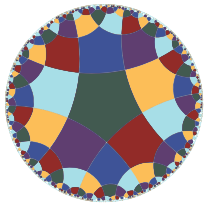
ExactLib

Imperative exact real arithmetic in functional-style C++



Timothy Spratt
tps07@ic.ac.uk

Supervisor: Dirk Pattinson
Second marker: Steffen van Bakel

BEng Computing
Individual Project Report

DEPARTMENT OF COMPUTING
IMPERIAL COLLEGE LONDON

Typeset in LaTeX using Adobe Minion Pro, Computer Modern and Inconsolata.

Abstract

The accurate representation and manipulation of real numbers is crucial to virtually all applications of computing. Real number computation in modern computers is typically performed using floating-point arithmetic that can sometimes produce wildly erroneous results. An alternative approach is to use *exact real arithmetic*, in which results are guaranteed correct to any arbitrary user-specified precision.

We present an exact representation of computable real numbers that admits tractable proofs of correctness and is conducive to an efficient imperative implementation. Using this representation, we prove the correctness of the basic operations and elementary functions one would expect an arithmetic system to support.

Current exact real arithmetic libraries for imperative languages do not integrate well with their languages and are far less elegant than their functional language counterparts. We have developed an exact real arithmetic library, *ExactLib*, that integrates seamlessly into the C++ language. We have used template metaprogramming and functors to introduce a way for users to write code in an expressive functional style to perform exact real number computation. In this way, we approach a similar level of elegance as a functional language implementation whilst still benefiting from the runtime efficiency of an imperative C++ implementation.

In comparison to existing imperative and functional libraries, ExactLib exhibits good runtime performance, yet is backed by proofs of its correctness. By writing a modern library in a popular imperative language we bring exact real arithmetic to a more mainstream programming audience.

Acknowledgements

First and foremost, I would like to thank Dirk Pattinson for supervising my project. He found time for me whenever I needed it, providing invaluable advice and feedback, and for that I am very grateful. Thanks go to my second marker Steffen van Bakel, in particular for his words of motivation on a dreary February afternoon that kicked me into gear when it was most needed. I would also like to thank my two tutors, Susan Eisenbach and Peter Harrison, who have provided ample support throughout my undergraduate career.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivation

The real numbers are inspired by our experience of time and space—or *reality*. They play a crucial role in daily life in the practical as well as philosophical sense. Classical physics describes reality in terms of real numbers: physical space is conceived as $\mathbb{R}^3$, for example, and the time continuum as $\mathbb{R}$. Indeed we model the world around us almost exclusively using the reals, measuring mass, energy, temperature, velocity with values in $\mathbb{R}$, and describing physical processes such as the orbits of planets and radioactive decay via constructs from real analysis, like differential equations.

The efficient representation and manipulation of real numbers is clearly central to virtually all applications of computing in which we hope to describe an aspect of the real world. The irrational nature of real numbers presents computer scientists with a serious problem: how can an uncountable infinity of numbers that do not all have finite descriptions be represented on a computer? It is perhaps disturbing to learn that the majority of computer systems fail miserably to meet this challenge. They use a representation consisting of only finitely many rational numbers, opting for a practical solution through a compromise between breadth and precision of accuracy and economy of efficiency for the representation.

One such solution, floating-point arithmetic, has become the *de facto* standard for the representation of real numbers in modern computers, offering a sufficient compromise for most numerical applications. However, there are plenty of cases [24] wherein floating-point representations lead to unacceptable errors in calculations, even when we use a high level of precision, such as 64-bit[1] double precision. We find an accumulation of small rounding errors can cause a large deviation from an actual exact value. A well-known example is the following iteration, which starts at the golden ratio, φ:

$$\gamma_0 = \frac{1+\sqrt{5}}{2} \quad \text{and} \quad \gamma_{n+1} = \frac{1}{\gamma_n - 1} \quad \text{for } n \geq 0.$$

It is trivial using induction to show that $\gamma_0 = \gamma_1 = \gamma_2 = \ldots$ such that (γ_n) converges to φ. However, when we implement the iteration in C++ using the program listed

[1] It is perhaps sobering to note that even with 50 bits we can express the distance from the Earth to the Moon with an error less than the thickness of a bacterium. [24]

in Appendix A.1, we see the calculations of γ_i diverge quickly from their exact value. After only 8 iterations, double precision floating-point arithmetic delivers inaccurate results, with the sequence seemingly converging to $1 - \varphi$, a negative number. Were we to implement this iteration with a higher precision floating-point representation, we would find much the same problem, albeit with a slight delay before similar incorrect behaviour is exhibited. Whilst our example may seem rather synthetic, there are certainly practical cases in which floating-point computation leads to a wildly erroneous results. On such example can be found in [24], wherein a sequence of deposits and withdrawals of certain values on a bank account results in a hugely incorrect balance when simulated using floating-point arithmetic.

The question arises whether or not we can develop a computer representation that does not suffer this fate—whether an implementation of *all* the real numbers is possible. The problem, of course, is that we cannot have a *finite description* of every real. However, it turns out one way to represent the inherent infiniteness of real numbers is by specifying them step-wise, as infinite data types, so that we can compute arbitrarily close approximations to a real number's exact value. This approach of lazy evaluation lends itself well to an implementation in a functional programming language, such as Haskell or ML. Unfortunately the elegance and natural expressiveness afforded by a lazy approach implemented in a functional language comes at a price; namely the often very poor runtime performance. Imperative approaches, on the other hand, are generally more efficient—CPUs themselves work in an imperative way, making it easier to exploit all the possibilities of current CPUs—but make it harder to represent mathematical concepts. Is there a way in which we can use the best parts of both programming paradigms—the elegance of functional programming with the efficiency of imperative?

1.2 Aim

We aim to develop a library capable of exact real arithmetic for an imperative programming language. This library, which we shall call ExactLib, should integrate as seamlessly as possible with the language it is designed for. In this sense, the data type provided by the library to represent real numbers should aim to fit into the language as though it were a built-in numeric type. The library should feature a suite of functions and operations[2] for our exact real data type. These should include all those one would expect from a real arithmetic library, from basic operations such as addition to more complicated functions such as the trigonometric and exponential functions.

Arbitrary-precision arithmetic is typically considerably slower than fixed-bit representations such as floating-point. We aim for an imperative approach, using the powerful C++ language, to maximise runtime performance. Implementation in an imperative language also exposes exact real number computation to a world outside of

[2] We shall often refer to binary functions (e.g. the addition function, +) as *operations* and to unary functions (e.g. sin) as simply *functions*.

academia, where functional languages are often (unjustly) confined. However, functional languages often provide elegant solutions to problems of a mathematical nature, such as real number computation. They are highly-expressive whilst remaining concise. We aim to introduce a functional style of programming to C++ so that our library can provide the user with the ability to write C++ in a functional style to perform exact real arithmetic, whilst in the background implement the actual calculations in an efficient imperative manner. Of course, since we desire a library that integrates with C++, the main focus of ExactLib should be a traditional imperative-style suite of functions for exact real arithmetic. But we believe an experimental functional-esque extension offers an interesting alternative to the user.

Along the way, we need to consider alternative representations of real numbers to find a representation suitable for our purposes. Specifically, this is a representation that is conducive to efficient implementation and admits tractable proofs of correctness for functions and operations. In this representation, we will need to develop representations of a number of basic operations and, importantly, prove their correctness. We aim to builld upon these to derive the Taylor series expansion for our representation of real numbers, since a variety of elementary functions can be expressed as series expansions. Examples include the exponential function, trigonometric functions and the natural logarithm. These operations are so prevalent in the majority of computation tasks that any implementation of an arithmetic system must provide them if it is to be of any practical use.

Finally, we aim to design ExactLib's core in a flexible way that makes it easy to extend the library with new operation definitions in a uniform way. In particular, we hope to deliver a design that allows proven mathematical definitions of functions and operations over exact reals to be easily translated into code. Providing programming idioms that help bring the code as close as possible to the mathematical definition goes some way to relaxing the need to prove the correctness of our code.

1.3 Contributions

The following is a brief summary of the main contributions made during the undertaking of this project.

- We consider existing alternative real number representations and present a representation of the computable reals that admits tractable proofs of correctness and is well-suited to efficient imperative implementation.

- We illustrate a general method for proving correctness over this representation. We use it to define and prove the correctness of a number of basic operations and elementary functions for our real number representation.

- We develop a C++ library, ExactLib, that implements our exact real operations and functions using our representation. It introduces a data type called real that is designed to integrate seamlessly into C++. There is a focus on ease of

use and minimising the number of dependencies so that the library functions standalone and is easy to import.

- We design ExactLib's core using an object-oriented hierarchy of classes that makes it easy to extend the library with new operations in a uniform way. Our use of C++ functors makes it easy to translate proven mathematical definitions of functions and operations into code.
- Using template metaprogramming, functors and operator overloading, we introduce higher-order functions and endow C++ with some functional programming concepts. Our library can then offer the user the ability to write code in an expressive, functional language-like style to perform exact real arithmetic, whilst in the background implementing the core in an efficient imperative manner.

1.4 Outline

This report comprises three main parts:

1. *Analysis* — explores existing approaches to real arithmetic and identifies the main issues involved in representing the real numbers. Two important types of representation are considered in depth: lazy redundant streams and Cauchy approximation functions. We choose the latter and define basic operations for this representation and prove their correctness. Building upon these, we do the same for elementary functions.
2. *Implementation* — describes the design of our exact real arithmetic library, ExactLib. In particular, we discuss how we have made it easy to extend our library with new functions/operations, how our library simplifies the translation of mathematical definitions of these functions into code, and how we have added simple functional programming idioms to C++.
3. *Evaluation* — we evaluate the project quantitatively by comparing the runtime performance of our implementation to existing imperative and functional exact real arithmetic libraries. Qualitatively, we reflect on the design of our library, evaluating its expressiveness, extensibility and functionality. We conclude by suggesting possible future work.

1.5 Notation

Before we continue, a brief word on notation is needed.

- If $q \in \mathbb{Q}$, then $\lfloor q \rfloor$ is the largest integer not greater than q; $\lceil q \rceil$ is the smallest integer not less than q; and $\lfloor q \rceil$ denotes "half up" rounding such that $\lfloor q \rceil - \frac{1}{2} \leq q < n + \frac{1}{2}$.

- The *identity function* on A, $\mathsf{id}_A : A \to A$, is the function that maps every element to itself, i.e. $\mathsf{id}_A(a) = a$ for all $a \in A$. For any function $f : A \to B$, we have $f \circ \mathsf{id}_A = \mathsf{id}_A \circ f = f$, where $\circ$ denotes function composition.
- The set of *natural* numbers is denoted $\mathbb{N}$ and includes zero such that $\mathbb{N} = \{0, 1, 2, \ldots\}$, whereas $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$ refers to the strictly positive naturals, such that $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$.
- The *cons* operator ":" is used to prepend an element a to a stream α and the result is written $a : \alpha$.
- Finally, we use $\langle \cdot, \cdot \rangle$ to denote infinite sequences (streams) and $(\ldots)$ to denote finite sequences (tuples).

PART I

ANALYSIS

CHAPTER 2

REPRESENTATIONS OF REAL NUMBERS

2.1 Floating-point arithmetic

Floating-point arithmetic is by far the most common way to perform real number arithmetic in modern computers. Whilst a number of different floating-point representations have been used in the past, the representation defined by the IEEE 754 standard [32] has emerged as the industry standard.

A number in floating-point is represented by a fixed-length *significand* (a string of significant digits in a given base) and an integer *exponent* of a fixed size. More formally, a real number x is represented in floating-point arithmetic base B by

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot B^e = s \cdot m \cdot B^{e-p+1}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the significand with digits $d_i \in \{0, 1, \dots, B-1\}$ for $0 \leq i \leq p-1$ where p is the precision, and e is the exponent satisfying $e_{\min} \leq e \leq e_{\max}$. The IEEE 754 standard specifies the values of these variables for five basic formats, two of which have particularly widespread use in computer hardware and programming languages:

- Single precision — numbers in this format are defined by $B = 2, p = 23, e_{\min} = -126, e_{\max} = +127$ and occupy 32 bits. They are associated with the float type in C family languages.
- Double precision — numbers in this format are defined by $B = 2, p = 52, e_{\min} = -1022, e_{\max} = +1023$ and occupy 64 bits. They are associated with the double type in C family languages.

Due to the fixed nature of the significand and exponent, floating-point can only represent a finite subset (of an interval) of the reals. It is therefore inherently inaccurate, since we must *approximate* real numbers by their nearest representable one. Because of this, every time a floating-point operations is performed, rounding must occur, and the rounding error introduced can produce wildly erroneous results. If several operations are to be performed, the rounding error propagates, which can have a drastic impact on the calculation's accuracy, even to the point that the computed result bears no semblance to the actual exact answer.

We saw a poignant example of this in our introduction, where the propagation of rounding errors through successive iterations of a floating-point operation lead to a dramatically incorrect result. Other well-known examples include the logistic map studied by mathematician Pierre Verhulst [26], and Siegfried Rump's function and Jean-Michel Muller's sequence discussed in [24], a paper that contains a number of examples that further demonstrate the shortcomings of floating-point arithmetic.

As shown in [22], we find that increasing the precision of the floating-point representation does *not* alleviate the problem; it merely delays the number of iterations before divergence from the exact value begins. No matter what precision we use, floating-point arithmetic fails to correctly compute the mathematical limit of an infinite sequence.

2.2 Alternative approaches to real arithmetic

As we have seen, there are cases in which floating-point arithmetic does not work and we are lead to entirely incorrect results. In this section we briefly describe some existing alternative approaches to the representation of real numbers (see [26] for a more comprehensive account).

2.2.1 Floating-point arithmetic with error analysis

An extension to floating-point arithmetic that attempts to overcome its accuracy problems is floating-point arithmetic with error analysis, whereby we keep track of possible rounding errors introduced during a computation, placing a bound on any potential error. The representation consists of two floating-point numbers: one is the result computed using regular floating-point arithmetic and the other is the interval about this point that the exact value is guaranteed to reside in when taking into consideration potential rounding errors.

Whilst this approach may allow one to place more faith in a floating-point result, it does not address the real problem. For a sequence like Muller's [24], which should converge to 6 but with floating-point arithmetic converges erroneously to 100, we are left with a large range of values. This is of little help, telling us only that we should not have any faith in our result rather than helping us determine the actual value.

2.2.2 Interval arithmetic

With interval arithmetic a real number is represented using a pair of numbers that specify an interval guaranteed to contain the real number. The numbers of the pair may be expressed in any finitely-representable format (such that they are rational numbers), for example floating-point, and define the lower and upper bounds of the interval. Upon each arithmetic operation, new upper and lower bounds are determined, with strict upwards or downwards rounding respectively if the bound cannot be expressed exactly.

It turns out interval arithmetic is very useful, but it unfortunately suffers from the same problem as floating-point arithmetic with error analysis: the bounds may not be sufficiently close to provide a useful answer—computations are not made any more exact.

2.2.3 Stochastic rounding

In the previous approaches, rounding is performed to either the nearest representable number or strictly upwards or downwards to obtain a bound. The stochastic approach instead randomly chooses a nearby representable number. The desired computation is performed several times using this stochastic rounding method and then probability theory is is used to estimate the true result.

The problem with stochastic rounding is that it clearly cannot guarantee the accuracy of its results. In general, though, it gives better results than standard floating-point arithmetic [17], and also provides the user with probabilistic information about the reliability of the result.

2.2.4 Symbolic computation

Symbolic approaches to real arithmetic involve the manipulation of unevaluated expressions consisting of symbols that refer to constants, variables and functions, rather than performing calculations with approximations of the specific numerical quantities represented by these symbols. At each stage of a calculation, the result to be computed is represented exactly.

Such an approach is particularly useful for accurate differentiation and integration and the simplification of expressions. As an example, consider the expression

$$6 \arctan\left(\frac{1}{\sqrt{3}} \sin^2\left(\frac{4}{7}\right) + \frac{1}{\sqrt{3}} \cos^2\left(\frac{4}{7}\right)\right).$$

Recalling the trigonometric identity $\sin^2\theta + \cos^2\theta \equiv 1$ and noting $\tan\frac{\pi}{6} = \frac{1}{\sqrt{3}}$, we see our expression simplifies to π. It is unlikely we would obtain an accurate value of π if we were to perform this calculation using an approximating numerical representation like floating-point arithmetic, and having the symbolic answer π is arguably of more use.

The main problem with symbolic computation is that manipulating and simplifying symbols is very difficult and often cannot be done at all. Even when it becomes necessary to evaluate the expression numerically, an approach such as floating-point or interval arithmetic is required, such that we are again at the mercy of the inaccuracies associated with arithmetic in the chosen representation. Symbolic computation is therefore rarely used standalone, and so although it is useful in certain applications, it is unfortunately not general enough as a complete approach to real arithmetic.

2.2.5 Exact real arithmetic

Exact real arithmetic guarantees correct real number computation to a user-specified arbitrary precision. It does so through representations as potentially infinite data structures, such as streams. Exact real arithmetic solves the problems of inaccuracy and uncertainty associated with the other numerical approaches we have seen, and is applicable in many cases in which a symbolic approach is not feasible [26].

2.3 Exact representations of real arithmetic

2.3.1 Computable real numbers

Before we continue, we should consider which real numbers we can represent. The set of real numbers $\mathbb{R}$ is uncountable[1] so we cannot hope to represent them all on a computer. We must focus our efforts on a countable subset of the real numbers, namely the *computable real numbers*. Fortunately, the numbers and operations used in modern computation conveniently all belong to this subset [31], which is closed under the basic arithmetic operations and elementary functions.

In mathematics, the most common way of representing real numbers is the decimal representation. Decimal representation is a specific case of the more general radix representation, in which real numbers are represented as potentially infinite streams of digits. For any real number, a finite prefix of this digit stream denotes an interval; specifically, the subset of real numbers whose digit stream representations start with this prefix. For instance, the prefix 3.14 denotes the interval $[3.14, 3.15]$ since all reals starting with 3.14 are between $3.14000\ldots$ and $3.14999\ldots = 3.15$. The effect of each digit can be seen as refining the interval containing the real number represented by the whole stream, so that a B-ary representation can be seen to express real numbers as a stream of shrinking nested intervals.

Definition 2.1 (Computable real number). Suppose a real number r is represented by the sequence of intervals

$$r_0 = [a_0, b_0], \ldots, r_n = [a_n, b_n], \ldots.$$

Then r is *computable* if and only if there exists a stream of shrinking nested rational intervals $[a_0, b_0] \supseteq [a_1, b_1] \supseteq [a_2, b_2] \supseteq \cdots$ where $a_i, b_i \in \mathbb{Q}$, whose diameters converge to zero, i.e. $\lim_{n\to\infty} |a_n - b_n| = 0$. Thus r is the unique number common to all intervals:

$$r = r_\infty = \bigcap_{n\geq 0} r_n.$$

The mapping $r : \mathbb{N} \to [\mathbb{Q}, \mathbb{Q}]$ is a computable function, as already defined by Turing [30] and Church [9]. We see that real numbers are converging sequences of rational numbers. From now on, unless otherwise stated, when we refer to *real numbers* we shall actually mean the *computable real numbers*.

[1] Proved using Georg Cantor's diagonal argument, in which it is shown $\mathbb{R}$ cannot be put into one-to-one correspondence with $\mathbb{N}$.

2.3.2 Representation systems

Just as we can denote the natural numbers exactly in several ways, e.g. binary, decimal, etc., there are several exact representation systems for the reals. Within one system there are often several representations for one single real—for example 3.14999... and 3.15—similar to the rationals, $\mathbb{Q}$, where $\frac{1}{3}$, $\frac{2}{6}$ and $\frac{962743}{2888229}$ refer to the same quantity.

Definition 2.2 (Representation system). A *representation system* of real numbers is a tuple (S, π) where $S \subseteq \mathbb{N} \to \mathbb{N}$ and π is a surjective function from S to $\mathbb{R}$. If $s \in S$ and $\pi(s) = r$, then s is said to be a (S, π)-representation of r.

As we require all real numbers to be representable, π is necessarily surjective, whereas we do not require unique denotations of the reals such that π need not be injective. We think of elements in S as denotations of real numbers. Elements of S are functions in $\mathbb{N} \to \mathbb{N}$ since $\mathbb{N} \to \mathbb{N}$ has the same cardinality as $\mathbb{R}$ and captures the infinite nature of representation systems. In a computer implementation, we desire the ability to specify a real number stepwise. A function s in $\mathbb{N} \to \mathbb{N}$ can be given by an infinite sequence $(s(0), s(1), \ldots)$ called a stream. We do not expect that we can overview a real number immediately, but that we should be able to approximate the real number by rational numbers with arbitrary precision using the representation.

In any approach to exact real arithmetic, the choice of representation determines the operations we can computably define and implement. Indeed there are bad choices of representations that do not admit definitions for the most basic of arithmetic operations. Different approaches also confer wildly different performances and design challenges when implemented as computer programs. With this in mind, the remainder of this chapter illustrates some of the issues involved in exact representations for computation, introduces the representation used in our work, and briefly discusses some of the other representations we considered.

2.3.3 Computability issues with exact representations

Although the standard decimal digit stream representation is natural and elegant, it is surprisingly difficult to define exact real number arithmetic in a computable way. Consider the task of computing the sum $x + y$ where

$$x = \frac{1}{3} = 0.333\ldots \qquad \text{and} \qquad y = \frac{2}{3} = 0.666\ldots$$

using the standard decimal expansion. Clearly the sum $x + y$ has the value $0.999\ldots = 1$, but we cannot determine whether the first digit of the result should be a zero or a one. Recall that at any time, only a finite prefix of x and y is known and this is the only information available in order to produce a finite prefix of $x + y$. Suppose we have read the first digit after the decimal point for both expansions of x and y. Do we have enough information to determine the first digit of $x + y$? Unfortunately not: we know x lies in the interval $[0.3, 0.4]$ and y in the interval $[0.6, 0.7]$, such that $x + y$ is contained in the interval $[0.9, 1.1]$. Therefore, we know the first digit should be a 0 or

a 1, but at this point we cannot say which one it is—and if we pre-emptively choose 0 or 1 as the first digit, then the intervals we can represent, $[0, 1]$ or $[1, 2]$ respectively, do not cover the output interval $[0.9, 1.1]$. Reading more digits from x and y only tightens the interval $[0.9 \ldots 9, 1.0 \ldots 1]$ in which $x + y$ lies in, but its endpoints are still strictly less and greater than 1. Hence, we cannot ever determine the first digit of the result, let alone any others.

Similarly, consider the product $3x$. Again the result should have the value $0.999 \ldots = 1$, and if we know the first digit after the decimal point for the expansion of x, we know x lies in the interval $[0.3, 0.4]$. Then the result $3x$ lies in the interval $[0.9, 1.2]$, and so we encounter the same problem: we cannot determine whether the first digit should be a 0 or a 1.

We see that when we perform computations with infinite structures such as the digit stream representation, there is no 'least significant' digit since the sequence of digits extends infinitely far to the right. Therefore, we must compute operations on sequences from left ('most significant') to right ('least significant'), while most operations in floating-point are defined from right to left; for example, consider arithmetic. In general, it is not possible to determine an operation's output digit without examining an infinite number of digits from the input. Thus the standard decimal expansion is unsuitable for exact arithmetic.

2.4 Lazy representations

In this section we consider some existing lazy representations of exact real arithmetic. We start by looking at the most-popular lazy representation, the redundant signed-digit stream, before briefly outlining two others. We finish by discussing the problems with lazy representations in general.

2.4.1 Redundant signed-digit stream

The problem presented above in 2.3.3 can be overcome by introducing some form of *redundancy*. In cases such as the one above, redundancy allows us to make an informed guess for a digit of the output and then compensate for this guess later if it turns out to have been incorrect.

One of the most intuitive ways to introduce redundancy is by modifying the standard digit stream representation to admit negative digits. This representation is termed the *signed-digit representation* and is covered more extensively in [25] and [12]. While the standard digit representation involves a stream of positive integer digits in base B, i.e. from the set $\{b \in \mathbb{N} \mid x < B\} = \{0, 1, 2, \ldots, B-1\}$, the signed digit representation allows digits to take negative base B values, such that the set of possible digits is extended to $\{b \in \mathbb{Z} \mid |x| < B\} = \{-B+1, -B+2, \ldots, -1, 0, 1, \ldots, B-1\}$. For elements d_i of this set, the sequence $r = \langle d_1, d_2, \ldots \rangle$ then represents the real number

$$[\![r]\!] = \sum_{i=1}^{\infty} \frac{d_i}{B^i}.$$

in the interval $[-B+1, B-1]$. A specific case of signed-digit representation is that of $B = 2$, called the *binary signed-digit representation*, or simply *signed binary representation*. In this representation, digits come from the set $\{-1, 0, 1\}$ and can represent real numbers in the interval $[-1, 1]$.

Consider again the example given in 2.3.3, where we attempted to compute the sum $x + y$ for the numbers $x = 0.333\ldots$ and $y = 0.666\ldots$. It should be clear how the negative digits admitted by the signed-digit representation can be used to correct previous output. After reading the first digit following the decimal point, we know x lies in the interval $[0.2, 0.4]$ and similarly that y lies in the interval $[0.5, 0.7]$ (the interval diameters are widened since a sequence could continue $-9, -9, \ldots$) such that their sum is contained in $[0.7, 1.1]$. We may now safely output a 1 for the first digit, since a stream starting with 1 can represent any real number in $[0, 2]$. If it turns out later that x is less than $\frac{1}{3}$, so the result $x + y$ is actually smaller than 1, we can compensate with negative digits.

The signed-digit representation we have presented can only be used to represent real numbers in the interval $[-B+1, B+1]$. Compare this to the standard decimal expansion, where we can only represent numbers in the range $[0, 1]$. To cover the entire real line, we introduce the decimal (or, more generally, *radix*) point. The radix point serves to specify the magnitude of the number we wish to represent. This is achieved using a mantissa-exponent representation, where an exponent $e \in \mathbb{Z}$ scales the number. Therefore, we refine our signed-digit representation such that the sequence and exponent pair $r = (\langle d_1, d_2, \ldots \rangle, e)$ represents the real number

$$[\![r]\!] = B^e \sum_{i=1}^{\infty} \frac{d_i}{B^i}.$$

This representation now covers the entire real line, i.e. r can represent any computable real number in $[-\infty, \infty]$.

The redundant signed-digit stream representation involves potentially infinite data structures that can be studied using the mathematical field of universal coalgebra. A coinductive strategy for studying and implementing arithmetic operations on the signed-digit stream representation is proposed in [19]. The signed-digit stream representation lends itself nicely to a realisation in a lazy functional programming language, and the implementations are often elegant, but in practice they exhibit very poor performance.

2.4.2 Continued fractions

Every real number r has a representation as a generalised continued fraction [10], defined by a pair of integer streams $(a_n)_{n \in \mathbb{N}}$ and $(b_n)_{n \in \mathbb{N}^+}$ such that

$$r = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \ddots}}$$

Some numbers that are difficult to express closed-form using a digit notation can be represented with remarkable simplicity as a continued fraction. For example, the golden ratio $\varphi = \frac{1+\sqrt{5}}{2}$ that we met in the introduction can be represented as a continued fraction defined simply by the sequences $(a_n) = (b_n) = (1, 1, 1, \ldots)$.

A number of examples of transcendental numbers and functions are explored in [33], including an expression of the mathematical constant π as the sequences

$$a_n = \begin{cases} 3 & \text{if } n = 0 \\ 6 & \text{otherwise} \end{cases}$$
$$b_n = (2n-1)^2$$

and the exponential function e^x as the sequences

$$a_n = \begin{cases} 1 & \text{if } n = 0, 1 \\ n + x & \text{otherwise} \end{cases}$$
$$b_n = \begin{cases} x & \text{if } n = 1 \\ (1-n)x & \text{otherwise.} \end{cases}$$

The continued fraction representation is incremental. However, a major disadvantage of continued fractions is that evaluation of the sequences is monotonic; once a digit has been calculated it cannot be changed, such that some standard arithmetic operations on the constructive reals are uncomputable with this representation. Jean Vuillemin discusses the performance of an implementation of exact real arithmetic with continued fractions in [31].

2.4.3 Linear fractional transformations

A one-dimensional linear fractional transformation (LFT)—also known as a Möbius transformation—is a complex function $f : \mathbb{C} \to \mathbb{C}$ of the form

$$f(z) = \frac{az + c}{bz + d}$$

with four fixed parameters a, b, c, d satisfying $ad - bc \neq 0$. In general these parameters are arbitrary complex numbers, but in the context of exact real arithmetic we only consider LFTs with integer parameters. A two-dimensional linear fractional transformation is a complex function $g : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ of the form

$$g(z_1, z_2) = \frac{az_1z_2 + cz_1 + ez_2 + g}{bz_1z_2 + dz_1 + fz_2 + h}$$

where a, b, c, d, e, f, g, h are fixed arbitrary complex parameters.

An advantage of linear fractional transformations is that they can be written as matrices, allowing their composition (and application) to be computed using matrix multiplication. The one-dimensional LFT f and two-dimensional LFT g above are represented as matrices M_1 and M_2 respectively, where

$$M_1 = \begin{pmatrix} a & c \\ b & d \end{pmatrix}, \qquad M_2 = \begin{pmatrix} a & c & e & g \\ b & d & f & h \end{pmatrix}.$$

It is shown in [12] that any real number r can be represented as an infinite composition of LFTs with integer coefficients applied to the interval $[-1, 1]$, such that

$$r = \lim_{n\to\infty} \left[\left(\prod_{i=1}^{n} M_i \right) ([-1, 1]) \right].$$

With the right choice of LFTs, each element of the sequence is a better approximation to r than the previous one. That is, like continued fractions, linear fractional transformations are incremental.

The attractiveness of LFTs is that many of the concise continued fractions for transcendental numbers and functions discussed in 2.4.2 and [33] can be directly translated into infinite products of LFTs, as demonstrated in [27].

2.4.4 Problems with lazy representations

The apparent advantage of a lazy approach is that computation proceeds incrementally on demand. Subjectively, it is also an extremely elegant approach and highly conducive to an implementation in a functional programming language. However, lazy representations suffer from a serious problem, which we refer to as the problem of *granularity*.

This is a problem best explained with an example. Consider two real numbers x and y represented as the redundant lazy streams $\langle 2, 1, 4, 5, \ldots \rangle$ and $\langle 3, 3, 3, 3, \ldots \rangle$ respectively, and suppose we wish to compute the sum $x + y$. We get $\langle 5, 4, 7, \circ, \ldots \rangle$ but cannot determine whether the last digit $\circ$ should be 8 or 9 since we do not know the fifth digits of the two summands. Thus, to compute an approximation of $x + y$ accurate to four digits, we require approximations to x and y accurate to five digits. Now consider evaluation of the expression $x + (x + (x + (x + (x + x))))$. By the same reasoning, if we desire n digits of accuracy, the outermost addition will require $n + 1$ digits of accuracy from its arguments, the next outermost $n + 2$, and so on, until the innermost addition which requires $n + 5$ digits of accuracy. In general, if we are to take advantage of the underlying machine's arithmetic capabilities, we should use machine words as the size of the digits in the representative lists for optimal efficiency. Suppose we use digits represented with 32 bits; then we are using 160 extra bits of accuracy in each of the deepest arguments, when in fact it can be shown that only $\lceil \log_2 6 \rceil = 3$ extra bits are required for a summation of six values: for a digit d_i that we are unsure about, we can simply use a single additional bit to represent whether its value should be $x_i + y_i$ or $x_i + y_i + 1$. The choice of representation means 157 digits more than necessary are computed. Whilst relatively minor for addition, the problem posed by coarse granularity is significantly worse for more complex operations than addition. For example, in the case of division, the number of extra digits calculated at each level of an expression can unnecessarily grow exponentially with expression depth [4], leaving us with an intractable computation.

Unfortunately granularity is not the only problem associated with lazy representations; other issues include:

- The large lists used to represent numbers require a lot of memory, leading to all sorts of hardware bottlenecks.
- Reasoning about correctness, termination and resource consumption for lazy representations seems to be quite difficult.
- Functional languages operate at a high level of abstraction from the machine; programs run slowly compared to their imperative counterparts.

Finally, one of our original project aims was to develop a library suitable for mainstream programming audiences. As such, we chose C++ and so a lazy representation is not appropriate for our purpose. Whilst it is possible to implement lazy evaluation in an imperative language like C++, it is unnatural and simply not the right tool for the job. Indeed it seems an exercise in futility: it is not likely our implementation of lazy lists would come close to achieving the efficiency of existing compilers for languages like Haskell, which are the result of 30 years of functional compiler research.

2.5 Cauchy sequence representation

In classical real analysis, real numbers are traditionally represented as Cauchy sequences of rational numbers. The real number defined by this sequence is the limit under the usual one-dimensional Euclidean metric. We start by looking at the most general case, the *naïve* Cauchy representation, in which no modulus of convergence[2] is required.

Definition 2.3 (Naïve Cauchy representation). A *naïve Cauchy representation* of a real number x is a function $f_x : \mathbb{N} \to \mathbb{Q}$ such that, for all $n \in \mathbb{N}$,

$$|f_x(n) - x| < \frac{1}{n}.$$

This says that for any constructive real number x, there is a function f_x that can provide an arbitrarily close approximation to x. As $n \to \infty$, the maximum error $1/n$ goes to 0. This approach of implementing real numbers directly as functions on a fixed countable set is referred to as a *functional*[3] *representation*.

In practice, there are a number of disadvantages to using this representation. First, the range of f_x is over the rationals, and computation on rationals is itself expensive. Second, potentially huge values of n are necessary to specify very accurate approximations to x. For example, to calculate 100 digits of accuracy we would have to use $n = 10^{100}$, an extremely large number, which makes for very expensive computation. Finally, since n takes integer values, it is not possible from f_x to obtain an approximation to x that is *less* accurate than an error of ± 1—a problem when x is very large, such as $x = 10^{50}$.

[2] If $x_1, x_2, \ldots$ is a Cauchy sequence in the set X, then a *modulus of convergence* is a function $\alpha : \mathbb{N} \to \mathbb{N}$ such that $\forall k \forall m, n > \alpha(k).|x_m - x_n| < 1/k$ that tells how quickly a convergent sequence converges.

[3] Not to be confused with functional programming languages.

In [5], Boehm suggests the following representation, which we shall refer to as the *functional Cauchy representation* or *approximation function representation*.

Definition 2.4 (**Functional representation 1**)**.** A *functional Cauchy representation* of a real number x is a computable function $f_x : \mathbb{Z} \to \mathbb{Z}$ such that, for all $n \in \mathbb{Z}$,

$$|B^{-n} f_x(n) - x| < B^{-n}$$

where $B \in \mathbb{N}, B \geq 2$ is the base used to specify precision.

The result produced by f_x is *implicitly scaled* so that it is possible to approximate a real number x to arbitrary accuracy. Informally, the argument n specifies the precision so that $f_x(n)$ produces an approximation to x that is accurate to within an error of B^{-n}. The argument n is typically (but not necessarily) positive, indicating that the result should be accurate to n places to the right of the decimal point.

It is easy to see that $(f_x/B^n)_{n\in\mathbb{N}}$ describes a Cauchy sequence of rational numbers converging to x with a prescribed ratio of convergence. As a simple example, we can represent the integer $y = 5$ as the approximation function $f_y(n) = B^n y = B^n \cdot 5$ so that it generates the sequence $(5)_{n\in\mathbb{N}}$.

Since n is now an exponent, specifying accurate approximations will no longer require the extremely large values that make computations expensive. Also since positive n yields a tolerance greater than 1, we can compute results of any accuracy. We need not store the scale factor B^{-n} and, critically for space efficiency, the size of the scale factor grows linearly with the precision of the real number. Boehm suggests $B = 2$ or $B = 4$ for effective implementation, with $B = 4$ simplifying mathematical exposition and $B = 2$ providing optimal performance for scaling $B^{-n} f_x(n)$. We shall use $B = 2$ such that $B^{-n} f_x(n)$ generates a sequence of dyadic rationals.

Theorem 2.5. Every computable real number can be represented as a Cauchy approximation function and vice versa.

Proof. Intuitively this seems obvious considering Definition 2.1 of computable real numbers. However, the proof is rather long and distracting, so we omit it here. It can be found in a number of papers including [22, 16, 5].

2.5.1 Representing functions

Definition 2.6 (**Functional representation 2**)**.** A *functional Cauchy representation* of a *real function* $F : \mathbb{R} \to \mathbb{R}$ is a function $f_F : (\mathbb{Z} \to \mathbb{Z}) \times \mathbb{Z} \to \mathbb{Z}$ that, for any $x \in \mathrm{dom}F$ and all functional Cauchy representations f_x of x, satisfies

$$|B^{-n} f_F(f_x, n) - F(x)| < B^{-n}.$$

Definition 2.7 (**Functional representation 3**)**.** A *functional representation* of a *real function* $F : \mathbb{R}^m \to \mathbb{R}$ is a function $f_F : (\mathbb{Z} \to \mathbb{Z})^m \times \mathbb{Z} \to \mathbb{Z}$ that, for any $\vec{x} = (x_1, x_2, \ldots, x_m) \in \mathrm{dom}F$ and all $f_{\vec{x}} = (f_{x_1}, f_{x_2}, \ldots, f_{x_m})$ representations of $\vec{x}$, satisfies

$$|B^{-n} f_F(f_{\vec{x}}, n) - F(\vec{x})| < B^{-n}. \tag{2.1}$$

We define[4] approximation functions to represent operations. To prove a function's *correctness* we must show its definition is such that it satisfies (2.1). In Chapters 3 and 4 we define operations and then use this method of proof to verify their correctness. The approximation functions we define to represent operations return elements in $\mathbb{Z}$. Whilst in practice the argument to an approximation—also in $\mathbb{Z}$—is relatively small, the values returned are usually very large. This defining of approximation functions in terms of integer arithmetic means an implementation will require an arbitrary-integer type. In practice, arbitrary-integer arithmetic libraries are relatively easy to implement and verify—a large number of good are available and open-source. In a sense we use easier-to-verify integer arithmetic to implement real number arithmetic. The remainder of our work assumes we have a verifiably-correct integer arithmetic.

2.5.2 Implementing functions

As we shall see in Chapter 3, the addition operation can be defined as

$$f_{x+y}(n) = \left\lfloor \frac{f_x(n+2) + f_y(n+2)}{B^2} \right\rfloor$$

It combines the functions for two real numbers x and y to give a functional representation of $x + y$. When called with an accuracy request of n, f_{x+y} first invokes f_x and f_y with $n + 2$ to compensate for the accuracy that will be lost in addition. It then adds the resultant integers returned by f_x and f_y, before scaling the result by B^2 so that f_{x+y} is scaled to n rather than $n + 2$.

We define other mathematical operations in a similar way, evaluating the approximation functions for their operands to some higher precision. Operations can be applied to one another such that we form directed acyclic graphs (DAGs), or *expression trees*, of approximation functions. Computation with this style of representation proceeds as in Figure 2.1. When the approximation function for an operation is called with precision argument n, the accuracy required from each of its arguments is calculated. These higher accuracies are passed to the approximation function for each of its arguments, and the process repeats. This continues until a leaf node—i.e. an operation whose approximation function does not invoke another, such as a constant value like 5—is reached, which simply returns its requested approximation. In general, when the arguments to a function have returned the requested approximations, the function must compute its result by shifting away the extra precision that was required to calculate an approximation for its arguments without round-off error. Computation proceeds back up the expression DAG until the root function that requested an approximation is reached.

In summary, computation flows down from the original node to the leaves and then back to the top. We see the requested precision passed to the original operation (node) becomes larger and larger as it is passed further and further down into the expression tree. On the way back up, at each stage a calculation is performed followed by

[4] Later, we shall often refer to the function definition as an *algorithm*, since it is slightly confusing to talk about proving a definition—one does prove definitions.

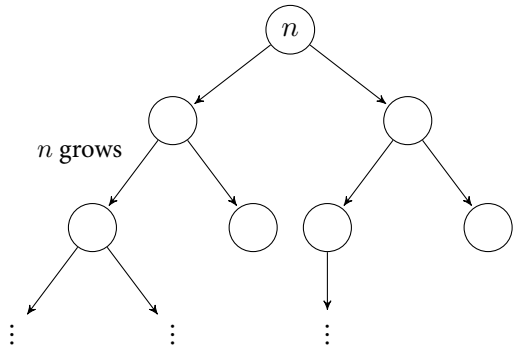


Figure 2.1: Top-down propagation of accuracy growth.

a scaling operation that puts the intermediate node's result into the appropriate form for use by its parent, before finally we reach the top and the requested approximation is produced. We characterise this computation as *top-down*.

The approximation function representation avoids the granularity problem associated with lazy representations. However, in general, functional representations cannot make use of previously calculated results when asked to produce approximations to higher[5] precisions—it is not an *incremental* representation. Whilst some operations have forms that are incremental, most have to start from scratch if they receive a request for an accuracy higher than any previously computed. This can be particularly problematic in certain situations. An alternative approach, used by Norbert Müller's extremely fast iRRAM library, uses precision functions and interval arithmetic to produce bottom-up propagation. Whilst this is more efficient in a lot of situations, importantly there is no proof for the correctness of its operations. Bottom-up exact number computation is also time-consuming to implement and not suitable given the time constraints of an undergraduate project.

After careful consideration we settle on the top-down approximation function (Cauchy sequence) representation for our exact real arithmetic library. It meets our requirement of a representation that is conducive to efficient implementation whilst still admitting tractable proofs of correctness for functions and operations.

[5]We can, though, store a cache of the best yet approximation and use it provide a result if a lower accuracy is requested, by scaling it down as necessary.

CHAPTER 3

BASIC OPERATIONS

In Chapter 2 we explored representations of real numbers and chose a representation, the functional dyadic Cauchy representation, that will form the basis of our work. In this chapter we develop the basic arithmetic operations, including some useful auxiliary functions, in our representation and prove their correctness using the method of proof shown in section 2.5.1 and as seen in [16].

3.1 Auxiliary functions and preliminaries

3.1.1 Representing integers

Algorithm 3.1. We can represent an integer x by the approximation function f_x that satisfies

$$\forall n \in \mathbb{Z} \,.\, f_x(n) = \lfloor 2^n x \rceil \,.$$

Proof. By the definition of $\lfloor q \rceil$ for $q \in \mathbb{Q}$,

$$\begin{aligned} & 2^n x - \frac{1}{2} \leq f_x(n) < 2^n x + \frac{1}{2} \\ \Leftrightarrow \quad & x - 2^{-(n+1)} \leq 2^{-n} f_x(n) < x + 2^{-(n+1)} \\ \Leftrightarrow \quad & -2^{-(n+1)} \leq 2^{-n} f_x(n) - x < 2^{-(n+1)} \\ \Leftrightarrow \quad & |2^{-n} f_x(n) - x| \leq 2^{-(n+1)}. \end{aligned}$$

Therefore $|2^{-n} f_x(n) - x| < 2^{-n}$ such that f_x represents the integer number x. □

3.1.2 Negation

Algorithm 3.2. If a real number x is represented by the approximation function f_x, then its *negation* $-x$ is represented by the approximation function f_{-x} that satisfies

$$\forall n \in \mathbb{Z} \,.\, f_{-x}(n) = -f_x(n).$$

Proof. By the definition of f_x, for all $n \in \mathbb{Z}$, we have $|2^{-n} f_x(n) - x| < 2^{-n} \Leftrightarrow |2^{-n}(-f_x(n)) - (-x)| < 2^{-n}$ and $f_{-x}(n) = -f_x(n)$ such that $|2^{-n} f_{-x}(n) - (-x)| < 2^{-n}$. □

3.1.3 Absolute value

Algorithm 3.3. If a real number x is represented by the approximation function f_x, then its *absolute value* $|x|$ is represented by the approximation function $f_{|x|}$ that satisfies

$$\forall n \in \mathbb{Z} \,.\, f_{|x|}(n) = |f_x(n)|.$$

Proof. By the definition of f_x, for all $n \in \mathbb{Z}$, we have

$$\begin{aligned} & |x - 2^{-n} f_x(n)| < 2^{-n} \\ \Leftrightarrow \quad & -2^{-n} < x - 2^{-n} f(n) < 2^{-n} \\ \Leftrightarrow \quad & 2^{-n}(f_x(n) - 1) < x < 2^{-n}(f_x(n) + 1). \end{aligned}$$

Therefore, since $f_x(n) \in \mathbb{Z}$ and $2^{-n} > 0$, if $x \leq 0$ then $f_x(n) \leq 0$ so that $|f_x(n)| = -f_x(n)$ and $|x| = -x$. Similarly, if $x \geq 0$ then $f_x(n) \geq 0$ so that $|f_x(n)| = f_x(n)$ and $|x| = x$. Hence $|2^{-n}|f_x(n)| - |x|| = |2^{-n} f_x(n) - x| < 2^{-n}$, and so $|2^{-n} f_{|x|}(n) - |x|| < 2^{-n}$. □

3.1.4 Shifting

Algorithm 3.4. If a real number x is represented by the approximation function f_x, then its shift $2^k x$, where $k \in \mathbb{Z}$, is represented by the approximation function $f_{2^k x}$ that satisfies

$$\forall n \in \mathbb{Z} \,.\, f_{2^k x}(n) = f_x(n + k)$$

Proof. From the definition of f_x, we have

$$\left|2^{-(n+k)} f_x(n + k) - x\right| < 2^{-(n+k)}$$

so that multiplying through by 2^k yields $\left|2^{-n} f_x(n + k) - 2^k x\right| < 2^{-n}$. Therefore $f_{2^k x}$ represents the real number $2^k x$. □

3.1.5 Most significant digits

Definition 3.5. A function $\mathsf{msd} : \mathbb{R} \to \mathbb{Z}$ extracts the *most significant digits* of a real number. Given a real number x, $\mathsf{msd}(x)$ satisfies

$$2^{\mathsf{msd}(x)-1} < |x| < 2^{\mathsf{msd}(x)+1}.$$

3.2 Addition and subtraction

3.2.1 Addition

Algorithm 3.6. If two real numbers x and y are represented by the approximation functions f_x and f_y respectively, then their *addition* $x + y$ is represented by the approximation function f_{x+y} that satisfies

$$\forall n \in \mathbb{Z} \,.\, f_{x+y}(n) = \left\lfloor \frac{f_x(n + 2) + f_y(n + 2)}{4} \right\rfloor. \tag{3.1}$$

Proof. By the definition of f_x and f_y, for all $n \in \mathbb{Z}$,

$$\left|2^{-(n+2)} f_x(n+2) - x\right| < 2^{-(n+2)} \qquad \text{and} \qquad \left|2^{-(n+2)} f_y(n+2) - y\right| < 2^{-(n+2)}.$$

We have

$$\begin{aligned} & \left|2^{-(n+2)}(f_x(n+2) + f_y(n+2)) - (x+y)\right| \\ \leq\ & \left|2^{-(n+2)} f_x(n+2) - x\right| + \left|2^{-(n+2)} f_y(n+2) - y\right| \\ <\ & 2^{-(n+1)}. \end{aligned} \tag{3.2}$$

By (3.1) and the definition of $\lfloor \cdot \rceil$ over $\mathbb{Q}$,

$$\begin{aligned} & f_{x+y}(n) - \frac{1}{2} \leq \frac{f_x(n+2) + f_y(n+2)}{4} < f_{x+y}(n) + \frac{1}{2} \\ \Leftrightarrow\quad & -\frac{1}{2} \leq \frac{f_x(n+2) + f_y(n+2)}{4} - f_{x+y}(n) < \frac{1}{2} \\ \Leftrightarrow\quad & \left|f_{x+y}(n) - \frac{f_x(n+2) + f_y(n+2)}{4}\right| \leq \frac{1}{2} \\ \Leftrightarrow\quad & \left|2^{-n} f_{x+y}(n) - 2^{-(n+2)}(f_x(n+2) + f_y(n+2))\right| \leq 2^{-(n+1)}. \end{aligned} \tag{3.3}$$

The triangle inequality states that $|a-b| \leq |a-c| + |c-b|$ for any $a, b, c \in \mathbb{Q}$. Therefore, by (3.3) and (3.2), we have

$$|2^{-n} f_{x+y}(n) - (x+y)| < 2^{-(n+1)} + 2^{-(n+1)} = 2^{-n}. \qquad \square$$

3.2.2 Subtraction

Since $x - y = x + (-y)$ for any two real numbers x and y, we trivially define subtraction f_{x-y} in terms of addition and negation, such that $f_{x-y} = f_{x+(-y)}$.

3.3 Multiplication

Reasoning about multiplication is slightly more complicated than the previous operations we have seen and so deserves some discussion. If x and y are two real numbers represented by the approximation functions f_x and f_y respectively, we seek an approximation function f_{xy} representing the multiplication xy such that

$$\forall n \in \mathbb{Z}.\ |2^{-n} f_{xy}(n) - xy| < 2^{-n}.$$

Naturally, the function f_{xy} depends on f_x and f_y. We aim to evaluate the multiplicands x and y to the lowest precision that will still allow us to evaluate the multiplication xy to the required accuracy. However, when performing multiplication, the required precision in one multiplicand of course depends on the approximated value of the other. We consider the following two cases.

Suppose we evaluate x to roughly half the required precision for the result xy. If the approximation is zero, we evaluate y to the same precision. If the approximation to y is also zero, the result xy is simply zero.

In the general non-zero case, we consider the error in the calculation's result in order to determine the lowest precision with which x and y need to be evaluated. As we shall see in our proof, if x is evaluated with error ε_x and y with error ε_y, we find that the overall calculation's error $|(x+\varepsilon_x)(y+\varepsilon_y)-xy|$ is bounded above by $2^{\mathsf{msd}(x)+1}|\varepsilon_y|+2^{\mathsf{msd}(y)+1}|\varepsilon_x|$. If xy needs to be evaluated to precision n, by definition we require $2^{\mathsf{msd}(x)+1}|\varepsilon_y| + 2^{\mathsf{msd}(y)+1}|\varepsilon_x| < 2^{-n}$. Therefore we choose $|\varepsilon_y| < 2^{-(n+\mathsf{msd}(y)+3)}$ and $|\varepsilon_x| < 2^{-(n+\mathsf{msd}(x)+3)}$. That is, we evaluate x to precision $n + \mathsf{msd}(x) + 3$ and y to precision $n + \mathsf{msd}(y) + 3$.

Algorithm 3.7. If two real numbers x and y are represented by the approximation functions f_x and f_y respectively, then their *multiplication* xy is represented by the approximation function f_{xy} that satisfies

$$\forall n \in \mathbb{Z}\,.\, f_{xy}(n) = \begin{cases} 0 & \text{if } f_x(\frac{n}{2}+1) = 0 \text{ and } f_y(\frac{n}{2}+1) = 0 \\ \left\lfloor \dfrac{f_x(n+\mathsf{msd}(x)+3)\,f_y(n+\mathsf{msd}(y)+3)}{2^{n+\mathsf{msd}(x)+\mathsf{msd}(y)+6}} \right\rfloor & \text{otherwise.} \end{cases}$$

Proof. In the first case, if $f_x(\frac{n}{2}+1) = 0$ and $f_y(\frac{n}{2}+1) = 0$, then by the definition of f_x and f_y we have $|x| < 2^{-(\frac{n}{2}+1)}$ and $|y| < 2^{-(\frac{n}{2}+1)}$. Therefore if $f_{xy}(n) = 0$, then

$$|f_{xy}(n) - xy| = |xy| \le |x||y| < 2^{-(n+2)} < 2^{-n}.$$

We now consider the general case. For the sake of notational brevity, let

$$i = n + \mathsf{msd}(y) + 3, \quad j = n + \mathsf{msd}(x) + 3, \quad r = \left\lfloor \frac{f_x(i)f_y(j)}{2^{i+j-n}} \right\rfloor.$$

By the definition of f_x, we have $\left|2^{-i}f_x(i) - x\right| < 2^{-i}$. Thus, at precision i we have approximation error $\varepsilon_x = 2^{-i}f_x(i) - x$ with $|\varepsilon_x| < 2^{-i}$ and write $x + \varepsilon_x = 2^{-i}f_x(i)$. Similarly, for f_y at j we have approximation error $\varepsilon_y = 2^{-j}f_y(j) - y$ with $|\varepsilon_y| < 2^{-j}$ and write $y + \varepsilon_y = 2^{-j}f_y(j)$. We note that the error in the multiplication result is

$$\begin{aligned} (x+\varepsilon_x)(y+\varepsilon_y) - xy &= x\varepsilon_y + y\varepsilon_x + \varepsilon_x\varepsilon_y \\ &= (x+\varepsilon_x)\varepsilon_y + (y+\varepsilon_y)\varepsilon_x - \varepsilon_x\varepsilon_y \\ &= \frac{1}{2}\left(x\varepsilon_y + y\varepsilon_x + (x+\varepsilon_x)\varepsilon_y + (y+\varepsilon_y)\varepsilon_x\right) \end{aligned}$$

such that $|(x+\varepsilon_x)(y+\varepsilon_y) - xy| = \frac{1}{2}|x\varepsilon_y + y\varepsilon_x + (x+\varepsilon_x)\varepsilon_y + (y+\varepsilon_y)\varepsilon_x|$. Therefore the error is bounded by the maximum of $|x\varepsilon_y + y\varepsilon_x|$ and $|(x+\varepsilon_x)\varepsilon_y + (y+\varepsilon_y)\varepsilon_x|$, i.e.

$$|(x+\varepsilon_x)(y+\varepsilon_y) - xy| \le \max\left(|x\varepsilon_y + y\varepsilon_x|, |(x+\varepsilon_x)\varepsilon_y + (y+\varepsilon_y)\varepsilon_x|\right). \tag{3.4}$$

Now, again by the definition of f_x we have $\left|2^{-i} f_x(i) - x\right| < 2^{-i}$ such that

$$2^{-i}(f_x(i) - 1) < x < 2^{-i}(f_x(i) + 1).$$

If $x \geq 0$, then we require $f_x(i) + 1 > 0$ and so $f_x(i) \geq 0$ since $f_x(i) \in \mathbb{Z}$. Otherwise, if $x < 0$, then $f_x(i) - 1 < 0$ and so $f_x(i) \leq 0$ since $f_x(i) \in \mathbb{Z}$. Therefore

$$2^{-i}(|f_x(i)| - 1) < |x| < 2^{-i}(|f_x(i)| + 1).$$

Recalling that by definition $\mathsf{msd}(x)$ satisfies $|x| < 2^{\mathsf{msd}(x)+1}$, we have $2^{-i}(|f_x(i)|-1) < |x| < 2^{\mathsf{msd}(x)+1}$. Thus, $|f_x(i)|-1 < 2^{\mathsf{msd}(x)+1+i}$ so that $|f_x(i)| \leq 2^{\mathsf{msd}(x)+1+i}$. Therefore we obtain

$$|x + \varepsilon_x| = \left|2^{-i} f_x(i)\right| = 2^{-i}\,|f_x(i)| \leq 2^{\mathsf{msd}(x)+1}$$

and a similar derivation for y at precision j yields

$$|y + \varepsilon_y| = \left|2^{-j} f_y(j)\right| = 2^{-j}\,|f_y(j)| \leq 2^{\mathsf{msd}(y)+1}.$$

By the definition of msd and remembering $|\varepsilon_x| < 2^{-i}$, $|\varepsilon_y| < 2^{-j}$,

$$|x\varepsilon_y + y\varepsilon_x| \leq |x\varepsilon_y| + |y\varepsilon_x| \leq |x||\varepsilon_y| + |y||\varepsilon_x| < 2^{\mathsf{msd}(x)+1} \cdot 2^{-j} + 2^{\mathsf{msd}(y)+1} \cdot 2^{-i}$$

and

$$\begin{aligned} |(x + \varepsilon_x)\varepsilon_y + (y + \varepsilon_y)\varepsilon_y| &\leq |(x + \varepsilon_x)\varepsilon_y| + |(y + \varepsilon_y)\varepsilon_y| \\ &\leq |x + \varepsilon_x||\varepsilon_y| + |y + \varepsilon_y||\varepsilon_y| \\ &< 2^{\mathsf{msd}(x)+1} \cdot 2^{-j} + 2^{\mathsf{msd}(y)+1} \cdot 2^{-i}. \end{aligned}$$

Therefore, $\max\big(|x\varepsilon_y + y\varepsilon_x|, |(x + \varepsilon_x)\varepsilon_y + (y + \varepsilon_y)\varepsilon_x|\big)$ is bounded by $2^{\mathsf{msd}(x)+1-j} + 2^{\mathsf{msd}(y)+1-i}$. Recalling (3.4), we get $|(x + \varepsilon_x)(y + \varepsilon_y) - xy| < 2^{\mathsf{msd}(x)+1-j} + 2^{\mathsf{msd}(y)+1-i}$. Since $(x + \varepsilon_x)(y + \varepsilon_y) - xy = 2^{-(i+j)} f_x(i) f_y(j) - xy$ and noting $\mathsf{msd}(x) = j - n - 3$, $\mathsf{msd}(y) = i - n - 3$, we have

$$\left|2^{-(i+j)} f_x(i) f_y(j) - xy\right| < 2^{-(n+1)} \tag{3.5}$$

Now, rewriting r using the definition of $\lfloor \cdot \rceil$, we obtain

$$\begin{aligned} & r - \frac{1}{2} \leq \frac{f_x(i) f_y(j)}{2^{i+j-n}} < r + \frac{1}{2} \\ \Leftrightarrow \quad & \left|\frac{f_x(i) f_y(j)}{2^{i+j-n}} - r\right| \leq 2^{-1} \\ \Leftrightarrow \quad & \left|2^{-(i+j)} f_x(i) f_y(j) - 2^{-n} r\right| \leq 2^{-(n+1)}. \end{aligned} \tag{3.6}$$

Finally, by the triangle inequality with (3.6) and (3.5),

$$\begin{aligned} |2^{-n} r - xy| &\leq \left|2^{-n} r - 2^{-(i+j)} f_x(i) f_y(j)\right| + \left|2^{-(i+j)} f_x(i) f_y(j) - xy\right| \\ &< 2^{-(n+1)} + 2^{-(n+1)} \end{aligned}$$

such that $|2^{-n} r - xy| < 2^{-n}$. Hence $f_{xy}(n) = r$ represents the multiplication xy. □

3.4 Division

3.4.1 Inversion

Algorithm 3.8. If a real number $x \neq 0$ is represented by the approximation function f_x, then its *inverse* x^{-1}, i.e. $\frac{1}{x}$, is represented by the approximation function $f_{x^{-1}}$ that satisfies

$$\forall n \in \mathbb{Z} \,.\, f_{x^{-1}} = \begin{cases} 0 & \text{if } n < \mathsf{msd}(x) \\ \left\lfloor \dfrac{2^{2n-2\mathsf{msd}(x)+3}}{f_x(n-2\mathsf{msd}(x)+3)} \right\rfloor & \text{otherwise.} \end{cases}$$

Proof. We first consider the case when $n < \mathsf{msd}(x)$. By the definition of msd, we have $2^{\mathsf{msd}(x)-1} < |x|$. Since $2^{\mathsf{msd}(x)-1} > 0$, we may deduce that

$$\left|\frac{1}{x}\right| < \frac{1}{2^{\mathsf{msd}(x)-1}}.$$

Because $\mathsf{msd}(x) > n$, we have $\mathsf{msd}(x) - 1 > n - 1$ so that $\mathsf{msd}(x) - 1 \geq n$ since $\mathsf{msd}(x) \in \mathbb{Z}$. Therefore

$$\left|2^{-n} \cdot 0 - \frac{1}{x}\right| < \frac{1}{2^{\mathsf{msd}(x)-1}} \leq \frac{1}{2^n}$$

and so $f_{x^{-1}}(n) = 0$ represents the real number $\frac{1}{x}$ for $n < \mathsf{msd}(x)$.

Let us now consider the second case. For the sake of notational brevity, we introduce $i = n - 2\mathsf{msd}(x) + 3$. Since $n \geq \mathsf{msd}(x)$, we have $\mathsf{msd}(x) + i \geq 3$. By the definition of msd and f_x,

$$\begin{aligned} & 2^{\mathsf{msd}(x)-1} < |x| < 2^{-i}\left(|f_x(i)| + 1\right) \\ \Leftrightarrow \quad & 2^{\mathsf{msd}(x)+i-1} < |f_x(i)| + 1 \\ \Leftrightarrow \quad & 2 < 2^{\mathsf{msd}(x)+i-1} < |f_x(i)| + 1 \end{aligned}$$

such that $|f_x(i)| > 1$. As we have seen before, by the definition of f_x,

$$2^{-i}(|f_x(i)| - 1) < |x| < 2^{-i}(|f_x(i)| + 1). \tag{3.7}$$

Since $|f_x(i)| > 1$, we have $2^{-i}(|f_x(i)| - 1) > 0$, and clearly $2^{-i}(|f_x(i)| + 1) > 0$. Therefore we can deduce from (3.7) that

$$\begin{aligned} & \frac{2^i}{|f_x(i)| + 1} < \frac{1}{|x|} < \frac{2^i}{|f_x(i)| - 1} \\ \Leftrightarrow \quad & 2^{-n}\frac{2^{n+i}}{|f_x(i)| + 1} < \frac{1}{|x|} < 2^{-n}\frac{2^{n+i}}{|f_x(i)| - 1} \end{aligned} \tag{3.8}$$

For any $q \in \mathbb{Q}$ we have $\lfloor q \rfloor \leq q \leq \lceil q \rceil$ so that

$$2^{-n}\left\lfloor \frac{2^{n+i}}{|f_x(i)| + 1} \right\rfloor < \frac{1}{|x|} < 2^{-n}\left\lceil \frac{2^{n+i}}{|f_x(i)| - 1} \right\rceil.$$

Now, let us write

$$\alpha = \frac{2^{n+i}}{|f_x(i)|+1}, \qquad \beta = \frac{2^{n+i}}{|f_x(i)|-1}, \qquad \xi = \left\lfloor \frac{2^{n+i}}{|f_x(i)|} \right\rceil.$$

Clearly $\alpha < \beta$ and $\lfloor \alpha \rfloor \le \xi \le \lceil \beta \rceil$. Once again, by the definition of f_x and msd, we have

$$2^{\mathsf{msd}(x)-1} < |x| < 2^{-i}(|f_x(i)|+1)$$

so that $|f_x(i)| > 2^{\mathsf{msd}(x)+i-1} - 1$. Since $\mathsf{msd}(x) + i \ge 3$, we get $|f_x(i)| \ge 2^{\mathsf{msd}(x)+i-1}$ and so

$$\frac{2^{n+i}}{|f_x(i)|} \le \frac{2^{n+i}}{2^{\mathsf{msd}(x)+i-1}} = \frac{2^{2(\mathsf{msd}(x)+i-1)-1}}{2^{\mathsf{msd}(x)+i-1}} = \frac{2^{\mathsf{msd}(x)+i-1}}{2} \le \frac{|f_x(i)|}{2}.$$

Now, by the definition of $\lfloor q \rceil$ for $q \in \mathbb{Q}$,

$$\xi - \frac{1}{2} \le \frac{2^{n+i}}{|f_x(i)|} < \xi + \frac{1}{2} \quad \Rightarrow \quad \xi - 1 \le \frac{2^{n+i}}{|f_x(i)|} - \frac{1}{2}. \tag{3.9}$$

Thus we have

$$\begin{aligned}(\xi-1)(|f_x(i)|+1) &\le \left(\frac{2^{n+i}}{|f_x(i)|} - \frac{1}{2}\right)(|f_x(i)|+1) \\ &= 2^{n+i} + \frac{2^{n+i}}{|f_x(i)|} - \frac{|f_x(i)|}{2} - \frac{1}{2} \\ &\le 2^{n+i} - \frac{1}{2} < 2^{n+i}.\end{aligned}$$

so that

$$\xi - 1 < \frac{2^{n+i}}{|f_x(i)|+1}.$$

Similarly, using the $\xi + \frac{1}{2}$ side of (3.9), we find

$$\frac{2^{n+i}}{|f_x(i)|-1} < \xi + 1$$

and therefore $\xi - 1 < \alpha < \beta < \xi + 1$. Multiplying through by 2^{-n}, we obtain

$$2^{-n}(\xi-1) < 2^{-n}\alpha < 2^{-n}\beta < 2^{-n}(\xi+1).$$

We assume $f_x(i) > 0$ such that $|f_x(i)| = f_x(i)$, $x > 0$ and $|x| = x$. It is left to the reader to confirm that equivalent reasoning holds if we instead assume $f_x(i) \le 0$ such that we replace $|f_x(i)| = -f_x(i)$ and $|x| = -x$ in our above formulation. Then, by (3.8) with $|x| = x$, we have

$$\begin{aligned}& 2^{-n}(\xi-1) < 2^{-n}\alpha < \frac{1}{x} < 2^{-n}\beta < 2^{-n}(\xi+1) \\ \Rightarrow\quad & 2^{-n}(\xi-1) < \frac{1}{x} < 2^{-n}(\xi+1) \\ \Leftrightarrow\quad & -2^{-n} < \frac{1}{x} - 2^{-n}\xi < 2^{-n}\end{aligned}$$

such that $f_{x^{-1}}(n) = \xi$ satisfies $\left|2^{-n} f_{x^{-1}}(n) - \frac{1}{x}\right| < 2^{-n}$ for $x > 0$ and we find the same holds for $x \leq 0$. □

3.4.2 Division

Since $x/y = x \cdot y^{-1}$ for any two real numbers x and $y \neq 0$, we trivially define division $f_{x/y}$ in terms of multiplication and inversion, such that $f_{x/y} = f_{x \cdot y^{-1}}$.

3.5 Square root

Algorithm 3.9. If a real number $x \geq 0$ is represented by the approximation function f_x, then its *square root* $\sqrt{x}$ is represented by the approximation function $f_{\sqrt{x}}$ that satisfies

$$\forall n \in \mathbb{Z} \,.\, f_{\sqrt{x}} = \left\lfloor \sqrt{f_x(2n)} \right\rfloor.$$

Proof. By the definition of f_x,

$$\left|2^{-2n} f_x(2n) - x\right| < 2^{-2n}. \tag{3.10}$$

Since $x \geq 0$, we have $f_x(2n) \geq 0$. If $f_x(2n) = 0$, then $|x| < 2^{-2n}$ such that $\left|\sqrt{x}\right| < 2^{-n}$. Clearly $f_{\sqrt{x}}(n) = \left\lfloor \sqrt{f_x(2n)} \right\rfloor = 0$ and so $\left|2^{-n} f_{\sqrt{x}}(n) - \sqrt{x}\right| = \left|\sqrt{x}\right| < 2^{-n}$. In the other case, $f_x(2n) \geq 1$ and so by (3.10)

$$\begin{aligned} & -2^{-2n} < x - 2^{-2n} f_x(2n) < 2^{-2n} \\ \Leftrightarrow \quad & 2^{-2n} \left(f_x(2n) - 1\right) < x < 2^{-2n} \left(f_x(2n) + 1\right) \\ \Leftrightarrow \quad & 2^{-n} \sqrt{f_x(2n) - 1} < \sqrt{x} < 2^{-n} \sqrt{f_x(2n) + 1}. \end{aligned}$$

We note that $\forall m \in \mathbb{Z}, m \geq 1$ we have $\left\lfloor \sqrt{m} \right\rfloor - 1 \leq \sqrt{m-1}$ and $\sqrt{m+1} \leq \left\lfloor \sqrt{m} \right\rfloor + 1$. Since $f_x(2n) \geq 1$,

$$\left\lfloor \sqrt{f_x(2n)} \right\rfloor - 1 \leq \sqrt{f_x(2n) - 1} \qquad \text{and} \qquad \sqrt{f_x(2n) + 1} \leq \left\lfloor \sqrt{f_x(2n)} \right\rfloor + 1$$

so that

$$\begin{aligned} & 2^{-n} \left(\left\lfloor \sqrt{f_x(2n)} \right\rfloor - 1 \right) < \sqrt{x} < 2^{-n} \left(\left\lfloor \sqrt{f_x(2n)} \right\rfloor + 1 \right) \\ \Leftrightarrow \quad & -2^{-n} < \sqrt{x} - 2^{-n} \left\lfloor \sqrt{f_x(2n)} \right\rfloor < 2^{-2n}. \end{aligned}$$

Finally, because $f_{\sqrt{x}}(n) = \left\lfloor \sqrt{f_x(2n)} \right\rfloor$ we have $\left|2^{-n} f_{\sqrt{x}}(n) - \sqrt{x}\right| < 2^{-n}$. □

CHAPTER 4

ELEMENTARY FUNCTIONS

In Chapter 3, we developed suitable algorithms for the basic arithmetic operations over our real number representation. This chapter builds upon these operations to implement higher-level transcendental functions, and finally we look at how we can compare real numbers.

A *transcendental function* is a function that does not satisfy a polynomial[1] equation whose coefficients are themselves polynomials, in contrast to an algebraic function, which does so. That is, a transcendental function is a function that cannot be expressed in terms of a finite sequence of the algebraic operations of addition, multiplication and root extraction. Examples of transcendental functions include the logarithm and exponential function, and the trigonometric functions (sine, cosine, hyperbolic tangent, secant, and so forth).

4.1 Power series

Many common transcendental functions, such as $\sin, \cos, \exp, \ln$ and so forth, can be expressed as power (specifically, Taylor) series expansions. Let $F : \mathbb{R} \to \mathbb{R}$ be a transcendental function defined by the power series

$$F(x) = \sum_{i=0}^{\infty} a_i x^{b_i}.$$

where $(a_i)_{i \in \mathbb{N}}$ is an infinite sequence of rationals. If $x \in \operatorname{dom} F \subseteq \mathbb{R}$, then $F(x)$ is represented by the approximation function $f_{F_x} = \mathsf{powerseries}(\{a_i\}, x, k, n)$ that satisfies

$$|2^{-n} f_{F_x}(n) - F(x)| < 2^{-n}$$

assuming these conditions also hold:

1. $|x| < 1$,
2. $|a_i| \leq 1$ for all i,
3. $\left|\sum_{i=k+1}^{\infty} a_i x^i\right| < 2^{-(n+1)}$.

[1] A *polynomial function* is a function that can be expressed in the form $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$ for all arguments x, where $n \in \mathbb{N}$ and $a_0, a_1, \ldots, a_n \in \mathbb{R}$.

The powerseries algorithm generates an approximation function $\mathbb{Z} \to \mathbb{Z}$ representing the function application $F(x)$—that is, the partial sum of k terms from the power series specified by a_i, b_i, x representing $F(x)$ to the given precision n. We aim to specify powerseries and prove the correctness of the general power series it generates. This general approach means it is then trivial to prove the correctness of any function expressible as a power series—we need only demonstrate that the conditions 1. and 2. hold.

The first two conditions ensure suitably fast convergence of the series. The third condition ensures the series truncation error is less than half of the total required error, 2^{-n}. We shall use the remaining $2^{-(n+1)}$ error in the calculation of

$$\sum_{i=0}^{k} a_i x^i. \tag{4.1}$$

Rather than calculating each x^i in (4.1) directly with exact real arithmetic using our previous methods, we cumulatively calculate each x^{i+1} using our previously calculated x^i. We construct a sequence (X_i) of length $k+1$ estimating each x_i to a precision of n', where n' is chosen so that the absolute sum of the errors is bounded by $2^{-(n+1)}$. Then the error for the entire series is $k \cdot 2^{-n'} + \left|\sum_{i=k+1}^{\infty} a_i x^i\right| < 2^{-(n+1)} + 2^{-(n+1)} = 2^{-n}$. Clearly when estimating x^i we need to evaluate the argument x to an even higher level of precision, n'', where $n'' > n'$. Specifically, based on what we learnt in section 3.2.1 whilst reasoning about addition, we take $n'' = n' + \lfloor \log_2 k \rfloor + 2$. We can construct (X_i) using only the value of $f_x(n'')$ as

$$X_0 = 2^{n''}, \quad X_{i+1} = \left\lfloor X_i \cdot 2^{-n''} f_x(n'') \right\rceil.$$

such that $\left|2^{-n''} X_i - x_i\right| < 2^{-n'}$. Computing $\sum_{i=0}^{k} a_i x^i$ involves summing over $k+1$ terms, each of have an error of $\pm 2^{-(n+\lfloor \log_2 k \rfloor + 2)}$. Therefore the error in the sum is bounded by $\pm 2^{-(n+1)}$. So we have $n' = n + \lfloor \log_2 k \rfloor + 2$ and thus $n'' = n' + \lfloor \log_2 k \rfloor + 2 = n + 2\lfloor \log_2 k \rfloor + 4$. Then we use the algorithm defined and proved in [16] where initially we have $T = \lfloor a_0 X_0 \rceil$ for terms, $S = T$ for our finite sum of terms, and $E = 2^{\lfloor \log_2 k \rfloor}$:

```
powerseries({a_i}, x, k, n):
    do
        do
            α_{i+1} = ⌊α_i 2^{-n''} f_x(n'')⌉;
            i = i + 1;
            T = ⌊a_i X_i⌉;
        while (a_i == 0 and i < t)
        S = S + T;
    while (T > E)
    return ⌊ S / 2^{n+n'} ⌉
end
```

So the approximation function $f_{F_x} = \left\lfloor \frac{S}{2^{n+n'}} \right\rceil$.

4.2 Logarithmic functions

4.2.1 Exponential function

The exponential function e^x, or $\exp(x)$, for any real number x can be defined by the power series

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots.$$

We represent $\exp x$ for any real number x as the accuracy function

$$\begin{aligned} f_{\exp_x}(n) &= \mathsf{powerseries}(\{a_i\}, x, k, n) \\ \text{where} \qquad a_i &= \frac{1}{i!} \text{ for } i = 0, 1, 2, \ldots, \\ x &= x, \\ k &= \max(5, n). \end{aligned}$$

To prove the correctness of $f_{\exp_x}$, all we need do is show conditions 1, 2 and 3 from the power series approximation definition hold. It is trivially clear that $|a_i| \leq 1$ for all $i \in \mathbb{N}$ so that condition 1 is satisfied. Assume for now that condition 2 holds so that $|x| < 1$. To show condition 3 is satisfied, we note that the truncation error is less than the k-th term, $\frac{x^k}{k!}$. Therefore, for $k \geq 5$, we have

$$\left| \sum_{i=k+1}^{\infty} \frac{x^i}{i!} \right| < \left| \frac{x^k}{k!} \right| < \frac{1}{k!} \leq \frac{1}{2^{n+1}}.$$

Let us return to condition 2. Rather than requiring $|x| < 1$, we shall enforce a stricter restriction, $|x| \leq \frac{1}{2}$, since the exponential series converges extremely quickly in this range. For any real number x outside of the range $[-\frac{1}{2}, \frac{1}{2}]$, the identity $\exp(x) = \exp\left(\frac{x}{2} + \frac{x}{2}\right) = \left(\exp\left(\frac{x}{2}\right)\right)^2$ allows us to calculate $\exp(x)$ using $\exp\left(\frac{x}{2}\right)$. Clearly repeated application of this identity strictly scales the argument until it is in the range $[-\frac{1}{2}, \frac{1}{2}]$, since there always exists a $k \in \mathbb{N}$ such that $\left|\frac{x}{2^k}\right| \leq \frac{1}{2}$. Thus, we can represent the efficient computation of $\exp(x)$ for any real number x in terms of $\exp\left(\frac{x}{2k}\right)$, raised to the power k. We refer to this technique as *range reduction* and make repeated use of it in the remainder of this chapter.

4.2.2 Natural logarithm

The natural logarithm $\ln x$ of a real number $x > 0$ can be defined in terms of the Taylor series expansion of $\ln(1 + x)$:

$$\ln(1 + x) = \sum_{i=1}^{\infty} \frac{(-1)^{i+1} x^i}{i} = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots \quad \text{for } -1 < x \leq 1.$$

We represent $\ln x$ for any real number $|x| < 1$ as the accuracy function

$$f_{\ln_x}(n) = \mathsf{powerseries}(\{a_i\}, x, k, n)$$

$$\text{where} \qquad \begin{aligned} a_i &= \frac{(-1)^{i+1}}{i} \text{ for } i = 1, 2, 3, \ldots, \\ x &= x, \\ k &= \max(1, n). \end{aligned}$$

Again, it is clear that $|a_i| \leq 1$ for all $i \in \mathbb{N}$ so that condition 1 is satisfied. To show that condition 3 holds, we observe the truncation error is less than the k-th term, $\frac{(-1)^{i+1}}{i}$. Therefore, for $k \geq 1$, we have

$$\left| \sum_{i=k+1}^{\infty} \frac{(-1)^{i+1} x^i}{i} \right| < \left| \frac{(-1)^{k+1} x^k}{k} \right| < \frac{(-1)^{k+1}}{k} \leq \frac{1}{2^{n+1}}.$$

For $|x| < \frac{1}{2}$, The series converges rapidly for $|x| < \frac{1}{2}$, and so we aim to scale the argument into this range. We consider a number of cases in order to fully define $\ln x$ over the positive reals in terms of this series:

1. If $\frac{1}{2} < x < \frac{3}{2}$, then $\frac{1}{2} < x - 1 < \frac{1}{2}$ so that we may define $\ln x = \ln(1 + (x - 1))$.
2. If $x \geq \frac{3}{2}$, then there exists a $k \in \mathbb{N}$ such that $\frac{1}{2} < \frac{x}{2^k} < \frac{3}{2}$. We note $\ln\left(\frac{x}{2^k}\right) = \ln x - k \ln 2$ so that we can define $\ln x = \ln\left(1 + \left(\frac{x}{2k} - 1\right)\right) + k \ln 2$.[2]
3. If $0 < x \leq \frac{1}{2}$, then $\frac{1}{x} \geq 2$ so that we can use the identity $\ln x = -\ln \frac{1}{x}$ and then apply case 2 above.

4.2.3 Exponentiation

Since $\exp$ and $\ln$ are inverse functions of each other, and recalling the logarithmic identity $\ln x^y = y \cdot \ln x$, we can define the computation of arbitrary real powers of a real number x as

$$x^y = \exp\left(y \cdot \ln x\right).$$

Note how we could have defined the square root operation derived in 3.5 by setting $y = \frac{1}{2}$ such that $\sqrt{x} = \exp\left(\frac{1}{2} \ln x\right)$. More generally, we implement the nth root of a real number x as the function $\sqrt[n]{x} = \exp\left(\frac{1}{n} \ln x\right)$.

4.3 Trigonometric functions

4.3.1 Cosine

The cosine of any real number x can be expressed as the Taylor series expansion

$$\cos x = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i)!} x^{2i} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots.$$

[2] We define the constant $\ln 2$ later in section 4.4.2.

We represent $\sin x$ for any real number x as the accuracy function

$$f_{\cos_x}(n) = \mathsf{powerseries}(\{a_i\}, x, k, n)$$

$$\begin{aligned} \text{where} \qquad a_i &= \frac{(-1)^i}{(2i)!} \text{ for } i = 0, 1, 2, \ldots, \\ x &= x^2, \\ k &= \max(2, n). \end{aligned}$$

It is trivially obvious that $|a_i| \leq 1$ for all $i \in \mathbb{N}$. To show condition 3 holds, we note that the truncation error will be less than the absolute value of the k-th term, $\frac{x^{2k}}{(2k)!}$. Therefore, for $k \geq 2$, we have

$$\left| \sum_{i=k+1}^{\infty} \frac{(-1)^i}{(2i)!} x^{2i} \right| < \left| \frac{x^{2k}}{(2k)!} \right| < \frac{1}{(2k)!} \leq \frac{1}{2^{n+1}}$$

so that condition 3 holds. Finally, we can perform range-reduction to ensure the argument x falls in the range $(-1, 1)$ as required by condition 1. We use the double-angle formula

$$\cos x = 2\cos^2\left(\frac{x}{2}\right) - 1.$$

4.3.2 Other trigonometric functions

We can define and prove the correctness of the inverse sine function $\arcsin$ in a similar way. The same goes for most of the other trigonometric functions, whether they are standard, inverse or hyperbolic. However, once we have defined some basic functions, we can use trigonometric identities that involve these functions to determine a number of additional trigonometric functions. This saves us from deriving their power series and proving their correctness. Table 4.1 overleaf lists a number of trigonometric functions defined in terms of functions we have previously reasoned about.

4.4 Transcendental constants

4.4.1 Pi

It is well known that the arctan Taylor series expansion of the reciprocal of an integer

$$\arctan\frac{1}{x} = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)x^{2i+1}}, \qquad x \in \mathbb{Z}, x \geq 2$$

convergences very rapidly and it is easy to show that this expansion satisfies our three conditions and so we can represent it. This power series can be used to quickly and

Function	Identity definition
Hyperbolic sine	$\sinh x = \dfrac{e^x - e^{-x}}{2}$
Hyperbolic cosine	$\cosh x = \dfrac{e^x + e^{-x}}{2}$
Hyperbolic tangent	$\tanh x = \dfrac{\sinh x}{\cosh x} = 1 - \dfrac{2}{e^{2x}+1}$
Inverse hyperbolic sine	$\operatorname{arsinh} x = \ln\left(x + \sqrt{x^2+1}\right)$
Inverse hyperbolic cosine	$\operatorname{arcosh} x = \ln\left(x + \sqrt{x^2-1}\right)$
Inverse hyperbolic tangent	$\operatorname{artanh} x = \frac{1}{2}\ln\dfrac{1+x}{1-x}$
Tangent	$\tan x = \dfrac{\sin x}{\cos x}$
Sine	$\sin x = \cos\left(\frac{\pi}{2} - x\right)$
Inverse cosine	$\arccos x = \dfrac{\pi}{2} - \arcsin x$
Inverse tangent	$\arctan x = \operatorname{sgn} x \cdot \arccos \dfrac{1}{\sqrt{x^2+1}}$

Table 4.1: Trigonometric identities

easily compute π. We begin by noting $\frac{\pi}{4} = \arctan 1$. By the familiar double-angle formulae,

$$\begin{aligned} & \tan 2\theta = \frac{2\tan\theta}{1-\tan^2\theta} \\ \Rightarrow\quad & \tan(2\arctan\theta) = \frac{2\theta}{1-\theta^2} \\ \Rightarrow\quad & 2\arctan\theta = \arctan\left(\frac{2\theta}{1-\theta^2}\right) \end{aligned}$$

and recalling the difference identity,

$$\arctan\alpha - \arctan\beta = \arctan\left(\frac{\alpha-\beta}{1+\alpha\beta}\right).$$

Therefore, we have

$$4\arctan\left(\frac{1}{5}\right) = 2\arctan\left(\frac{2\left(\frac{1}{5}\right)}{1-\left(\frac{1}{5}\right)^2}\right) = \arctan\left(\frac{2\left(\frac{5}{12}\right)}{1-\left(\frac{5}{12}\right)^2}\right) = \arctan\left(\frac{120}{119}\right)$$

and

$$\arctan\left(\frac{120}{119}\right) - \arctan\left(\frac{1}{239}\right) = \arctan\left(\frac{\frac{120}{129} - \frac{1}{239}}{1+\left(\frac{120}{119}\right)\left(\frac{1}{239}\right)}\right) = \arctan 1.$$

Hence, we can efficiently compute π using the reciprocal integer Taylor series expansion of arctan with

$$\pi = 16\arctan\left(\frac{1}{5}\right) - 4\arctan\left(\frac{1}{239}\right).$$

Alternatively, we could attempt to use the more-recent Bailey-Borwein-Plouffe (BBP) identity

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i}\left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6}\right)$$

derived in [2], which has been shown to be more efficient in terms of correct digits calculated per second.

4.4.2 ln 2

We can compute the important mathematical constant $\ln 2$ using the Taylor series expansion for $\ln(1+x)$ by noting that

$$\begin{aligned} & 7\ln\left(1+\frac{1}{9}\right) - 2\ln\left(1+\frac{1}{24}\right) + 3\ln\left(1+\frac{1}{80}\right) \\ = \; & \ln\left(\frac{10^7 \cdot 24^2 \cdot 81^3}{9^7 \cdot 25^2 \cdot 80^3}\right) \\ = \; & \ln 2. \end{aligned}$$

4.4.3 e

Trivially, we may represent the constant e using our representation of the exponential function from section 4.2.1. We have

$$e = \exp(1) = \left(\exp\left(\tfrac{1}{2}\right)\right)^2.$$

4.5 Comparisons

Comparison presents a slight problem, since it is undecidable on real numbers. If we consider the equality comparison between two real numbers, both π, represented as infinite decimal streams, no algorithm that determines whether or not they are equal will ever be able. We need a practical solution to this problem, in our implementation we provide a comparison operator that identifies two values as equal if they are within a specified tolerance. We refer to this as *relative comparison* and it is guaranteed to terminate but not guaranteed to be correct. Alternatively, we provide a comparison operator that always returns the correct result but could potentially loop forever. This is called the *absolute comparison*. Proofs for these comparisons can be found in [16].

PART II

IMPLEMENTATION

CHAPTER 5

THE EXACTLIB LIBRARY

5.1 Why C++?

One of the major aims of my project was to produce a library for exact real arithmetic that could be easily imported and used by applications. We also wanted to bring exact real arithmetic to a more mainstream programming audience, which ultimately lead us to an imperative, rather than functional, language. Specifically, we chose C++, and in this section we outline a number of reasons why.

It is well known that code written in C++ compiles to very efficient executables. Despite its efficiency, C++ remains flexible, particularly through its extremely powerful template system and support of full object-oriented programming. Especially important to us is the technique of template metaprogramming, where templates are used by the compiler to generate temporary source code before merging them with the rest of the source code for compilation. As we shall see, this affords a degree of power simply unachievable by a language such as Java. Unlike many popular languages, such as Java, C# and Python, C++ grants the programmer full control over the heap. This is useful when we come to optimise expression trees and enables future work to perhaps improve performance.

A focus on ease of use means it is important for our library to seamlessly integrate with the language. In fact, from a design point of view, we aim to produce a library suitable for inclusion in the language's standard library, as an alternative to using the language's floating-point type(s). Seamless integration with the language means the addition of two real numbers should look identical to the addition of two numbers of a built-in integral or floating-point type. C++ provides facilities such as operator overloading that helps us achieve just this, allowing us to create data types that seamlessly integrate with the language. One of the major reasons for disregarding Java is its complete lack of support for operator overloading. We believe integration is important, since most existing arithmetic libraries do not fit well at all with their respective languages. For example, iRRAM [23], written in C++, requires explicit initialisation—including setting precision levels—and release of the library before and after use, and cannot use standard C++ streams for input and output. It would require tedious rewriting to use iRRAM in an existing code base.

Finally, C++ is one of the most popular languages around today—if not *the* most popular. Compilers exist for nearly every platform and its use is ubiquitous across science, engineering and industry. So, not only will a library written in C++ have a

large potential audience for its use, but it will also coexist in an active community of trained mathematicians and programmers that may help further work on the library.

The downside of C++ is that, from a design perspective, it is starting to show its age. Newer languages like C# and Java have a more flexible, cleaner and simpler syntax. They fix many of the mistakes made during the design and evolution of C++, and are generally a lot easier to use when working with a code-base of any substantial size. C++ is also a notoriously large and complex beast—something I can attest to as I near the end of my project!

5.1.1 Naming conventions

User code in modern object-oriented languages, such as C++, C# and Java, commonly follows a relatively standard naming convention for identifiers. Pascal casing is used for classes, namespaces, interfaces, and so forth, so that a class might be named ExpressionFactory. Otherwise, Camel casing is used for local and instance variables, parameters and methods—for example, a variable could be named isReadable, or a function removeAll.

The C++ standard library, called the Standard Template Library (STL), adheres to a rather different style, inherited from its C ancestry. Since we wish for ExactLib to integrate nicely with C++, we follow the STL's naming conventions. The popular Boost [11] collection of C++ libraries, a number of which are candidates for inclusion in the upcoming C++0x standard, follow the same naming convention. Specifically, we use a lowercase/underscore naming convention for nearly every identifier apart from template parameters, which use Pascal casing. For example, a class and function might be called reverse_iterator and for_each, whilst a template parameter could be UnaryFunction. Since primitive types such as int and double follow the same naming convention, it would look out of place if we were to name our real number type Real. Whilst it may be more aesthetically pleasing, it is nonetheless necessary to follow the style of the STL if we wish for our library to integrate well.

5.2 Integer representation

Recalling our scaled functional representation from Definition 2.4, a real number x is represented by a function f_x that satisfies

$$|2^{-n} f_x(n) - x| < 2^{-n}.$$

To evaluate a real number to a required precision, we pass f_x an integer n specifying the desired accuracy. We scale the value returned from $f_x(n)$ by dividing by 2^n. The use of n as an *exponent* during scaling means that very large numbers can be achieved for relatively small values of n. Therefore, in practice, we do not require more than a 32-bit integer to store n before computational time complexity becomes the restrictive factor. In C++, a suitable type that we shall use is the built-in int. However, the integer returned by $f_x(n)$ could potentially be huge. To achieve exact real arithmetic through

theoretically[1] infinite-precision calculations, we shall require an arbitrary-precision integer type.

A number of C++ libraries providing arbitrary-precision integer arithmetic exist—an up-to-date list can be found on Wikipedia. The GNU Multiple Precision Library (GMP) [18], actually written in C but for which a C++ wrapper is provided, is generally accepted as the fastest and most comprehensive. However, one of the major objectives of ExactLib is independence—we desire a standalone library for ease of installation, with no dependencies other than the STL. Therefore, we provide our own simple implementation of arbitrary-precision integer arithmetic, but allow the user to specify a different arbitrary-precision library—such as the more efficient and stable GMP—at compile-time if they have it installed. Our simple big-integer arithmetic is based entirely on the C++ xintlib library, which crucially is based on solid theory that proves its correctness [28]. In any non-trivial use, it is advisable to install and use the GMP, but our simple implementation lowers the barrier of entry for quick and easy use.

We implement our real arithmetic using an abstract class big_integer that acts as an interface, specifying all the operations an arbitrary-precision integer representation must support—addition, multiplication, absolute value, rounding up/down, and so on. Two concrete classes simple_integer and gmp_integer inherit from big_integer, with the former realising our simple implementation and the latter being a wrapper for the GMP library. If a user wishes to use a different arbitrary-precision integer arithmetic library, all they need do is write one that realises the big_integer interface or write a conforming wrapper for an existing library. For example, the provided gmp_integer wrapper inherits big_integer and realises the operator+ method using GMP's mpz_add function to implement the addition of two GMP integers:

```
gmp_integer gmp_integer::operator+(const gmp_integer& rhs)
{
   gmp_integer result;
   mpz_add(result.value, this->value, rhs.value);
   return result;
}
```

Here, the value field is an mpz_t object, which is GMP's representation of an arbitrary-precision integer.

5.3 Library overview and usage

Before we describe the internals of the ExactLib library, it helps to briefly demonstrate how the library is used. This will provide scope for our later, more technical discussion.

ExactLib is packaged in the exact namespace and the main type it exposes is the real type. A real object represents exactly a real number, and finite approximations can be taken to any precision the user desires. We have designed the real class to resemble a

[1] Limited only by time and hardware constraints.

built-in primitive type as closely as possible—arithmetic operations and comparisons on reals are supported as overloaded operators (+, *, <, >=, etc.) in much the same way they are for built-in types like int and double. Similarly, functions like sin, abs and exp are provided for reals in the same way they are for integral and floating-point types in the standard C++ cmath header file. We support outputting and inputting to and from C++ streams, so that we may write and read reals to and from the standard input or some file in the same way we would with any other variable.

5.3.1 Constructing real numbers

A real object representing a real number can be constructed in a number of ways:

- From an integer, e.g. real x = 10. This is the most basic representation of a real and uses Algorithm 3.1 for the functional representation of integers.
- From a string, e.g. real x = "45.678". This represents exactly the rational number 45.678 by parsing the string and expressing it as a fraction $45678/1000$. The real created is the result of dividing two real integer objects, i.e. real(45678) / real(1000).
- From an existing real, e.g. real x = y or real x(y). This creates a deep copy of the real object y, and the two variables are independent of each other.
- From a provided library constant, e.g. real::pi. This is predefined by ExactLib and exactly represents π as the series expansion derived in section 4.4.1. Transcendental constants are defined as public static constants in the real class so that they are not dumped unqualified in the global namespace.
- Via operations on reals, e.g. real(10) + real(5), and functions, e.g. real::sin(2). Similarly to the transcendental constants, functions like sin are scoped to the real class to prevent namespace clashes with their inexact namesakes defined in C++'s cmath header file.

5.3.2 Evaluating real numbers

The real class provides a method evaluate(int n) that evaluates the real number to a precision n and returns the approximation's numerator as an unscaled big_integer. However, this large integer value is of little use in terms of a outputting a user-friendly representation. Therefore, our library overloads the << and >> operators so that real objects may output to and read from the standard C++ input/output streams (iostreams). This allows us to print a decimal representation of a real number to a stream. By default, numbers are outputted to 2 decimal places, but we support the use of the iostream manipulator std::setprecision with reals to set the precision of the output. Once a precision level has been set, this level will be used for the remaining output, but we can use the std::precision manipulator mid-stream to switch between different numbers of decimal places. For example, the code

```
real x;
real pi = real::pi;
real e = real::e;

std::cout << "> ";
std::cin >> x;

std::cout << x << std::endl
         << std::setprecision(40) << pi << std::endl
         << std::setprecision(20) << e << std::endl;
         << real::ln2 << std::endl;
```

will output the user-provided number to 2 decimal places, π to 40 decimal places, e to 20 decimal places and $\ln 2$ to 20 decimal places:

```
> 103.212
103.21
3.1415926535897932384626433832795028841972
2.71828182845904523536
0.69314718055994530942
```

Clearly providing an iostream manipulator for reals over streams is a lot more flexible and design-wise more appropriate than providing a static method on real that sets its output precision.

In addition to evaluating a real number for output, we overload cast operators so that a real number can be cast to double, float and signed/unsigned long and int types. Naturally, the user should take care and be aware during casting due to the high potential to lose precision.

5.3.3 Performing operations and comparing real numbers

The real class overloads a number of arithmetic operators, allowing the same style of arithmetic manipulation as the primitive types. Elementary functions are provided as public static methods on the real class, e.g. real::sqrt(const real& x). This prevents namespace conflicts with the standard cmath mathematical functions, since we must qualify our functions with the class scope real::. We discuss the operations and functions on reals in detail in the following sections, since they arguably form the centrepiece of our project.

As we explored in section 4.5, comparing real numbers is slightly cumbersome. There are two types of comparison that can be performed: absolute and relative. The comparison operators <, >, ==, <=, etc. are overloaded for real objects and use the static method real::comparisons_type(comparison t), where comparison is an enumeration type with two enumerators comparison::absolute and comparison::relative, to set whether absolute or relative comparison should be performed by the overloaded comparison operators. By default, we use absolute comparison since this is the most practical and is fit for most purposes.

5.3.4 Higher-order functions

Finally, ExactLib explores an alternative way to represent computations over real numbers. Inspired by functional programming languages, we wrap our operations and functions over the reals as C++ functors, so that they act as first-class citizens. To prevent them clashing with their traditional namesakes, we place these functors in a separate namespace, exact::lambda. To demonstrate, assume we have issued appropriate namespace using directives so that the functors can be used unqualified:

```
typedef function<real, real> real_function;

real_function f = compose(sin, sqrt);
real x = f(pi);

real_function logistic = 4 * id * (1 - id);
for (int i = 0; i < 20; ++i) {
   x = logistic(x);
   std::cout << x << std::endl;
}
```

Then after the first function application, x is the result of the equivalent traditional computation real::sin(real::sqrt(real::pi)) and represents the exact real number $\sin\sqrt{\pi}$. The function logistic represents the well-known logistic iteration $x_{i+1} = 4 \cdot x_i \cdot (1 - x_i)$ and illustrates how ExactLib makes performing iterations using exact real arithmetic easy. Recall that we have seen floating-point arithmetic fail whilst performing iterations, so the ease of performing them instead using ExactLib's concise higher-order functions is particularly useful.

5.4 Library internals

5.4.1 real and base_real classes

The main class exposed to users, real, is actually just a template specialisation in which we use our simple_integer implementation for arbitrary-precision integer arithmetic and the built-in int type for specifying precision:

```
template <class Result, class Arg = int>
class real { ... };

typedef base_real<simple_integer> real;
```

A user may choose to use the GMP integer library discussed earlier by changing the first template argument to gmp_integer, or by defining additional typedefs so that multiple real types with different arbitrary-precision integer arithmetic libraries may be used in the same application. This generic approach is seen throughout C++'s STL and the Boost libraries; for example, the ostream type, used for file and standard out-

put, is in fact a typedef for the generic basic_ostream class instantiated with template argument char.

The base_real class is a generic class in which we use C++ templates to specify the arbitrary-precision integer type and precision argument type[2] used in representing the reals. For simplicity's sake, in the remainder of this section we shall refer simply to real, although in general really we mean base_real.

The real class is the main type that users interact with and represents real numbers exactly. When an operation is performed upon a given real, a new real object is created containing information about the operation and references to the real object(s) representing the operation's argument(s). No evaluation of the actual real number is done—instead we build an expression tree, whose nodes represent operations, that describes the computation. Only when the user requests an approximation of some precision to the real number—either by calling the method evaluate(int n) or outputting the real to a stream—is the real number evaluated. A number of constructors are defined that allow users to construct real objects from int, float or string variables. A default constructor that creates a real object representing zero is also provided.

Whilst the real class is employed by users to create real numbers and perform exact calculations on them, it actually only provides an interface, or glue, between the user and the expression tree of functions representing a real number. The actual underlying structure is based on the abstract class functional_real, which encapsulates the notion of an approximation function as introduced in Definition 2.4, and the hierarchy of operations and functions its subclasses form. Every real object has a private field internal that points to its underlying functional_real expression tree of approximation functions. The functional_real type and its subclasses comprise the core of our real number representation and is described in section 5.4.2. All arithmetic operations and functions defined on real objects actually work with their internal member variable. For example, we overload the addition operator + for reals in the following manner:

```
real operator+(const real& x, const real& y)
{
   return real(new functional_add(x.internal, y.internal));
}
```

Here, functional_add is a concrete subclass of functional_real. The result is a new real object with its internal pointer set to a new expression tree constructed on the heap. This tree has the approximation-function representation of an addition, functional_add, at its root and the trees of arguments x and y as children. Note the real objects x and y are not affected—the new functional_add object links directly to their internal pointers. In this way, the expression trees of x and y are left unchanged and intact.

[2] As we have seen, int suffices and is the template parameter's default type, but we allow the user to specify otherwise for the purpose of generality.

Similarly, updating the value of a real variable using the assignment operator = should not cause the objects referenced in the assignee's expression tree to be destroyed, unless no other expression tree references them. This suggests we require some form of reference-counted pointers to keep track of the operation nodes we have allocated on the heap. Whilst on the surface this may suggest a language with built-in garbage collection such as Java or C# might have been more suitable, having full power over the heap means manipulation of expression trees to optimise evaluation may be possible in future work. We discuss expression trees and reference-counted pointers later in section 5.5.

5.4.2 functional_real and its subclasses

The functional_real class and the hierarchy it admits form the core of our implementation. Its name refers to our *functional* Cauchy representation, introduced in Definition 2.4, of *real* numbers as approximation *functions*. The functional_real class is abstract and encapsulates the notion of an approximation function. Operations and functions, like those defined in Chapters 3 and 4, are implemented as concrete subclasses of the functional_real type. We package operations into classes that inherit functionality so that the user can focus on providing the operation-specific function definition. Each operation (+, ∗, etc.) or function (sin, exp, etc.) applied to functional_reals creates a new object of the corresponding subclass type, e.g. functional_multiply. Since the new object generated belongs to a class derived from functional_real, we can continue to apply operations and functions on the new object, representing more and more complex expressions. In this sense, an expression tree—or directed acyclic graph (DAG)—is built, where nodes correspond to subclass objects containing operation semantics.

Since the functional_real class and its derivatives should embody the notion of a function, we implement them as *functors*. A C++ function object, or functor, allows an object to be invoked as though it were an ordinary function by overloading operator(). We begin by defining an abstract base class function[3]:

```
template <class Arg, class Result>
class function : public std::unary_function<Arg, Result>
{
public:
   virtual Result operator()(const Arg& x) const = 0;
};
```

If f is an object of a concrete derivative of function, then f(n) invokes the operator() method and returns an object of type Result given the argument n of type Arg. In our case, Arg is the type used to specify the precision, i.e. int, and the result type Result is an arbitrary-precision integer type, e.g. simple_integer. Subclasses then override operator() to implement their class-specific functionality.

[3] Note std::unary_function is a base class from the STL that simply provides two public user-friendly typedefs argument_type and result_type for Arg and Result.

On top of function, we provide a subclass functional_real, also abstract and generic. It adds approximation function-specific features, such as the caching of approximations which we discuss later, and forms the practical root of our hierarchy. A final abstract layer is defined and comprises two 'helper' classes that slightly simplify creation of the specific concrete unary and binary approximation function classes, e.g. square root and addition. The binary helper has the following form:

```
template <class Arg, class Result>
class binary_functional : public functional_real<Arg, Result>
{
protected:
   functor_ptr<Arg, Result> f1;
   functor_ptr<Arg, Result> f2;

public:
   binary_functional(const functional_real<Arg, Result>* p1,
                     const functional_real<Arg, Result>* p2)
      : f1(p1), f2(p2) {}
};
```

and the unary class is similar. The functor_ptr type is our reference-counted pointer type. Internally, it contains a pointer to a functional_real and manages a static table of these pointers to heap memory, with a count for the number of objects that link to each tree node in memory. The functor_ptr class also provides some syntactic sugar: it overloads operator() so that we do not have to get the internal pointer and dereference it before calling operator() on the functional_real object it points to—that is, we can write f1(n) rather than (*(f1.get()))(n).

The virtual method operator()(int n) declared for the functional_real hierarchy should be the main focus for subclasses. It is to be overriden by concrete subclasses to compute an approximation to precision n for the real number represented by the subclasses' operation. This method defines the implementation-specific semantics of an operation. In Chapters 3 and 4 we proved the correctness of a number of basic operations and elementary functions on real numbers in terms of integer arithmetic. Implementation is then trivial using our class structure and the integer arithmetic type Result, which should be a concrete subclass of big_integer, e.g. gmp_integer. For example, given two functional representations f_x and f_y of the real numbers x and y, we defined a functional representation f_{x+y} of the addition $x + y$ as $f_{x+y}(n) = \left\lfloor \frac{f_x(n+2)+f_y(n+2)}{4} \right\rfloor$. It is very easy to implement this definition as a class functional_add that inherits functional_real via binary_functional and overrides the operator() method to provide the specific addition semantics:

```
template <class Arg, class Result>
class functional_add : binary_functional<Arg, Result>
{
public:
   functional_add(const functional_real<Arg, Result>* x,
```

```
                const functional_real<Arg, Result>* y)
    : binary_functional<Arg, Result>(x, y) {}

  Result operator()(const Arg& n) const
  {
    return Result::scale(f1(n+2) + f2(n+2), -2);
  }
};
```

Here we assume big_integer provides a scale(x, k) method that shifts[4] an integer by k (i.e. multiplies x by 2^k) and rounds the result. f1 and f2 are functor_ptrs set to x and y respectively, which are each an object of a concrete subclass of functional_real that in turn implements its functional semantics through operator().

New operations on real numbers can be defined by first reasoning about their correctness as we did in Chapters 3 and 4. ExactLib then makes it very easy to implement this operation by deriving a class from a helper unary_functional or binary_functional that overrides the operator() method in order to realise the operation's specific semantics. In this way, we have a mechanism that simplifies and standardises the inclusion of new real operators/functions for us as the developers of the library or indeed for users.

5.4.3 Hierarchy of functional_real subclasses

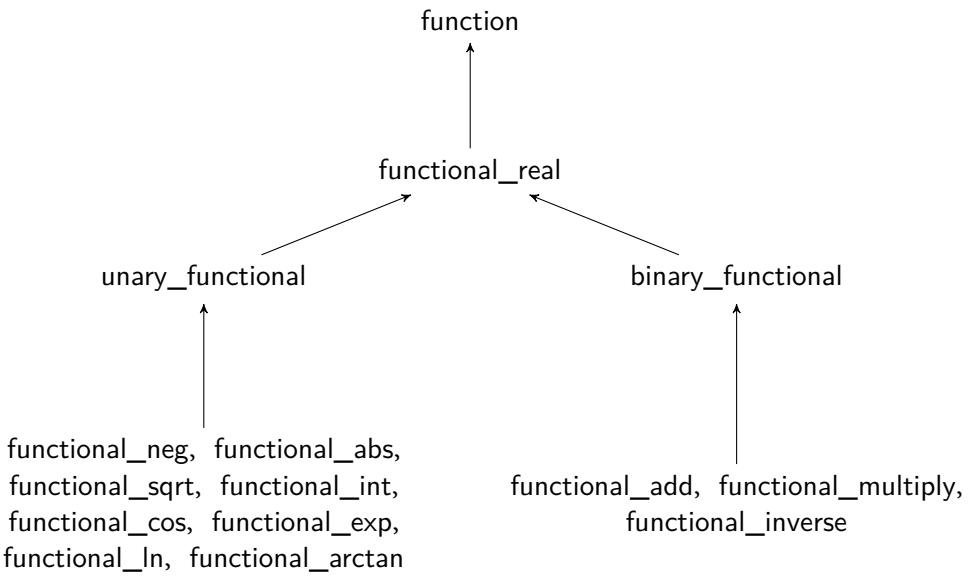


Figure 5.1: Hierarchy of core operation subclasses

[4]Recall the shift function is defined in Algorithm 3.4.

5.5 Expression trees

In this section, we address the problem of unnecessary growth due to common subexpressions during the formation of expression trees. We of course aim to minimise the size of trees—as we noted in Chapter 2, trees can grow to be very large and this can become quite memory intensive since we must store each node. Consider the following example:

```
real x = real::sqrt(2);
for (int i = 0; i < 2; ++i)
   x = x + x;
```

As addition operations are performed, a naive implementation might build a directed acyclic graph for computation similar to that of Figure 5.2, where two edges emanating from a node connect to the addition operation's operands. We have 7 nodes, but really only 3 are unique and therefore necessary. Noting that nodes are duplicated, a better implementation might form the expression tree in Figure 5.3, whereby multiple references (pointers) may point to the same computations.

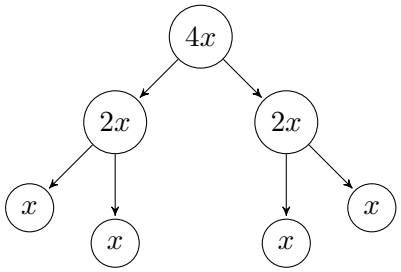


Figure 5.2: DAG with single references.

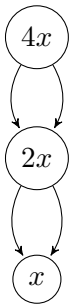


Figure 5.3: DAG with multiple references.

5.5.1 Reference-counted pointer type

We have already seen that each time an operation is encountered, we return a real whose internal pointer references a newly created object of a functional_real subclass type. To achieve the above optimisation, we wish to allow many objects of type real to point to the same functional_real-derived object. These objects are created on the heap and pointers are used to link them to real objects. However, C++ does not support garbage collection, and so we must keep track of our functional_real subclass objects. If such an object has no pointer left referencing it, it should be freed from the heap. Otherwise, if at least one real object points to it, the object should of course stay on the heap. The functor_ptr type is our reference-counted pointer type. Internally, it contains a pointer to a functional_real such that it can reference objects of any type that inherits functional_real. The class manages a static table of these pointers to heap

memory, with a count for the number of objects that link to each tree node in memory. When the count falls to zero, we free the memory from the heap.

5.5.2 Caching

Finally, to improve performance, each functional_real stores a big_integer representing its best approximation known to date and the precision this was made at. If a call to operator()(int n) requests a precision less than or equal to a computation we have already performed, we simply return our cached big_integer numerator, shifted as necessary; otherwise the call proceeds and the cache is updated.

This is of particular benefit to performance when we have a binary operation whose operands are the same object (recall Figure 5.3). For most binary operations, when an approximation is requested, it first computes an approximation to some precision of the first argument, and then an approximation to the second argument to the same precision. Without caching, we would needlessly perform the same computation twice. With it, we

5.6 Higher-order functions

ExactLib explores a novel alternative to representing arithmetic computations over real numbers in an imperative language. This is inspired by functional programming languages, where functions are considered first-class objects. Specifically, this means they can be passed as arguments to other functions, returned as values from other functions, and assigned to variables or stored in data structures.

Let us consider the example of function composition. Mathematically, given two functions $f : Y \to Z$ and $g : X \to Y$, their composition is a function $f \circ g : X \to Z$ such that $(f \circ g)(x) = f(g(x))$. In a functional language such as Haskell, the composition operator $(.)$ allows us to define a composite function:

```
descending :: Ord a => [a] -> [a]
descending = reverse . sort
```

We can then apply descending to list arguments, which will first sort and then reverse the given list. ExactLib provides similar higher-order function capabilities for functions over reals. The classes that support this functionality are found in the lambda namespace of exact. Assuming we have issued namespace using directives for exact::lambda::compose, exact::lambda::sqrt, etc., an example of usage might be:

```
function<real, real> safe_sqrt = compose(sqrt, abs);
real root = safe_sqrt(-2);
```

Clearly C++ cannot support quite the same level of syntactic conciseness afforded by Haskell, but our higher-order offerings are still very useable and represent an interesting alternative to the more traditional imperative style of arithmetic we have implemented.

Although ExactLib aims to integrate as well as possible with imperative C++ code, we believe a functional-style approach to computation is a worthwhile addition. It is reasonable to assume anyone using an exact real arithmetic library is doing so to perform mathematical calculations, and it is well known that functional languages and function-style approaches are highly suited to mathematical endeavours. Calculations written in are likely easier translated into a functional-style calculus. We bring the performance benefits of C++ together with—at least some—of the elegance of functional programming.

5.6.1 Implementing higher-order functions

Returning to the composition example, we can define the higher-order function as

```
template <class Func1, class Func2>
class composition : public function<typename Func2::argument_type,
                                    typename Func1::result_type>
{
private:
   Func1 f;
   Func2 g;

public:
   composition(const Func1& x, const Func2& y) : f(x), g(y) {}

   typename Func1::result_type
   operator()(const typename Func2::argument_type& x) const
   {
      return f(g(x));
   }
};
```

and we provide a user-friendly helper function for constructing composition function objects:

```
template <class Func1, class Func2>
inline composition<Func1, Func2> compose(const Func1& f1, const Func2
    & f2)
{
   return composition<Func1, Func2>(f1, f2);
}
```

The composition class supports the general composition of functions with differing domains and ranges, but for our composition we shall only instantiate the template with functions of type real_functor<real, real>. The real_functor wraps a functional_real subclass as a first-class object. That is, a functor whose operator() method applies the given real argument to the operation represented by the functional_real subclass, thus returning a real number that is the result of a function application:

```
template <class UnaryFunctional>
class real_functor : public function<real, real>
{
public:
   real operator()(const real& x) const
   {
      return real(new UnaryFunctional(x.internal));
   }
};
```

It serves the same purpose as the static functions real::sin(const real&), etc.: it exposes operations to the user as first-class objects so that they may be used in a higher-order application such as the composition compose(sqrt, abs). For example, we expose the square root and absolute value functions as real_functors like so:

```
namespace lambda
{
   static real_functor<functional_sqrt<Arg, Result> > sqrt;
   static real_functor<functional_cos<Arg, Result> > cos;
   ...
}
```

Returning to the composition class, it should now be clear how it works. Initialisation of a composition object with two real_functor real functions results in a functor representing the their composition. When we apply the composition to a real argument x, its operator() method is invoked with the argument x. In turn, the operator() method is called on the real_functor g. As we have seen, this has the effect of applying the operation defined by g to x, thus creating a new real object g(x). This real is then similarly applied to the real function f and we return the real whose expression tree represents the computation f(g(x)).

5.6.2 Other structural operators

In addition to composition, we have the following two structural operators:

- Product: given two functions $f_1 : X_1 \to Y_1$ and $f_2 : X_2 \to Y_2$, the *product* $f_1 \times f_2$ is the function $f_1 \times f_2 : X_1 \times X_2 \to Y_1 \times Y_2$ such that $(f_1 \times f_2)(x_1, x_2) = (f_1(x_1), f_2(x_2))$.
- Juxtaposition: given two functions $f_1 : X \to Y_1$ and $f_2 : X \to Y_2$, the *juxtaposition* $\langle f_1, f_2 \rangle$ is the function $\langle f_1, f_2 \rangle : X \to Y_1 \times Y_2$ such that $\langle f_1, f_2 \rangle(x) = (f_1(x), f_2(x))$.

We implement these higher-order functions in a similar way to the composition functor. It would take too long to present their classes so we instead list their helper function signatures to give the reader an idea of their usage:

```
function<std::pair<Arg1, Arg2>, std::pair<Result1, Result2> >
```

```
product(const function<Arg1, Result1>& f1,
        const function<Arg2, Result2>& f2);

function<Arg, std::pair<Result1, Result2> >
juxtaposition(const function<Arg, Result1>&,
              const function<Arg, Result2>&);
```

5.6.3 Projections

Although we have only provided a unary functor, real_functor, functions of several variables can be easily implemented using std::pairs or std::vectors to form tuples. Typically, users only require binary operations, so we provide two specialised projections on std::pairs and a general one on the ith component of a std::vector of values:

```
function<Arg, Result1>
first(const function<Arg, std::pair<Result1, Result2> >& f);

function<Arg, Result2>
second(const function<Arg, std::pair<Result1, Result2> >& f);

function<Arg, Result>
projection(const function<Arg, std::vector<Result> >& f, int i);
```

5.6.4 Partial function application

The following helper functions return functors with one of the parameters bound to a value:

```
function<Arg2, Result>
bind_first(const function<std::pair<Arg1, Arg2>, Result> >& f,
           const Arg1& x);

function<Arg1, Result>
bind_second(const function<std::pair<Arg1, Arg2>, Result> >& f,
            const Arg2& x);
```

For example, assuming we have a real_functor wrapper add for the functional_add addition operator, we could define an increment function on real numbers:

```
function<real, real> increment = bind_first(add, 1)
std::cout << increment(real::pi); // 4.14159265...
```

5.6.5 Functional arithmetic operators

If arithmetic operations can be applied to the return type of a function (as we have for any function that returns a real), then we can define a higher-order addition function.

For example, if we have two functions $f_1 : X \to Y$ and $f_2 : X \to Y$ with + defined over Y, then $f_1 + f_2 : X \to Y$ such that $(f_1 + f_2)(x) = f_1(x) + f_2(x)$. In the case of function addition, we provide the following helper function:

```
function<Arg, Result> operator+(const function<Arg, Result>& f1,
                                const function<Arg, Result>& f2);
```

Then we can write

```
function<real, real> f = lambda::sin + lambda::cos;
real x = f(real::pi);
```

so that x represents exactly the real number $\sin \pi + \cos \pi = -1$.

Similarly, it is useful to overload arithmetic operators that mix functors and constant values so that we can easily form functions like $x \mapsto f(x) + 5$ and $x \mapsto 3 \cdot g(x)$. For example, we have the following helper function overloads for addition:

```
function<Arg, real> operator+(const function<Arg, real>& f,
                              const real& k);

function<Arg, real> operator+(const real& k,
                              const function<Arg, real>& f);
```

PART III

EVALUATION

CHAPTER 6

EVALUATION

In this chapter, we evaluate our project both quantitatively and qualitatively. Our quantitative analysis consists of comparing the runtime performance of our work to various existing libraries. Qualitatively, we consider the design of our library and its suitability to exact real number computation.

6.1 Performance

To quantitatively evaluate our project, we compare ExactReal's runtime performance against existing exact real arithmetic systems, split into functional and imperative. We also consider the performance difference between using our the libraries simple arbitrary-precision integer type and the GNU GMP. All tests were performed on a system with a 2.13 GHz Intel Core 2 Duo processor and 4 GB of RAM running Mac OS X Snow Leopard.

6.1.1 Comparison to functional language implementations

We present our results in table 6.1. Clearly ExactLib performs comparatively very well, with the GMP version running significantly faster than our simple provision, as expected. David Lester's *ERA* [21] performs the best amongst the functional languages. This is probably because, like ExactLib, it is based on a constructive representation of real numbers, whereas the others use lazy representations. Abbas Edalat *et al.*'s *IC Reals* [13] and Thurston and Thielemann's *NumericPrelude* [29] perform the worst. Both of these systems are based on lazy representations—*IC Reals* using linear fractional transformations and *NumericPrelude* using a redundant signed digit stream—which suggests that, despite their elegance, they are not particularly practical in terms of performance.

6.1.2 Comparison to imperative language implementations

We present our results in table 6.2. Norbert Müller's *iRRAM* [23] library is the clear winner in terms of performance amongst the exact real libraries, and is the closest match to C++'s floating-point double type. However, *iRRAM* uses a bottom-up propagation constructive approach, for which we have no proof of correctness for operations.

Expression	ExactLib (simple)	ExactLib (GMP)	Few Digits	ERA	IC Reals
π	1.241 s	0.444 s	9.129 s	4.341 s	7.366 s
$\log \pi$	1.984 s	0.692 s	14.543 s	7.654 s	9.918 s
e	3.301 s	2.312 s	19.445 s	14.143 s	17.988 s
$\sin(\tan(\cos 1))$	6.341 s	3.003 s	> 1 m	> 1 m	> 1 m
$\exp(\exp(\exp(1/2)))$	5.778 s	2.220 s	·	47.004 s	27.783 s
π^{1000}	3.135 s	0.789 s	> 1 m	44.413 s	40.900 s
$\sin((3e)^3)$	5.931 s	2.232 s	·	> 1 m	> 1 m

Table 6.1: Comparison to functional implementations

Expression	ExactLib (GMP)	iRRAM	XR	CR	double
π	0.444 s	0.0985 s	3.304 s	0.831 s	0.00098 s
$\log \pi$	0.692 s	0.112 s	5.872 s	0.583 s	0.0010 s
e	2.312 s	0.491 s	11.346 s	3.309 s	0.0028 s
$\sin(\tan(\cos 1))$	3.003 s	0.667 s	> 1 m	4.131 s	0.0031 s
$\exp(\exp(\exp(1/2)))$	2.220 s	1.283 s	·	2.990 s	0.0022 s
π^{1000}	0.789 s	0.164 s	18.761 s	0.459	0.0024 s
$\sin((3e)^3)$	2.232 s	0.646 s	47.788 s	11.431 s	0.0024 s

Table 6.2: Performance comparison

Hans Boehm's *Constructive Reals (CR)* [7] is based on solid theory and proofs of correctness, and performs well. Whilst their constructive approach is similar, ExactLib (GMP) is probably only slightly faster because it uses C++ and the GMP rather than the slower Java and inferior *java.math.BigInteger* type *CR* uses—switching to the GMP would likely see *CR* outperform ExactLib since it has been worked on over a number of years. Keith Brigg's *XRC* [8] performs by far the worst, and this is most likely because it uses a lazy representation. This confirms our concerns about the lazy approach: even when using the extremely efficient C language,[1] its runtime performance is still poor.

It is clear that floating-point arithmetic performs much better. As we have discussed, this is to be expected due to hardware optimisations and its finite nature. Norbert Müller's *iRRAM* is the only library that can really compete, but is not without its shortcomings. However, ExactLib performs well and seems to offer a good balance between reasonably-elegant proof and reasonable performance, all in a mainstream language.

[1] This is perhaps not fair: one could rightly argue that C is not at all suited to the implementation of a lazy system; but the poor results of the functional implementations still stand.

6.1.3 Empirical correctness

We return to the iteration presented at our introduction in Chapter 1. If we recall, the iteration should converge to the golden ratio φ straight away but when use floating-point it strangely converges to $1 - \varphi$, a completely incorrect and indeed negative number. Appendix A.2 lists the code that implements this iteration using ExactLib to perform exact real number computation. Setting the precision to 30, we obtain the following output:

```
0: 1.61803398874989484820458683436 6
1: 1.61803398874989484820458683436 6
2: 1.61803398874989484820458683436 6
3: 1.61803398874989484820458683436 6
4: 1.61803398874989484820458683436 6
...
```

Comparing these results to the list of digits known to be correct[2] suggests that our library correctly performs exact real number computation. This of course was one of the major aims and we have provided a solution to the problem presented at the outset of our project.

6.2 Library design

In this section, we evaluate our project qualitatively. Recurring metrics are the expressiveness, ease-of-use and extensibility of our design.

6.2.1 Translation between theory and implementation

The design of our library means it is trivial to implement exact real operations. That is, it is easy to translate our proven mathematical algorithms of Chapters 3 and 4, and any new definitions, into code.

In section 5.4.2 we illustrated how we implement the addition operation. As another example, consider the square root function defined in Algorithm 3.5. Recall if a real number $x \geq 0$ is represented by the approximation function f_x, then the approximation function $f_{\sqrt{x}}$ represents $\sqrt{x}$ if it satisfies

$$\forall n \in \mathbb{Z} \,.\, f_{\sqrt{x}} = \left\lfloor \sqrt{f_x(2n)} \right\rfloor .$$

To implement this, we simply define an operation class that inherits functional_real via unary_functional and then overload its operator() method:

```
template <typename Arg, typename Result>
class functional_sqrt : unary_functional<Arg, Result>
{
```

[2] http://jaohxv.calico.jp/pai/ephivalue.html

```
public:
   functional_sqrt(const functional_real<Arg, Result>* p)
      : unary_functional<Arg, Result>(p) {}

   Result operator()(const Arg& n) const
   {
      return Result::floorsqrt( f(2*n) );
   }
};
```

The overridden operator() method is the important thing to observe: it implements the specific approximation function definition for $f_{\sqrt{x}}$. Importantly, we note the semantic and syntactic closeness to our mathematical definition that providing f as a functor affords. This suggests that we do not need to prove the correctness of our code since it is so close to the proven mathematical definitions. Our use of functors allows us to mimic the syntax of mathematical functions so that translating operation definitions on reals to and from an implementation is easy and their use feels natural.

By defining an abstract base operation class and using polymorphism to work with its hierarchy, any class that inherits functional_real can be used in an expression tree. This means users can extend the library with new operations in a uniform way, and they will work seamlessly in computations with the other operations that have been defined.

6.2.2 Ease-of-use and expressivity

From a user's point of view, our library allows mathematical calculations to be expressed in a number of different ways. We have seen two ways provided by ExactLib: through traditional imperative functions, e.g. real::sqrt, and through functors suitable for use with higher-order functions, e.g. lambda::sqrt.

The latter is a particular expressive way for users to translate mathematical expressions into exact real computations. For example, returning to the golden ratio iteration, we have:

$$\gamma_0 = \frac{1+\sqrt{5}}{2} \quad \text{and} \quad \gamma_{n+1} = \frac{1}{\gamma_n - 1} \quad \text{for } n \geq 0. \tag{6.1}$$

Using ExactLib, we can easily translate this into an exact real computation:

```
real x = (1 + real::sqrt(5)) / 2;
function<real, real> iteration = 1 / (lambda::id - 1);

std::vector<real> gamma;
for (int i = 0; i < 100; ++i) {
   gamma.push_back(x);
   x = iteration(x);
}
```

Note the similarity of the first two lines to the recurrence relation (6.1). This expressive power is made possible through our functional arithmetic operators (section 5.6.5) and the identity functor lambda::id. Unfortunately, we cannot be quite as expressive and elegant as a library written in Haskell, but our work is an improvement over existing imperative libraries.

The core hierarchy of subclasses extending from functional_real form the internal representation of computations as trees of approximation functions. As we have seen, we provide a user-facing class, real, that allows users to build and execute this internal tree. But this decoupled design means there is nothing to stop developers from extending the library with an alternative user-facing class to express the construction and execution of exact real calculations in a different style.

6.2.3 Functionality and flexibility

As we have discussed, the real type provided by our library is actually a specialisation of the generic base_real type. This allows the user to specify their own arbitrary-precision integer type, so long as it implements the big_integer interface. Such a generic approach is unique amongst exact real arithmetic systems and means our library can easily be made compatible with any new integer library that becomes available. This is important since the integer arithmetic used is a fundamental factor affecting performance and correctness.

In terms of functionality, we have provided the major operations and functions one would expect from a real arithmetic library. We have also made it easy for users to extend the library with their own functionality.

6.3 Representation and paradigm

Whilst lazy representations such as the redundant signed digit stream are arguably more elegant, we have seen that the time performance of their implementations leave much to be desired. On the other hand, the performance of Norbert Müller's iRRAM is very good, but it is complicated to implement and more importantly there is no proof of its correctness. We feel our approach of a solid representation theory, tractable proofs of correctness, and reasonable implementation performance strikes a good balance.

The power of C++'s templates and functors have fully validated our choice of language over more modern alternatives such as C# and Java. Their role in our development of higher-order functions have resulted in a library that benefits from the runtime efficiency of C++ and approaches a similar level of elegance as a functional language implementation of a lazy list representation.

CHAPTER 7

CONCLUSIONS & FUTURE WORK

7.1 Conclusions

We have presented a representation of computable real numbers that admits tractable proofs of correctness and is well-suited to efficient imperative implementation. The correctness of a number of basic operations and elementary functions have been proved for this representation. A library, ExactLib, has been developed that implements our exact real arithmetic in C++. We have designed it to integrate seamlessly with C++, defining a data type called real, and associated operations and functions, that aims to behave in the same way as C++'s built-in numeric types. We have also introduced a highly-expressive functional style of performing exact computations as a more elegant alternative to the traditional imperative style usually offered. This gives the user the benefit of being able to write code in a functional style to perform exact real arithmetic, whilst in the background implementing the actual calculations in an efficient imperative manner. The design of our library's core is flexible and makes it easy to extend the library in a uniform way. In particular, proven mathematical definitions of functions and operations can be easily translated into code and their closeness to the mathematics relaxes the need to prove the correctness of our code. Any arbitrary-precision integer library can be used, provided we can supply an appropriate wrapper—a generality that is important since this choice greatly affects performance. A simple arbitrary-integer type is packaged with ExactLib so that it may be imported as a standalone library with no dependencies other than the standard C++ library. We hope that by providing a standalone library in a high-performance mainstream language like C++ will make performing exact real number computation easier.

During our research we investigated a number of lazy representations of the real numbers. In retrospect, this was perhaps time not well spent, since we were aiming to provide an implementation in an imperative language, C++. After experimentation at the early stage, it quickly became clear that trying to implement a lazy approach in an imperative language is simply using the wrong tool for the job.

On the other hand, we have been very pleased with our choice of using C++. The power afforded by its template system, functors and operator overloading, among others, has made it possible to produce a well-designed library that fits the language. Whilst it was unclear during the preliminary stages, we have realised C++ currently best fits the needs of an exact arithmetic library, with C slightly too low-level and not

expressive or elegant enough, and Java not quite being efficient enough and with poor support for generic types.

Whilst providing an arbitrary-precision integer type for standalone inclusion of the library seemed a nice feature, we have seen that it is simply unwise not to use the GNU GMP. Looking back, we should not have wasted time rewriting `xintlib` into a suitable integer type.

Finally, we have seen that, although iRRAM's bottom-up interval approach offers the best runtime performance of exact real arithmetic libraries, the top-down Cauchy approximation function representation can be implemented to provide very reasonable performance, whilst still exhibiting tractable proofs of correctness. In terms of elegance and ease-of-use, our experience has shown that it is possible to encourage concise functionality in an imperative language.

7.2 Future work

Whilst C++ offers some powerful features—notably its template system, operator overloading and functors—that help us provide a more elegant library to users, there is a definite limit to its expressiveness. We believe work using languages like C# and Ruby or Python could be worthwhile in the future. C# is a mainstream language that is compiled[1] but offers generics—similar to C++ templates—and some native functional features that could prove to be useful to a real arithmetic library. Ruby and Python support more powerful metaprogramming features that make it easy to write lightweight Domain Specific Languages (DSLs), which could be particularly useful to provide users with a syntax close to mathematical notation for describing computations using exact reals. However, they are both interpreted[2] rather than compiled, which severely affects runtime performance.

The most important factor affecting performance in all representations is hardware optimisation. Floating-point arithmetic is particularly efficient because it is implemented in hardware on most computer systems. Future work that could propose a representation similar in form to the approximation function representation but that has elements that are candidates for hardware optimisation would represent an important step in imperative exact real number computation. Until an exact representation is found that is suitable for hardware optimisation we cannot match the performance seen with floating-point arithmetic.

[1] Actually, it is just-in-time compiled, but in practice this is just as efficient.

[2] Implementations of Ruby and Python targeting the Java Virtual Machine are under development but are relatively immature as of writing.

BIBLIOGRAPHY

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[2] D. Bailey, P. Borwein, S. Plouffe. *On the Rapid Computation of Various Polylogarithmic Constants*. Mathematics of Computation, vol. 66, pp. 903–913, 1997.

[3] J. Blanck. *Exact Real Arithmetic Systems: Results of Competition*. Computability and Complexity in Analysis, 4th International Workshop CCA, 2000.

[4] H. Boehm, R. Cartwright, M. Riggle, M. O'Donnell. *Exact real arithmetic: A case study in higher order programming*. Proceedings of the 1986 Lisp and Functional Programming Conference, pp. 162–173, 1986.

[5] H. Boehm. *Constructive real interpretation of numerical programs*. SIGPLAN Notices, vol. 22, p. 214–221, 1987.

[6] H. Boehm. *Exact Real Arithmetic: Formulation Real Numbers as Functions*. Department of Computer Science, Rice University, March 1993.

[7] H. Boehm. *The constructive reals as a Java library*. HP Laboratories, Palo Alto, 2004. http://www.hpl.hp.com/techreports/2004/HPL-2004-70.pdf.

[8] K. Briggs. *XRC: Exact reals in C*. http://keithbriggs.info/xrc.html.

[9] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[10] D. Collins. *Continued Fractions*. The Massachusetts Institute of Technology Undergraduate Journal of Mathematics, volume 1, 1999.

[11] B. Dawes, D. Abrahams. *Boost C++ Libraries*. http://www.boost.org/.

[12] A. Edalat, R. Heckmann. *Computing with real numbers*. Lecture Notes in Computer Science 2395, 39–51, 2002.

[13] L. Errington, R. Heckmann. *Using the IC Reals Library*. Imperial College London, January 2002.

[14] H. Geuvers, M. Niqui, B. Spitters, F. Wiedijk. *Constructive analysis, types and exact real numbers*. Technical paper, Radboud University Nijmegen, December 2006.

[15] D. Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computing Surveys, vol. 23, pp. 5–48, 1991.

[16] P. Gowland, D. Lester. *The correctness of an implementation of exact arithmetic*. Manchester Arithmetic Project, Manchester University.

[17] S. Graillat, F. Jézéquel, Y. Zhu. *Stochastic Arithmetic in Multiprecision*. University Pierre et Marie Curie.

[18] T. Granlund, *et al. The GNU Multiple Precision Arithmetic Library*. http://gmplib.org/.

[19] M. Herrmann. *Coinductive Definitions and Real Numbers*. Final-year undergraduate project, Imperial College London, 2009.

[20] V. Lee, Jr. *Optimizing Programs over the Constructive Reals*. PhD thesis, Rice University, 1991.

[21] D. Lester. *ERA: Exact Real Arithmetic*. School of Computer Science, University of Manchester, 2001. http://www.cs.manchester.ac.uk/arch/dlester/exact.html.

[22] V. Ménissier-Morain. *Arbitrary precision real arithmetic: design and algorithms*. Journal of Symbolic Computation, 1996.

[23] N. Müller. *iRRAM — Exact Arithmetic in C++*, 2008. http://www.informatik.uni-trier.de/iRRAM/.

[24] J. Müller, *et al. Floating-Point Arithmetic*. École Normale Supérieure de Lyon, 2009.

[25] D. Phatak, I. Koren. *Hybrid Signed-Digit Number Systems: A Unified Framework for Redundant Number Representations with Bounded Carry Propagation Chains*. IEEE Transactions on Computers 8, vol. 43, pp. 880–891, August 1994.

[26] D. Plume. *A Calculator for Exact Real Number Computation*. Departments of Computer Science and Artificial Intelligence, University of Edinburgh, 1998.

[27] P. Potts. *Computable Real Number Arithmetic Using Linear Fractional Transformations*. Technical report, Department of Computing, Imperial College London, June 1996.

[28] A. Shvetsov. *xintlib: C++ template library for big-number integer calculations*. http://code.google.com/p/xintlib.

[29] D. Thurston, H. Thielemann. *NumericPrelude*. http://www.haskell.org/haskellwiki/Numeric_Prelude.

[30] A. M. Turing. *On computable numbers with an application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, 2 42 (1), pp. 230–265, 1936.

[31] J. Vuillemin. *Exact real computer arithmetic with continued fractions*. IEEE Transactions on Computers 39, 1087–1105, August 1990.

[32] Institute of Electrical and Electronics Engineers (IEEE). *The IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008)*. http://ieeexplore.ieee.org/servlet/opac?punumber=4610933, August 2008.

[33] Wikimedia Foundation, Inc. *Generalized Continued Fraction*. Wikipedia, accessed May 2011. http://en.wikipedia.org/wiki/Generalized_continued_fraction.

CHAPTER A

CODE LISTINGS

A.1 Golden ratio iteration (floating-point)

```
#include <iostream>
#include <iomanip>
#include <cmath>

int main()
{
   const int max = 100;
   double u = (1 + std::sqrt(5))/2;

   std::cout << std::setprecision(13) << "u0\t" << u << std::endl;

   for (int i = 1; i <= max; ++i)
   {
     u = 1/(u - 1);
     std::cout << "u" << i << "\t" << u << std::endl;
   }

   return 0;
}
```

A.2 Golden ratio iteration (ExactLib)

```
#include <iostream>
#include <iomanip>
#include "exact.hpp"
using exact::real; using exact::function;

int main()
{
   const int max = 100;

   real x = (1 + real::sqrt(5)) / 2;
   function<real, real> iteration = 1 / (lambda::id - 1);
```

```
12
13      std::cout << std::setprecision(30) << "x0\t" << x << std::endl;
14      for (int i = 0; i < max; ++i)
15      {
16         std::cout << "x:" << i << "\t" << x << std::endl;
17         x = iteration(x);
18      }
19
20     return 0;
21  }
```