(x));
val complemento = fn: 'a opens -> 'a closed
```

The UNION of two open sets with characteristic function f_1 and f_2 can be implemented by using the function

$$f(x) = Sierpinski_{OR}(f_1(x), f_2(x)) \tag{3.2}$$

as the characteristic function of the result open set. Similarily INTERSECTION can be done by using

$$f(x) = Sierpinski_{AND}(f_1(x), f_2(x)) \tag{3.3}$$

as the characteristic function of the result open set.

```
> fun opens_union (cfo(x)) (cfo(y)) =
     cfo (fn a => (sierp_or (x a) (y a)));
val opens_union = fn: 'a opens -> 'a opens -> 'a opens

> fun opens_intersection (cfo(x)) (cfo(y)) =
     (cfo(fn a => sierp_and (x a) (y a)));
val opens_intersection = fn: 'a opens -> 'a opens -> 'a opens
```

3.4 Inequality and relational operator

3.4.1 notEqual

As stated in **Theorem 2.1** in 2.1.4, the equality problem between two Exreals are undecidable. Hence instead of mapping notEqual x y $\mapsto$ bool , we try to map notEqual x y $\mapsto$ Sierpinski Space. The idea is to add a false to the result Sierpinski Space whenever it is still possible for x and y to be equal (the unobservable false) or change the result to *Sierp*(*c_true*()) (the observable true). One of the side effect of the signed digit representation is that for any particular real number the representations are not unique. In fact, every real number has infinitely many Exreal representations. Thus the notEqual function is not trivial.
Consider two exreal *R(ix, fx)* and *R(iy, fy)*, we first look at the integer part of the two exreals.

- If $|ix - iy| > 2$ then just simply return *Sierp(c_true())*
- If $|ix - iy| = 2$ then either $fx = [-1, -1, -1...]$, $fy = [1, 1, 1...]$ or $fx = [1, 1, 1...]$, $fy = [-1, -1, -1...]$. Thus we have,

```
> fun checkm11 (Cons(x,xq),Cons(y,yq)) =
      if x= ~1
      then if y= 1
         then Cons (false, fn ()=>checkm11(xq(),yq()))
         else c_true()
      else c_true();
val checkm11 = fn: int seq * int seq -> bool seq
> fun check1m1 (Cons(x,xq),Cons(y,yq)) =
      if x=1
      then if y= ~1
         then Cons(false, fn ()=> check1m1(xq(),yq()))
         else c_true()
      else c_true();
val check1m1 = fn: int seq * int seq -> bool seq
```

  In these two cases, the notEqual function would look like
  fun notEqual (R(ix,fx)) (R(iy,fy)) = Sierp(checkm11(fx,fy)) , fun notEqual (R(ix,fx)) (R(iy,fy)) = Sierp(check1m1(fx,fy)).

- If $ix = iy$ then we look at *fx* and *fy*. If $|head(fx) - head(fy)| = 2$ then again use checkm11 and check1m1 to generate the result. If $|head(fx) = head(fy)|$

then add a false to the result and then look at tail(fx) and tail(fy). If $head(fx) - head(fy) = 1$ then subtract 2 from head(fx) then the function would reach one of the other cases or return $Sierp(c\_true())$. Similarly for $head(fy) - head(fx) = 1$.

- If $ix - iy = 1$ Then subtract 2 from head(fx) then the function would reach one of the other cases or return $Sierp(c\_true())$. Also similarly for $iy - ix = 1$.

3.4.2 isPositive

Relational operations are also not decidable in Exreals. Using the same idea, we return a Sierpinski Space as result. If exreal x is possible to be positive then add a false to the result otherwise return $Sierp(c\_true())$. The style of code is quite similar to notEqual. With the function isPositive, we can easily have our relational operator inplemented as :

- fun greatthan x y = isPositive (rplus x $r\_minus(y)$)
- fun lessthan x y = isPositive (rplus $r\_minus(x)$ y))

The isPositive and relational operation functions are essential to implementing Exreal intervals later.

3.5 Real Interval

The interval on exreal is defined as a tuple of two exact real numbers, the left boundary and the right boundary. The interval datatype defined here is the open interval where the two boundary points are considered not in the interval.

```
datatype Interval = I of exreal * exreal;
```

One most important function regarding real intervals would be generating the characteristic function of a given interval. If an interval (a,b) is given, then the

characteristic function of this interval should perform as given any exreal input x it returns a Sierpinski Space result which is top if and only if the exreal $x \in (a, b)$. Thus both $x > a$ and $x < b$ should be satisfied. And these two conditions would be equivalent to $x - a > 0$ and $b - x > 0$. Thus using isPositive in 3.4.2 and intersection of the open set in 3.3, the characteristic function can be written as,

$$f(x) = Sierpinski_{AND}(isPositive(x - a), isPositive(b - x)) \tag{3.4}$$

Since the characteristic function has the form of taking an exreal and goes to a Sierpinski Space. We can then make any real intervals an open set on exact reals.

```
fun makeinterval (I(l,r)) = cfo(fn z =>
   (let val a = (isPositive ((rplus (z) (r_minus(l))))) in
      let val b = (isPositive ((rplus (r) (r_minus(z))))) in
         (sierp_and (a) (b))
      end
   end));
```

3.6 Disguised Exreal

3.6.1 Prefix Real

Prefix Real is a datatype we defined as the truncated exact real number for internal use, which also use signed digit representation. A prefix real consist of an integer and a integer list instead of an integer with an integer lazy list.

```
datatype prefix_real = pref_r of int * int list;
```

One application is that we can make an interval from a given prefix real. Let's say we have a prefix real $x = pref\_r(i, f)$, then we can define the function make_pref_interval as :

$$\begin{aligned} &make\_pref\_interval(x) = \\ &makeinterval(I(R(i, f@[-1, -1, -1, ...]), R(i, f@[1, 1, 1, ...]))) \end{aligned} \tag{3.5}$$

3.6.2 Disguised Exreal

Converting between exreal and prefix real numbers are not always trivial. One direction is quite trivial, if we want to cast an exreal number into a prefix real number with n bits in the fraction part, we simply take the first n bit of the fraction part of the exreal number and truncate the rest.

```
fun exr2pref (R(i,f)) n = pref_r(i,get(n,f));
```

But in the other direction, casting an prefix real into an exreal number is not trivial. Here the idea of disguised exreal is introduced. Let's say we have a prefix real number *pref_r(i,f)*, where f has n bits. First we introduce an exception we named as *Do_Not_Look*, which means do not look at any further bits. Then we can cast an integer list into an integer lazy list by copying all the bits from the list to the lazy list and raise a *Do_Not_Look* exception when it reaches the end of the list.

```
exception Do_not_Look;

fun disguise_aux (x::xs) = Cons(x,fn ()=> disguise_aux xs)
|     disguise_aux [] = raise Do_not_Look;

fun disguise (pref_r(x,xs)) = (R(x,(disguise_aux(xs))));
```

The ML typing system would consider the disguise function returns an exreal number as a result. And the returned result can be used in any exreal functions. The only difference would be disguised exreals would raise *Do_Not_Look* exception when extra information is required.

3.7 Compact set on Exreal

Based on **Definition 2.11**, a compact set on exact real number can be described as an open set of open set of exact real numbers. Thus we can define it as :

```
datatype 'a cptset = cpt of (('a opens) opens);
```

One key function that wen can operate on compact set is the intersection of a closed set with an compact set. A closed set intersect with a compact set result in a compact set which has the characteristic function [12] :

$$Compact(x) = f(UNION(x, Complement(g))), \tag{3.6}$$

where f is the characteristic function of the compact set and g is the chacharacteristic function of the closed set. Thus the intersection can be implemented as:

```
fun cfc_intsec_cpt (f) (cpt(cfo(x : (('a opens)->Sierpinski)))) =
(cpt(cfo(fn a => (x (opens_union (complementc(f)) a)))));
> val cfc_intsec_cpt = fn: 'a closed -> 'a cptset -> 'a cptset
```

Now we try to show that the unit interval is a compact set and construct the unit interval as a compact set. From **Definition 2.11**, we need to construct the universal quantification function for unit interval $[-1, 1]$.
By definition, given an arbitrary predicate function this function should return a Sierpinski Space element that maps to $\top$ if and only if all element with in the unit interval would return $\top$ when applied to the predicate function. In order to achieve that, we make use of the disguised Exreal. Let's assume our predicate function is f. When we apply a disguised Exreal $x = disguise(pref\_r(i, frac))$ to f, it reaches one of three following situations,

- **Case 1** Returns a Sierpinski Space starting with false. Then we keep evaluating until it reaches one of the other two cases.
- **Case 2** Returns a Sierpinski Space starting with true. This means that the given information is enough to show that $f(x) \to \top$. Hence all Exreal number starting with this prefix would map to $\top$ when applied to f as well.
- **Case 3** Raise a *Do_not_Look* exception.

Thus we start with a list with only one element, the prefix real number *pref_r(0,[])*. Then we try to apply the predicate function to every element x_i after being disguised in the list , if we get **Case 1**, we keep evaluating; if we get **Case 2**, we delete the element from the list, since all Exreal with this prefix

would return $\top$ as well; if we get an exception, then we handle it by generating three new prefix real number by add 1,0,-1 to tail of x_i and add them to the tail of the list. And we add a false to the result Sierpinski Space as long as the list is not empty and add *c_true()* to the result Sierpinski Space if we reaches the empty list.

```
> fun generate_pref (pref_r(x,xs)) =
      [(pref_r(x,xs@[1])),(pref_r(x,xs@[0])),(pref_r(x,xs@[~1]))];
val generate_pref = fn: prefix_real -> prefix_real list

> fun compact1 f [] _ = (Sierp(c_true()))
|   compact1 f (x::xs) x_result =
      (if (head(sierp_extract_seq(x_result()))) = true
         then if (xs=[]) then (Sierp(c_true())) else
         (compact1 f xs (fn () => (f (disguise (headls(xs))))))
      else (Sierp(Cons(false, fn ()=>
         (sierp_extract_seq(compact1 f (x::xs) (fn () =>
         (Sierp((tail(sierp_extract_seq(x_result()))))))))))
         handle Do_not_Look =>
         (Sierp(Cons(false, fn ()=> (sierp_extract_seq(
         compact1 f (xs@(generate_pref x)) (fn () =>
         (f (disguise (headls(xs@(generate_pref x))))))))))));
val compact1 = fn: (exreal -> Sierpinski) ->
    prefix_real list -> (unit -> Sierpinski) -> Sierpinski

> fun compact f = compact1 f [(pref_r(0,[]))] (fn () =>
       (f (disguise (pref_r(0,[]))))));
val compact = fn: (exreal -> Sierpinski) -> Sierpinski
```

Proposition: Every real number has a signed digit name WITHOUT consecutive 11 or 1-1 or -1-1 or -11. [13] Proof can be found in *Relative Computability and Uniform Continuity of Relations* by Arno Pauly and Martin Ziegler.

Then we can improve the *generate_pref* function to become,

```
> fun generate_pref2 (pref_r(x,xs)) =
      let val temp = (last_ele xs) in (if temp=1 then [(pref_r(x,xs@[0]))]
         else if temp= (~1) then [(pref_r(x,xs@[0]))]
         else [(pref_r(x,xs@[1])),(pref_r(x,xs@[0])),(pref_r(x,xs@[~1]))])
      end;
val generate_pref2 = fn: prefix_real -> prefix_real list
>
```

Thus with the compact function and followed from definition 2.11, we can construct the unit interval as a compact set,

```
> val unitint = (cpt(cfo(fn U => compact(getf U))));
val unitint = cpt (cfo fn): exreal cptset
```

3.8 Inverse function Calculator for Exreal

The idea of constructing an inverse function calculator is that the result of $f^{-1}(y) = x$, if x is unique then we can have a closed singleton set $\{x\}$. Then we can intersect any compact set which contains this singleton x with this closed set. The result will be a compact set contains this singleton x. Then we use admissibility of the developed Exact Real to extract x from the compact set.
Note: the inverse function calculator has a limitation that it can only compute $f^{-1}(y)$ if and only if $f^{-1}(y)$ maps to a unique Exact Real x.

3.8.1 Admissibility of Exreal

Here we restrict the admissibility to the unit interval, [-1,1]. Since for any arbitrary real number x, we can always let $x' = x/n$ for some natural number n, such that $x' \in [-1, 1]$. Then we can use admissibility on x' and then get x by $x = x' * n$.
To extract x from singleton $\{x\}$, we start with a prefix real with an empty list, *pref_r(0,[])*. Each step, we try to extract one bit of x by testing whether the Exact Real interval generated by the current prefix real number contains x. If yes, we would find that the result of applying the open set, which is the interval generated by the current prefix real number, to the compact set characteristic function would return a result Sierpinski Space that eventually maps to $\top$. After finding the current bit, we keep the current prefix and generate three new prefix real by add 1,0,-1 to the tail respectively and perform the same operation to get the next bit.

```
> fun findindex_f
      (Sierp(Cons(x,xf))) (Sierp(Cons(y,yf))) (Sierp(Cons(z,zf)))
      = if x=true then 1 else
      if y=true then 0 else
      if z=true then ~1
      else findindex_f (Sierp(xf())) (Sierp(yf())) (Sierp(zf()));
val findindex_f = fn: Sierpinski -> Sierpinski -> Sierpinski -> int

> fun admisibility_aux x (pref_r(i,f)) =
      let val temp = (findindex_f
         (x (make_pref_interval((pref_r(i,f@[1])))))
         (x (make_pref_interval((pref_r(i,f@[0])))))
         (x (make_pref_interval((pref_r(i,f@[~1]))))))
      in
```

```
      Cons(temp, fn ()=> (admisibility_aux x (pref_r(i,f@[temp]))))
      end;
val admisibility_aux = fn:
   (exreal opens -> Sierpinski) -> prefix_real -> int seq

> fun admisibility (cpt(cfo(x))) =
   (R(0, (admisibility_aux (x) (pref_r(0,[])))));
val admisibility = fn: exreal cptset -> exreal
```

3.8.2 Inverse function calculator

Using admissibility, we can then derive the inverse function calculator.
To construct an inverse function calculator for $f(x)$. First we need to constructed the unit interval as a compact set from **Chapter 3.7**,

```
> val unitint = (cpt(cfo(fn U => compact(getf U))));
val unitint = cpt (cfo fn): exreal cptset
```

Then we construct the compact singleton set which contains the result of $f^{-1}(y)$,

```
> fun singleton_inv f y = cfc_intsec_cpt (cfc (fn x => (notEqual y (f x)))) unitint;
val singleton_inv = fn: (exreal -> exreal) -> exreal -> exreal cptset
```

Then we can construct the inverse function calculator by extracting singleton x.

```
> fun inverse f y = admisibility (singleton_inv f y);
val inverse = fn: (exreal -> exreal) -> exreal -> exreal
```

The inverse function would take an input y and return $f^{-1}(y)$. The inverse function calculator can work on any function from exreal $\rightarrow$ exreal as long as $f^{-1}(y)$ has a unique solution.
A simple application would be implementing divide by 2 function without actually coding the division.

```
> fun div2 y = inverse (fn x=> (rplus x x) ) y;
val div2 = fn: exreal -> exreal
```

```
> fun testdiv2 y n = exr2r (inverse (fn x=> (rplus x x)) (r2exr y n)) n;
val testdiv2 = fn: real -> int -> real

> testdiv2 0.875 4;
val it = 0.4375: real
```

Chapter 4

Evaluation

4.1 The Timer Structure in SML

In Standard ML Basis Library, there exist a Timer structure. The Timer structure provides facilities for measuring the passing of wall clock (real) time and the amount of time the running process has had the CPU (user time), has been active in the OS kernel (system time), and has spent on garbage collection (GC time). Here we mostly use the built in Timer structure for performance analysis. When traditional complexity analysis is achievable, then the Timer can provide supporting evidence to show that derived complexity is correct. When traditional complexity analysis is not achievable, we run a series of test using the Timer structure and reach an estimate for the Time cost. Here we only test the CPU time and neglect the garbage collection time.
A simple testing function can be written as :

```
> fun testadd a b n = exr2r (rplus (r2exr a n) (r2exr b n)) n;
val testadd = fn: real -> real -> int -> real
>
> val testtimer =
let val t = Timer.startCPUTimer()
   in
   (testadd 0.14 0.17 100,
   Time.toString(#usr(Timer.checkCPUTimer(t))) ^ "second")
end;
val testtimer = (0.31, "0.153second"): real * string
>
```

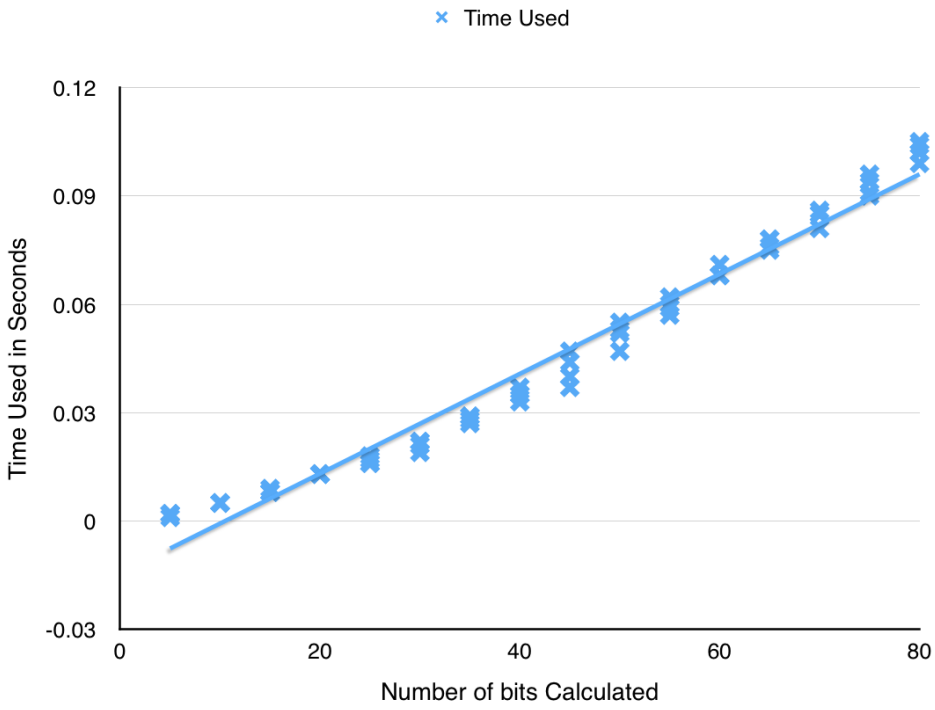
Here the testadd function can be changed to any function that we would like

to perform a test on. The test result returns not only the running time but also the result of the function thus the correctness of the function can also be tested. Then for any function that we want to analyse, we can run this test on different n, compute the first n bit of the result. Then use MatLab to generate graphs to show how time cost increases as n increases.

4.2 Complexity Analysis of Arithmetic on Exreals

4.2.1 Addition

We consider the case that given two arbitrary Exreal number and correctly produce the first n bit of the result Exreal number. First bitwise addition of the two given Exreals up to $n + 2$ bits are required, because we would two extra bits to determine the nth bit. This would take linear time , has complexity $\mathcal{O}(n)$. Then with the added result, the evaluation of the result and covert it to proper signed binary is performed. For each bit, we need to look at 3 bit, current bit and 2 further bits. Extracting 3 bits of added result takes constant time, $\mathcal{O}(1)$ time. Then we test the extracted result against all 39 possible situations and produce the result of the current bit result, this step also requires constant time, $\mathcal{O}(1)$ time, even though it has a rather high factor. Thus the overall complexity for computing the first n bits of the addition result would take $\mathcal{O}(n) * (\mathcal{O}(1) + \mathcal{O}(1)) = \mathcal{O}(n)$ time. Thus the addition would take linear time even tough it has a rather high factor. We use 3 different addition examples and calculate them up to 5, 10, 15, 20... up to 80 bits respectively and the test results below also nicely confirm that addition takes linear time. The results shown as below.



4.2.2 Multiplication

Recall the equation 3.1,

$$\begin{aligned} A * B &= (int1 + frac1) * (int2 + frac2) \\ &= int1 * int2 + int1 * frac2 + int2 * frac1 + frac1 * frac2 \end{aligned} \tag{4.1}$$

Here the three addition in the result is actually two Exreal number addition, since we combine two terms $int1 * int2$ and $frac1 * frac2$ as one Exreal $R(int1 * int2, frac1 * frac2)$.

We evaluate the complexity part by part in the equation. Let's say that $int1$ has i_1 bit in binary and $int2$ has i_2 bit in binary. Thus base on section 3.1.4, calculating $int1 * frac2$ and $int2 * frac1$ to the nth bit is equivalent to calculating i_1 / i_2 Exreal addition to the nth bit. Using the result from 4.2.1, then $int1 * frac2$ and $int2 * frac1$ has a complexity of $\mathcal{O}(i_1 * n)$ and $\mathcal{O}(i_2 * n)$ respectively.

For the $frac1 * frac2$ part, we generate a lazy list of Exreals and add them

together. Each of the Exreals would take linear time to generate, thus the complexity is equivalent to doing $(n+2)$ Exreal addition and it has a complexity of $\mathcal{O}((n+2)*n)*\mathcal{O}(n) = \mathcal{O}(n^3)$. And the integer part multiplication takes $\mathcal{O}(i_1 * i_2)$ time. Thus the overall complexity would take

$$\begin{aligned}(\mathcal{O}(n^3) + \mathcal{O}(i_1 * i_2)) + \mathcal{O}(i_1 * n) + \mathcal{O}(i_2 * n) = \\ \mathcal{O}(n^3) + \mathcal{O}((i_1 + i_2) * n) + \mathcal{O}(i_1 * i_2)).\end{aligned} \tag{4.2}$$

When n is large and i_1, i_2 are small, then the complexity can also be considered as $\mathcal{O}(n^3)$

But in reality the program runs at an extremely slow speed. We use 3 different multiplication examples and calculate them up to 1,2,3 bits respectively. The results shown as below.

Number of Bits Calculated	Time used in Seconds	Test Case
1	0.924	0.5*0.25 =0.125
2	219.455	
3	>7200	
1	0.086	0.62*0.53 = 0.3286
2	28.072	
3	7157.312	
1	1.462	0.95*0.8 = 0.76
2	465.264	
3	>7200	

4.3 Sierpinski Space and complexity

4.3.1 Sierpinski Space Complexity

Due to the semi-decidable nature of the Sierpinski Space datatype, we can make any functions, that takes any arbitrary input and outputs Sierpinski Space, become a linear time function[10]. Consider an arbitrary computable function $f(x)$ such that f maps some x into the Sierpinski Space. Here x is not necessarily an Exreal number, x can be any data types, such as open sets or compact sets etc.

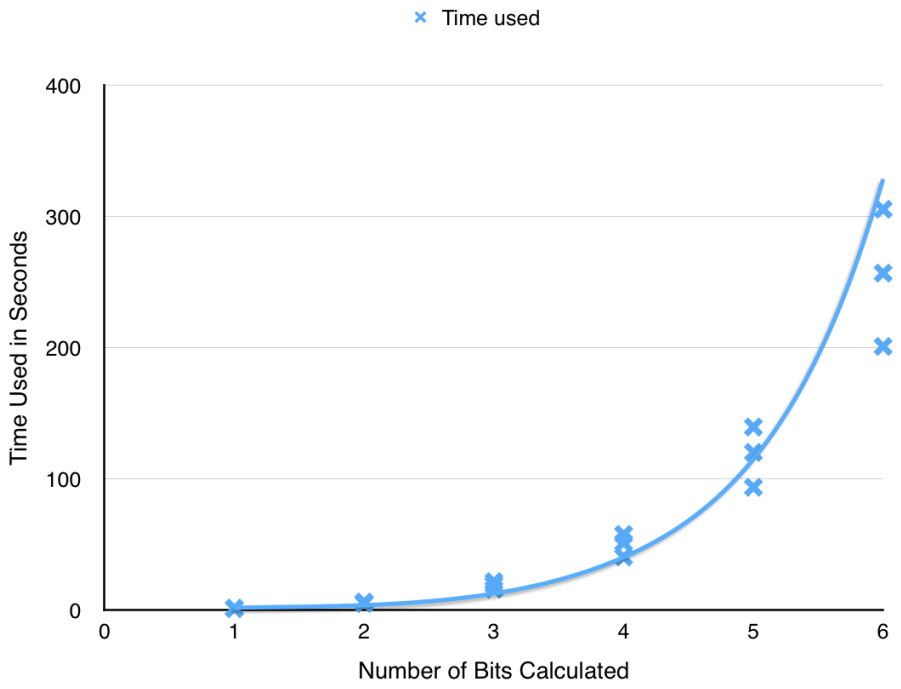
We can construct a linear time computable function $g(x)$ that is equivalent to $f(x)$.
$g(x)$ is defined as :
At any given time t, $g(x)$ is asked to produce a bit of result, then $g(x)$ simulates $f(x)$ up until time t and save the current state for continuation. If $f(x)$ has output any result up until time t, then $g(x)$ returns the same result and ask f about the next bit result. If $f(x)$ outputs nothing then $g(x)$ adds a false to the result.
$g(x)$ and $f(x)$ are equivalent since if $f(x)$ will ever output any true at time t that makes the result maps to $\top$ then so will $g(x)$ at time t. Otherwise $g(x)$ would constantly return false as well. Thus it is meaningless to analyse the complexity of any functions that outputs Sierpinski Space.

4.3.2 Inverse function calculator performance

Since it is impossible to analyse the complexity of Sierpinski Space functions in a meaningful way, we would use Timer structure to estimate the time cost of the inverse function calculator. Here we test the performance of the inverse function of $x + x$ which give us the divide by 2 function in Exreal using Timer structure. We use the test code as shown in **Chapter 3.8.2** and calculate 3 test cases result up to 1,2,3,4,5,6 bits respectively.
The three test cases are $0.5/2 = 0.25$, $0.75/2 = 0.375$, $0.875/2 = 0.4375$.

Number of Bits Calculated	Time used in Seconds	Test Case
1	1.193	0.5/2 =0.25
2	4.881	
3	15.813	
4	40.429	
5	93.489	
6	200.784	
1	1.170	0.75/2 =0.375
2	5.971	
3	19.338	

4	51.158	
5	120.058	
6	256.660	
1	1.249	0.875/2 =0.4375
2	5.964	
3	21.602	
4	57.564	
5	139.377	
6	305.312	



The result indicates that the inverse function calculator for $f(x) = x + x$ has an exponential complexity with an exponent between 2 and 3.

4.4 Functionality

As discussed in introduction, this project has a different approach then other exact real packages available. The inverse function calculator allow us to im-

plement certain functions in Exact Real very easily. Such as the square root function might not be easy to implement directly. But using the inverse function calculator, it can be implemented using the exact code for divide by 2 function, except changing the addition into multiplication, like below,

```
> fun exr_sqrt y = inverse (fn x=> (rmult x x) ) y;
val exr_sqrt = fn: exreal -> exreal
```

The project limitation would be the performance, it would took a very long time before the square root function can return any meaningful result but the package do provide us with an extremely easy way of implementing the function. Thus despite the rather poor performance of the package, the extendability is huge.

Chapter 5

Conclusion

In this project, I have achieved the primary original goals, developing an Exact real package in ML. The project started with researching in current exact real packages and different real representations. Then the signed digit representation is chosen for the project. And the exreal data structure is then implemented with basic arithmetic operation implemented as well. Then using the idea of synthetic topology, topological data structure such as open/closed set and compact set on exact real number is implemented. A very important application of the project, an abstract inverse function calculator is implemented and tested. Then I did a complexity analysis and performance test on the developed datatype.

5.1 Lessons Learned

- Do not get too ambitious when designing the goal of the project. In the original project proposal, one of the goal is to have a performance comparison between my developed exact real package with the current existing iRRAM package. This turn out to be not realistic. The performance of the designed exact real package is no where near iRRAM.

5.2 Future Work

If more time were given, an abstract method of integrating continuous functions on Exreal can be implemented. Integration can be achieved in a synthetic frame-

work is shown in *Computable Stochastic Processes* by Pieter Collins [6]. Also there is room for a performance improvement as well.

Bibliography

[1] The MPFR library. http://www.mpfr.org/.

[2] Poly/ML home page. http://www.polyml.org/.

[3] Standard ML of New Jersey. http://www.smlnj.org/.

[4] Jens Blanck. Exact real arithmetic systems: Results of competition, 2000.

[5] Vasco Brattka, Guido Gherardi, and Alberto Marcone. The Bolzano-Weierstrass Theorem is the jump of Weak König's Lemma. *Annals of Pure and Applied Logic*, 163(6):623–625, 2012. also arXiv:1101.0792.

[6] Pieter Collins. Computable stochastic processes. arXiv:1409.4667, 2014.

[7] Abbas Edalat and Peter John Potts. A new representation for exact real number. 1998.

[8] Martin Escardó. Synthetic topology of datatypes and classical spaces. *Electronic Notes in Theoretical Computer Science*, 87, 2004.

[9] John K. Hunter. An introduction to real analysis, 2013.

[10] Akitoshi Kawamura and Arno Pauly. Function spaces for second-order polynomial time. arXiv 1401.2861, 2014.

[11] Norbert Th. Müller. irram - exact arithmetic in C++. http://irram.uni-trier.de/.

[12] Arno Pauly. On the topological aspects of the theory of represented spaces. http://arxiv.org/abs/1204.3763, 2012.

[13] Arno Pauly and Martin Ziegler. Relative computability and uniform continuity of relations. *Journal of Logic and Analysis*, 5, 2013.

[14] Klaus Weihrauch. *Computable Analysis*. Springer-Verlag, 2000.

Appendix A

Key Source Code

A.1 Exreal Datatype

```
datatype 'a seq = Cons of 'a * (unit -> 'a seq);
datatype exreal = R of int * int seq;
fun head (Cons(x,_)) = x;
fun tail (Cons(_,xf)) =xf();

fun zeros () = Cons(0,fn ()=>zeros ());
fun ones () = Cons(1,fn ()=>ones ());
fun mones () = Cons(~1,fn ()=>mones ());



(* Conversion between real and exact real *)
fun ll2r _ 0 _ = 0.0
| ll2r (Cons(x,xq)) n i = (Real.fromInt(x)/(Math.pow(2.0,i))) + (ll2r (xq()) (n-1) (i+1.0));

fun exr2r (R(i,f)) n = Real.fromInt(i)+(ll2r f n 1.0);

fun r2ll x 0 i = zeros()
|  r2ll x n i = if Real.signBit(x - 1.0/Math.pow(2.0,i)) then Cons(0,fn ()=> r2ll x (n-1) (i+1.0))
else Cons(1, fn ()=> r2ll (x- 1.0/Math.pow(2.0,i)) (n-1) (i+1.0));

fun r2exr r n = (R(floor r, r2ll (r - Real.fromInt(floor r)) (n) (1.0)));

(* Addition *)
fun take (0, xq) = []
  | take (n, Cons(x,xf)) = x :: take (n-1, xf());



fun nlength [] = 0
  | nlength (x::xs) = 1 + nlength xs;
```

```
fun frac2list_1 (0, xq) = []
  | frac2list_1 (n, Cons(x,xf)) = x :: frac2list_1 (n-1, xf());

fun frac2list x = frac2list_1(~1,x);


fun eval_result x = if (take(2,x) =[1,2] orelse take(2,x)=[~1,~2]) then Cons(0, fn()=>eval_result(Cons(0,fn ()=>tail(tail(x)))
else if (take(3,x)=[1,1,1] orelse take(3,x)=[1,1,2]) then eval_result(Cons(0, fn()=>Cons(~1,fn ()=>tail(tail(x)))))
else if (take(3,x)=[~1,~1,~1] orelse take(3,x)=[~1,~1,~2]) then eval_result(Cons(0, fn()=>Cons(1,fn ()=>tail(tail(x)))))
else if (take(3,x)=[1,1,0] orelse (take(3,x)=[1,1,~1] orelse take(3,x)=[1,1,~2])) then Cons(1, fn()=>eval_result(Cons(1,fn ()=
else if (take(3,x)=[~1,~1,0] orelse (take(3,x)=[~1,~1,1] orelse take(3,x)=[~1,~1,2])) then Cons(~1, fn()=>eval_result(Cons(~1,
else if (take(3,x)=[1,~1,~1] orelse take(3,x)=[1,~1,~2]) then Cons(0, fn()=>eval_result(Cons(1,fn ()=>tail(tail(x)))))
else if (take(3,x)=[1,~1,0] orelse (take(3,x)=[1,~1,1] orelse take(3,x)=[1,~1,2])) then Cons(1, fn()=>eval_result (Cons(~1,fn
else if take(2,x) = [1,~2] then Cons(0, fn()=>eval_result (Cons(0,fn ()=>tail(tail(x)))))
else if take(2,x) = [1,0] then Cons(1, fn()=>eval_result (Cons(0,fn ()=>tail(tail(x)))))
else if take(2,x) = [0,2] then Cons(1, fn()=>eval_result(Cons(0,fn ()=>tail(tail(x)))))
else if take(2,x) = [0,~2] then Cons(~1, fn()=>eval_result(Cons(0,fn ()=>tail(tail(x)))))
else if take(2,x) = [0,0] then Cons(0, fn()=>eval_result(Cons(0,fn ()=>tail(tail(x)))))
else if (take(3,x) = [0,1,2] orelse take(3,x)=[0,1,1]) then Cons(1, fn()=>eval_result(Cons(~1,fn ()=>tail(tail(x)))))
else if (take(3,x) = [0,1,~2] orelse (take(3,x)=[0,1,~1] orelse take(3,x)=[0,1,0])) then Cons(0, fn()=>eval_result(Cons(1,fn (
else if (take(3,x) = [0,~1,~2] orelse take(3,x)=[0,~1,~1]) then Cons(~1, fn()=>eval_result(Cons(1,fn ()=>tail(tail(x)))))
else if (take(3,x) = [0,~1,2] orelse (take(3,x)=[0,~1,1] orelse take(3,x)=[0,~1,0])) then Cons(0, fn()=>eval_result(Cons(~1,fn
else if (take(3,x)=[~1,1,1] orelse take(3,x)=[~1,1,2]) then Cons(0, fn()=>eval_result(Cons(~1,fn ()=>tail(tail(x)))))
else if (take(3,x)=[~1,1,0] orelse (take(3,x)=[~1,1,~1] orelse take(3,x)=[~1,1,~2])) then Cons(0, fn()=>eval_result (Cons(~1,f
else if take(2,x) = [~1,0] then Cons(~1, fn()=>eval_result (Cons(0,fn ()=>tail(tail(x)))))
else if take(2,x) = [~1,2] then Cons(0, fn()=>eval_result (Cons(0,fn ()=>tail(tail(x))))) else eval_result(Cons(0,fn()=>tail(x


fun first_var (x,y) = x;
fun sec_var (x,y) = y;
fun frac_sum1 (Cons(x,xq),Cons(y,yq)) = Cons(x+y,fn ()=>(frac_sum1(xq(),yq())));
fun rplus (R(x1,y1)) (R(x2,y2)) = let val temp = frac_sum1(y1,y2) in
if head(temp) = 2 orelse (take(2,temp)=[1,2] orelse (take(3,temp)=[1,1,1] orelse take(3,temp)=[1,1,2])) then (R(x1+x2+1,eval_r
else if head(temp) = ~2 orelse (take(2,temp)=[~1,~2] orelse (take(3,temp)=[~1,~1,~1] orelse take(3,temp)=[~1,~1,~2])) then (R(
else (R(x1+x2,eval_result(temp)))
end;

fun multplus ([x]) = x
| multplus (x::xs) = rplus (x) (multplus xs);




fun frac_sum x y = let val temp = frac_sum1(x,y) in
if head(temp) = 2 orelse (take(2,temp)=[1,2] orelse (take(3,temp)=[1,1,1] orelse take(3,temp)=[1,1,2])) then (1,eval_result (t
else if head(temp) = ~2 orelse (take(2,temp)=[~1,~2] orelse (take(3,temp)=[~1,~1,~1] orelse take(3,temp)=[~1,~1,~2])) then (~1
else (0,eval_result(temp))
```

```
end;


fun list_frac_plus ([x]) = x
| list_frac_plus (x::xs) = sec_var( frac_sum (x) (list_frac_plus xs));







(* multiplication *)
(* the fraction multiply with fraction is curently slow and cause the slow of multiplication *)

fun revApp ([], ys) = ys
| revApp (x::xs, ys) = revApp (xs, x::ys);

fun tobinary n = let
  fun tb' 0   acc = acc
    | tb' num acc = tb' (num div 2) (num mod 2 :: acc)
in
  tb' n []
end;

fun todecimal1 [] acc = 0
| todecimal1 (x::xs) acc = x*acc + (todecimal1 xs acc*2);

fun todecimal x = todecimal1 (revApp (x,[])) 1;


fun split ([], _) = (0,[0])
| split ([x], a) = (x,a)
| split ((x::xs), y) = split(xs, y@[x]);


fun minus_f (Cons(x,xf)) = Cons(~1*x, fn ()=> (minus_f (xf())));

fun r_minus (R(i,f)) = (R(~1*i,(minus_f (f))));

fun get 1 x =head(x)
| get k x = get (k-1) (tail(x));


fun addz 0 x 0 = zeros()
| addz 0 x 1 = x
| addz 0 x (~1) = minus_f(x)
| addz n x z = Cons(0,fn ()=> addz (n-1) x z);

fun addele x (Cons(0,yq)) n = Cons((addz n x 0), fn ()=> (addele x (yq()) (n+1)))
| addele x (Cons(1,yq)) n = Cons((addz n x 1), fn ()=> (addele x (yq()) (n+1)))
```

```
| addele x (Cons(~1,yq)) n = Cons((addz n x ~1), fn ()=> (addele x (yq()) (n+1)));


fun mult_frac_aux1 k x =
let val temp = list_frac_plus(take(3,x)) in
(get k temp, Cons(temp,fn()=> tail(tail(tail(x)))))
end;


fun mult_frac_aux2 x k =
let val temp = mult_frac_aux1 k x in
Cons(first_var(temp), fn ()=> (mult_frac_aux2 (sec_var(temp)) (k+1)))
end;

fun mult_frac x y = mult_frac_aux2 (addele x y 1) 1;

fun tailn 0 x = x
| tailn n x = tailn (n-1) (tail(x));

fun list_exr ([]) (y) (n) (res) = res
| list_exr (1::x) y n res = list_exr x y (n+1) ((R(todecimal(take(n,y)),(tailn n y)))::res)
| list_exr (0::x) y n res = list_exr x y (n+1) res;


fun mult_frac_int x y = let val temp = revApp(tobinary x,[]) in
multplus (list_exr temp y 0 [])
end;

fun rmult (R(i1,f1)) (R(i2,f2)) = rplus (R(i1*i2, mult_frac (f1) (f2))) (rplus (mult_frac_int i1 (f2)) (mult_frac_int i2 (f1))
```

A.2 Sierpinski Space

```
datatype Sierpinski = Sierp of bool seq;
fun c_true () = Cons(true, fn ()=>c_true());
fun c_false () = Cons(false, fn ()=>c_false());

fun sierp_extract_seq (Sierp(x)) = x;

fun interleave (Cons(x,xf), yq) = Cons(x, fn()=> interleave(yq, xf()));

fun sierp_or (Sierp(a)) (Sierp(b)) = Sierp(interleave(a,b));

fun sierp_and (Sierp(Cons(x,xf))) (Sierp(Cons(y,yf))) =
if x=true then (Sierp(Cons(y,yf)))
else (Sierp(Cons(false,fn ()=> sierp_extract_seq (sierp_and (Sierp(Cons(y,yf))) (Sierp(xf()))))));


fun sierp_and_ls [] = (Sierp(c_true()))
```

```
| sierp_and_ls (s::ss) = (Sierp(Cons(false, fn ()=> (sierp_extract_seq(sierp_and s ((sierp_and_ls ss)))))));


fun sierp_is_top (Sierp(Cons(x,xq))) =
if x = true
then true
else sierp_is_top (Sierp(xq()));

fun append ([],ys) = ys
| append (x::xs,ys) = x::append(xs,ys);

fun multi_sierp_or (Cons((Sierp(Cons(x,xq))),sf)) =
if x = true
then (Sierp(c_true()))
else (Sierp(Cons(false, fn()=>interleave (sierp_extract_seq(multi_sierp_or (sf())),(xq())) )));

fun list_sierp_or ((Sierp(Cons(x,xq)))::xs) = Sierp(Cons(x,fn ()=> (sierp_extract_seq(list_sierp_or (xs@[(Sierp(xq()))])
| list_sierp_or [] = Sierp(c_false());
```

A.3 Open/Closed Set

```
datatype 'a opens = cfo of ('a -> Sierpinski);
datatype 'a closed = cfc of ('a -> Sierpinski);

fun opens_union (cfo(x)) (cfo(y)) = cfo (fn a => (sierp_or (x a) (y a)));

fun opens_intersection (cfo(x)) (cfo(y)) = (cfo(fn a => sierp_and (x a) (y a)));



(* Hausdoff *)
fun check1m1 (Cons(x,xq),Cons(y,yq)) =
if x=1
then if y= ~1
then Cons(false, fn ()=> check1m1(xq(),yq()))
else c_true()
else c_true();

fun check0m1 (Cons(x,xq),Cons(y,yq)) =
if x=0
then if y= ~1
then Cons(false, fn ()=> check0m1(xq(),yq()))
else c_true()
else c_true();


fun checkm1m1 (Cons(x,xq),Cons(y,yq)) =
if x= ~1
```

```
then if y= ~1
then Cons(false, fn ()=> checkm1m1(xq(),yq()))
else c_true()
else c_true();


fun check10 (Cons(x,xq),Cons(y,yq)) =
if x=1
then if y=0
then Cons(false, fn ()=> check10(xq(),yq()))
else c_true()
else c_true();
fun check00 (Cons(x,xq),Cons(y,yq)) =
if x=0
then if y=0
then Cons(false, fn ()=> check00(xq(),yq()))
else c_true()
else c_true();
fun checkm10 (Cons(x,xq),Cons(y,yq)) =
if x= ~1
then if y=0
then Cons(false, fn ()=> check10(xq(),yq()))
else c_true()
else c_true();

fun check11 (Cons(x,xq),Cons(y,yq)) =
if x=1
then if y= 1
then Cons(false, fn ()=> check11(xq(),yq()))
else c_true()
else c_true();

fun check01 (Cons(x,xq),Cons(y,yq)) =
if x=0
then if y= 1
then Cons(false, fn ()=> check01(xq(),yq()))
else c_true()
else c_true();

fun checkm11 (Cons(x,xq),Cons(y,yq)) =
if x= ~1
then if y= 1
then Cons (false, fn ()=>checkm11(xq(),yq()))
else c_true()
else c_true();


fun notEqual_seq (Cons(x,xq)) (Cons(y,yq)) =
if x-y = 1 then Cons(false, fn ()=> notEqual_seq (xq()) (Cons(head(yq())-2, fn ()=> tail(yq()) )))
else if y-x = 1 then Cons(false, fn ()=> notEqual_seq (yq()) (Cons(head(xq())-2, fn ()=> tail(xq()) )))
```

```
else if x=y then Cons(false, fn ()=> notEqual_seq (xq()) (yq()))
else if x-y=2 then checkm11(xq(),yq())
else if y-x=2 then check1m1(xq(),yq())
else c_true();


fun notEqual (R(ix,fx)) (R(iy,fy)) =
if ix = iy then Sierp(notEqual_seq (fx) (fy))
else if ix-iy=2 then Sierp(checkm11(fx,fy))
else if iy-ix=2 then Sierp(check1m1(fx,fy))
else if ix-iy=1 then Sierp(notEqual_seq (fx) (Cons(head(fy)-2, fn ()=> tail(fy))))
else if iy-ix=1 then Sierp(notEqual_seq (fy) (Cons(head(fx)-2, fn ()=> tail(fx))))
else Sierp(c_true());

fun checkm1 (Cons(x,xq)) = Cons(not(x=(~1)), fn ()=> checkm1 (xq()));

fun check1 (Cons(x,xq)) = Cons(not(x=1), fn ()=> check1 (xq()));


fun isPositive (R(0,Cons(x,xq))) =
if x=0 then
(Sierp(Cons(false,fn ()=> (sierp_extract_seq (isPositive (R(0,xq())))))))
else
if x=1 then
(Sierp(checkm1 (xq())))
else (Sierp(c_false()))
| isPositive (R(1,f)) = (Sierp(checkm1 (f)))
| isPositive (R(i,f)) =
if i>1 then
(*mones*)
(Sierp(c_true()))
else (Sierp(c_false()));
```

A.4 Compactness

```
datatype 'a cptset = cpt of (('a opens) opens);

fun sierp_c (Sierp(Cons(true,xq))) = (Sierp(Cons(false,fn ()=>sierp_extract_seq (sierp_c(Sierp(xq()))))))
| sierp_c (Sierp(Cons(false,xq))) = (Sierp(Cons(true,fn ()=>sierp_extract_seq (sierp_c(Sierp(xq()))))));

fun complementc (cfc(x)) = (cfo(x));

fun complemento (cfo(x)) = (cfc(x));

fun cfc_intsec_cpt (f) (cpt(cfo(x : (('a opens)->Sierpinski)))) = (cpt(cfo(fn a => (x (opens_union (complementc(f)) a)))

exception Do_not_Look;
```

```
fun handle_aux x = if head(sierp_extract_seq x)=true then (Sierp(c_true()))
else ((Sierp(Cons(false,fn ()=> (sierp_extract_seq((handle_aux (Sierp(tail(sierp_extract_seq(x)))))) handle Do_not_Look => (c_

fun auxf f = (fn x => (f x handle Do_not_Look=> (Sierp(c_false()))));

fun multiappend [] = []
| multiappend (x::xs) = x@(multiappend xs);

fun disguise_aux (x::xs) = Cons(x,fn ()=> disguise_aux xs)
| disguise_aux [] = raise Do_not_Look;

fun disguise (pref_r(x,xs)) = (R(x,(disguise_aux(xs))));

fun generate_dis (pref_r(x,xs)) = [disguise(pref_r(x,xs@[1])), disguise(pref_r(x,xs@[0])), disguise(pref_r(x,xs@[~1]))];
fun generate_pref (pref_r(x,xs)) = [(pref_r(x,xs@[1])),(pref_r(x,xs@[0])),(pref_r(x,xs@[~1]))];


fun last_ele [] = 0
| last_ele (x::xs) = if xs=[] then x else last_ele xs;
fun generate_pref1 (pref_r(x,xs)) = if (last_ele xs)=1 then [(pref_r(x,xs@[1])),(pref_r(x,xs@[0]))]
else if (last_ele xs)= (~1) then [(pref_r(x,xs@[~1])),(pref_r(x,xs@[0]))]
else [(pref_r(x,xs@[1])),(pref_r(x,xs@[0])),(pref_r(x,xs@[~1]))];


fun generate_pref2 (pref_r(x,xs)) =
let val temp = (last_ele xs) in (if temp=1 then [(pref_r(x,xs@[0]))]
else if temp= (~1) then [(pref_r(x,xs@[0]))]
else [(pref_r(x,xs@[1])),(pref_r(x,xs@[0])),(pref_r(x,xs@[~1]))])
end;




fun dis_seq (x) = Cons((map disguise (x)) , fn ()=> dis_seq (multiappend (map generate_pref (x))));

val pr = dis_seq([(pref_r(0,[0])),(pref_r(0,[1])),(pref_r(0,[~1]))]);

(*fun aux_f f = (fn x =>(handle_aux (f x)));*)

fun compact_aux f x = Cons((handle_aux (sierp_and_ls (map (auxf f) (head(x))))), fn () => (compact_aux f (tail(x))));

fun compact f x = multi_sierp_or (compact_aux f x);

fun admisibility (cpt(cfo(x))) =

let val temp = (findindex_i (mapq x (table(1))) [] 1) in

(R(temp, (admisibility_aux (x) (pref_r(temp,[])))))
end;
```

```
fun headls (x::xs) =x;
fun taills [] =[]
| taills (x::xs) = xs;


(*
fun compact1 f [] _ = (print("base case");(Sierp(c_true())))
| compact1 f (x::xs) x_result =
(if (head(sierp_extract_seq(x_result()))) = true
then (print_prefixreal(x);(if (xs=[]) then (Sierp(c_true())) else (compact1 f xs (fn () => (f (disguise (headls(xs))))))
else (Sierp(Cons(false, fn ()=> (sierp_extract_seq(compact1 f (x::xs) (fn () => (Sierp((tail(sierp_extract_seq(x_result(
handle Do_not_Look => (print("");(Sierp(Cons(false, fn ()=> (sierp_extract_seq(compact1 f (xs@(generate_pref x)) (fn () =
*)

fun print_list [] = print("   ")
| print_list (x::xs) = (print(Int.toString(x));print_list(xs));
fun print_prefixreal (pref_r(i,f)) = print_list(f);




fun compact1 f [] _ = (Sierp(c_true()))
| compact1 f (x::xs) x_result =
(if (head(sierp_extract_seq(x_result()))) = true
then if (xs=[]) then (Sierp(c_true())) else (compact1 f xs (fn () => (f (disguise (headls(xs))))))
else (Sierp(Cons(false, fn ()=> (sierp_extract_seq(compact1 f (x::xs) (fn () => (Sierp((tail(sierp_extract_seq(x_result(
handle Do_not_Look => (Sierp(Cons(false, fn ()=> (sierp_extract_seq(compact1 f (xs@(generate_pref x)) (fn () => (f (disg




fun compact2 f [] _ = (Sierp(c_true()))
| compact2 f (x::xs) x_result =
(if (head(sierp_extract_seq(x_result()))) = true
then if (xs=[]) then (Sierp(c_true())) else (compact1 f xs (fn () => (f (disguise (headls(xs))))))
else (Sierp(Cons(false, fn ()=> (sierp_extract_seq(compact1 f (x::xs) (fn () => (Sierp((tail(sierp_extract_seq(x_result(
handle Do_not_Look => (Sierp(Cons(false, fn ()=> (sierp_extract_seq(compact1 f (xs@(generate_pref1 x)) (fn () => (f (dis


fun compact3 f [] _ = (Sierp(c_true()))
| compact3 f (x::xs) x_result =
(if (head(sierp_extract_seq(x_result()))) = true
then if (xs=[]) then (Sierp(c_true())) else (compact1 f xs (fn () => (f (disguise (headls(xs))))))
else (Sierp(Cons(false, fn ()=> (sierp_extract_seq(compact1 f (x::xs) (fn () => (Sierp((tail(sierp_extract_seq(x_result(
handle Do_not_Look => (Sierp(Cons(false, fn ()=> (sierp_extract_seq(compact1 f (xs@(generate_pref2 x)) (fn () => (f (dis
```

```
fun compact_new f = compact1 f [(pref_r(0,[]))] (fn () => (f (disguise (pref_r(0,[])))));
fun compact_new1 f = compact2 f [(pref_r(0,[]))] (fn () => (f (disguise (pref_r(0,[])))));
fun compact_new2 f = compact3 f [(pref_r(0,[]))] (fn () => (f (disguise (pref_r(0,[])))));
```

A.5 Admissibility

```
datatype prefix_real = pref_r of int * int list;
datatype Interval = I of exreal * exreal;

fun makeinterval (I(l,r)) = cfo(fn z =>
(let val a = (isPositive ((rplus (z) (r_minus(l))))) in
let val b = (isPositive ((rplus (r) (r_minus(z))))) in
(sierp_and (a) (b))
end
end)
);

fun get(0,xq) = []
|  get(n,Cons(x,xf)) = x :: get(n-1,xf());


fun exr2pref (R(i,f)) n = pref_r(i,get(n,f));

fun list2lazym1 [] = mones()
| list2lazym1 (x::xs) = Cons(x, fn()=>(list2lazym1 xs) );

fun list2lazy1 [] = ones()
| list2lazy1 (x::xs) = Cons(x, fn()=>(list2lazy1 xs) );

fun pref2exrm1 (pref_r(i,f)) = (R(i,(list2lazym1 f)));
fun pref2exr1 (pref_r(i,f)) = (R(i,(list2lazy1 f)));

fun make_pref_interval x = makeinterval (I((pref2exrm1 x),(pref2exr1 x)));

(*map interval *)
fun table(n) = Cons(makeinterval( I((R(n-1,zeros())),(R(n+1,zeros())))), fn ()=> table(n+1));

fun mapq f (Cons(x,xq)) = (Cons(f x, fn ()=> (mapq f (xq()))));

(*find index for interger part*)
fun map f [] = []
| map f (x::xs) = (f x) :: (map f xs);

fun take_head (Sierp(Cons(x,xf))) = x;
```

```
fun checktrue [] n = ~1
| checktrue (x::xs) n = if x=true then n else checktrue xs (n+1);

fun index_list x = checktrue (map take_head x) 0;

(* cl stands for current list, ll stands for current list length, ci stands for current index, which is what we wanted. *
fun findindex_i (Cons(Sierp(Cons(x,xf)), sf)) cl ci =
if x=true then ci else
if index_list cl = ~1 then
findindex_i (sf()) (cl@[Sierp(xf())]) (ci+1)
else index_list cl;

fun findindex_f (Sierp(Cons(x,xf))) (Sierp(Cons(y,yf))) (Sierp(Cons(z,zf)))
= if x=true then 1 else if y=true then 0 else if z=true then ~1 else findindex_f (Sierp(xf())) (Sierp(yf())) (Sierp(zf()

fun admisibility_aux x (pref_r(i,f)) =
let
val temp = (findindex_f (x (make_pref_interval((pref_r(i,f@[1]))))) (x (make_pref_interval((pref_r(i,f@[0]))))) (x (make_
in
Cons(temp, fn ()=> (admisibility_aux x (pref_r(i,f@[temp]))))
end;

fun singleton x = (cpt(cfo (fn u=> ((getf u) x))));


fun admisibility1 (cpt(cfo(x))) = (R(0, (admisibility_aux (x) (pref_r(0,[])))));

fun testcase (cpt(cfo(x))) = x (make_pref_interval((pref_r(0,[1]))));
```

Appendix B

Project Proposal

Shi Shu

Fitzwilliam College

ss2099

Computer Science Part II Project Proposal

An Exact Real Package for ML

24th October 2013

Project Originator: Dr. Arno Pauly

Supervisors: Dr. Arno Pauly

Director of Studies: Dr. Robert Harle

Overseers: Dr. Richard Gibbens / Prof. Larry Paulson

1.Introduction

Many applications of computers require the use of real arithmetic. Unfortunately, there are a number of significant problems associated with performing real arithmetic using floating point approximation, the method most commonly used by programmers, and which forms part of most computer languages and the instruction set of many modern CPUs. These problems stem from the fact that only a finite set of reals are represented exactly using this approach, and rounding occurs after each floating point operation. They can result in complete loss of accuracy in even relatively simple computations.

Originally, the idea of the project came from a paper "A new introduction to the theory of represented space" by Dr. Arno Pauly (See *http://arxiv.org/pdf/1204.3763v2.pdf*). Following from the paper, a synthetic and functional approach to solve differential equation can be derived. Hence implementing the algorithm in a functional language would be straight forward. Also the algorithm would require the use of exact real numbers, continuous functions, open sets and so on. Thus ML can be a good choice of programming language here, which then directly lead to core part of the project, developing an exact real package for ML.

There are many existing exact real packages in different languages using different implementations. Among them, the iRRAM (See *http://irram.uni-trier.de/*) is a very efficient C++ package for error-free real arithmetic based on the concept of a Real-RAM. Its capabilities range from ordinary arithmetic over trigonometric functions to linear algebra even with sparse matrices.

The project mainly focuses on building an exact real package, evaluating its performance and comparing it with other exact real implementations (iRRAM in C++). Then completing the algorithm of the differential equation solver using the package will be an extension to the project. As a simpler example, it can be proved that the inverse of a continuous injection from a compact and admissible space to a Hausdorff space can be computed. Hence implementing an inverse function calculator using the package can be another extension.

In terms of representing real number, "Computable Analysis" by Professor Klaus Weirauch discusses the problem from a theoretical prospective in detail and can be used as a starting point. And also classical complexity theory would not work with infinite lists (and subsequently, real numbers etc). "Complexity theory for operators in analysis" by Akitoshi Kawamura and Stephen Cook (See *https://www.cs.toronto.edu/~sacook/homepage/stoc.2010.pdf*) describes how complexity theory can be applied to this setting.

2. Resource Required

The core part of the project will be written in Standard ML.

The iRRAM package of C++ will be needed to compare with the ML implementation.

Both my own laptop and desktop in Intel Lab will be used for programming.

Backup plans will include Dropbox and Google Drive.

GitHub will be used for version control.

LaTex will be used for dissertation writing.

3. Starting Point

The project will be undertaken with the background knowledge acquired from the following courses in previous years :

Programming in ML from Part IA course.

Computation Theory from Part IB course.

4. The Substance and Structure of the Project

The objective of the project is to build an exact real arithmetic package for ML and compare the implemented package with the other existing exact arithmetic packages available for other languages(i.e. iRRAM). A competition between existing exact real systems is conducted by Jens Blanck (See *http://www-compsci.swan.ac.uk/~csjens/pdf/20640389.pdf*). A similar testing method might be used here.

Thus the project contains 2 main parts and 2 extensions.

1. Implement an exact real arithmetic package for ML.
   A. Find a suitable representation of real numbers in ML.
   B. Develop algorithms for all operations under such representation.
   C. Implement such data structures and operations in ML.
   D. Compute the complexity of all the operations both in terms of time and memory for this implementation.
2. Compare the implemented package with other packages in terms of time and memory efficiency.
3. (Extension) Implement an inverse function calculator using the package.
4. (Extension) An algorithm of solving differential equations can be derived from the theory of represented space. Implement the algorithm using the exact real packages built in ML to construct a DE solver.

5. Success Criteria

For the project to be considered successful the following items must be completed.

1. A working exact real package in ML should be delivered.

2. The performance of the implementation should be evaluated.

Further to the core part of the project, the extension can be regarded as successful if a working inverse function calculator is implemented and the differential equation solver is able to solve DE correctly.

6. Timetable and Milestones

The timetable below is divided into 2-4 week slots.

Week 1 and 2 (Oct. 24th to Nov. 6th)

1. Research possible ways of representing exact reals. Read related material in "Computable Analysis".

2. Understand some of the current implementations of exact real arithmetic in other languages.

3. Learn LaTex to prepare for dissertation writing.

Week 3, 4, 5 and 6 (Nov. 7th to Dec. 4th)

1. Find the suitable representation of real numbers in ML.

2. Implement the real numbers representation in ML.

Week 7, 8 and 9 (Dec. 5th to Dec. 25th)

1. List all the operations that the real number should be able to support.

2. Design algorithms for every operation.

Week 10 and 11 (Dec. 26th to Jan. 8th)

1. Implement all the exact real operations.

2. Create test data to test the package.

Week 12, 13 and 14 (Jan. 9th to Jan. 29th)

1. Debug the system.

2. Evaluate the complexity of all the operations under such implementation.

3. Write the progress report.

4. Prepare the presentation.

Week 15 and 16 (Jan. 30th to Feb. 12th)

1. Start writing the implementation part of the dissertation.
2. Test the implemented package against iRRAM.
3. Analyse the reasoning behind the possible performance difference.

Week 17 and 18 (Feb.13th to Feb. 26th)

1. Implement the inverse function solver using the package.
2. Start writing the evaluation part of the dissertation.

Week 19, 20 and 21 (Feb. 27th to Mar. 19th)

1. Implement the abstract algorithm of the DE solver in ML.
2. Add Extension to the draft dissertation.

Week 22, 23 and 24 (Mar. 20th to Apr. 9th)

1. Finish the draft dissertation.
2. Make the implemented package support exception handling.

Week 25 and 26 (Apr. 10th to Apr. 23rd)

1. Polish dissertation and formatting dissertation correctly.
2. Add reference and code to dissertation.

Week 28 and 29 (Apr. 24th to May 7th)

1. Kept in case any of the previous steps takes longer than expected.

Week 30 (May 8th to May 15th)

1. Finalise the dissertation.
2. Print and hand in the dissertation.