University of Edinburgh
Departments of Computer Science and Artificial Intelligence

A Calculator for Exact Real Number Computation

4th Year Project Report

Dave Plume

Supervisors: Martín Escardó and Alex Simpson

1998

Abstract: The most usual approach to real arithmetic on computers consists of using floating point approximations. Unfortunately, floating point arithmetic can sometimes produce wildly erroneous results. One alternative approach is to use *exact real arithmetic*. Exact real arithmetic allows exact real number computation to be performed without the roundoff errors characteristic of other methods. We observe that conventional representations such as binary are inadequate for this purpose, and consider two alternative representations of reals. Algorithms for the basic arithmetic operations, transcendental functions, integration, and function minimum and maximum are implemented. These include new algorithms for direct multiplication of signed binary streams, division, and the evaluation of limits of Cauchy sequences. A theoretical and experimental analysis of some of these algorithms is performed which shows that algorithms for the same operation using different representations behave differently. It also shows that even relatively simple expressions can require many digits of input to compute even a single digit of output and that one of the representations can exhibit a form of expression swell which destroys performance. The experimental work shows the performance of the implemented system to be poor on complex expressions, and we discuss why this might be so.

Acknowledgements

I am extremely grateful to my supervisors Martín Escardó and Alex Simpson for all their help and encouragement. I would particularly like to thank Alex for his proof reading, and Martín for his time, patience, and advice during the writing of this report.

I would also like to thank the departments of Computer Science and Artificial Intelligence at the University of Edinburgh for the use of their resources, and in particular the support staff for their assistance and for maintaining reliable computing services.

Finally, I would like to acknowledge the writers of HUGS, the Haskell tools "Alex" and "Happy", and the Glasgow Haskell Group, whose software made my implementation possible.

Contents

1. Introduction

In this work we have developed a calculator for exact real arithmetic. In addition we have experimented with and analysed the representations and algorithms required to do so. The calculator includes the basic arithmetic operations, a number of transcendental functions, integration and function minimum and maximum. These are accessed through a simple text prompt user interface.

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

There are a number of alternative approaches to real arithmetic on a computer, including floating point arithmetic with error analysis, interval arithmetic, a stochastic rounding technique, and the symbolic manipulation of expressions as performed by computer algebra systems. Each of these approaches has advantages and disadvantages.

Exact real arithmetic is a method of performing arithmetic operations to arbitrary precision and obtaining results which are guaranteed correct. It relies on the use of finite sized representations of infinite objects to represent numbers. A finite portion of this infinite object may be evaluated explicitly to return a result with the required precision to the user, but the object itself still expresses the number exactly.

There are a number of possible representations one might use for exact real arithmetic. The representations used in this project are based on familiar digit representations (eg. binary, decimal), and include the use of signed binary digits, and the use of dyadic rationals as 'pseudo-digits'.

The techniques used to develop algorithms for the arithmetic and other operations on these representations include range analysis, exploiting the relationship between list operations and mathematical ones that exist when we use these representations, and the use of identities to re-express numerals.

There are numerous potential applications of an exact real arithmetic package, although the performance and memory constraints are likely to make this approach impractical for 'everyday' real arithmetic. These potential applications might include:

- Validation of test data or outputs during the testing of hardware and software products.
- Scientific programming and simulation.
- Work with fractals and iterated function systems.
- Situations where the programming model is more important than high performance.
- Small computations or sub-computations which are highly sensitive to small variations. Gaussian elimination with iterative refinement is an example of such a computation.

Unfortunately at the present time the theory behind exact real arithmetic, especially regarding tractability, is not sufficiently advanced to seriously discuss the potential applications of a such a package in more than abstract terms. See Ko [18] for further discussion of these issues.

Developing a calculator for exact real arithmetic is interesting because it involves both finding representations and developing algorithms real number computation, and also implementing these representations and operations. The end product demonstrates that this approach can actually be used in a real application, albeit a simple one.

The definition of a "calculator for exact real arithmetic" is an extremely open-ended one. At the very least one would expect the basic arithmetic operations and a simple user interface. However such a calculator might also include an extensive library of transcendental functions or functional operations, or have an elaborate user interface and expression/programming language. We have taken a middle road, implementing a reasonable selection of transcendental functions and some functional operations, and including simple variables and variable precision output in the expression language of the user interface. In addition to this implementation, we have then spent time analysing the algorithms we have developed and experimenting with the implementation.

1.1 Work Performed

The work performed is composed of three distinct parts; finding suitable representations for reals and developing algorithms for the required operations on them, designing and implementing algorithms and a user interface, and analysing the performance of the algorithms and implementation.

We have used two basic representations of reals on the closed interval $[-1, 1]$; streams of signed binary digits, and streams dyadic rationals which are contained in the interval $[-1, 1]$ and treated as digits. Many algorithms are easier to define using the dyadic rational representation, but they often perform poorly if the size of the rationals involved grows. The signed binary digit representation does not suffer from this problem, and is easier to use to implement other algorithms because we can use pattern matching with the small, finite set of digits. In fact we use the signed binary representation as the main representation, but observe that for some algorithms, such as division and the functional operations, it is much easier if we take a signed binary input, generate a dyadic output, and then convert the result back into the signed binary representation.

The whole real line $(-\infty, \infty)$ is represented using a mantissa which is a stream of either dyadic or signed binary digits, and an exponent of arbitrary size. A representation of reals as a sequence of nested closed intervals whose end-points are expressed using the signed binary (mantissa, exponent) representation mentioned above is also used because it is useful when implementing algorithms for transcendental functions.

The operations fall into three broad categories; the basic arithmetic operations and ones which convert between representations, high level transcendental functions, and the functional operations. Many of the algorithms for the basic operations are already known, but in this work we develop algorithms computing the division of two numbers, for the multiplication of two signed binary streams, and for computing the limits of Cauchy sequences and hence many transcendental functions.

The algorithm for the multiplication of signed binary developed streams does so working solely with signed binary digits. The novel (but considerably simpler) approach originally proposed during personal communications with Alex Simpson and Martín Escardó was to take a signed binary digit input, generate output using the dyadic digits, and then convert the result back into a signed binary representation.

The division algorithm developed and proved is more complex than the algorithms for the other basic arithmetic operations. It is also considerably more efficient than an alternative approach tested which found the reciprocal of the denominator by computing the limit of an iterative approximation of reciprocal, and then multiplying this by the numerator.

The new algorithm designed for the computation of limits of Cauchy sequences is a development of an idea originally proposed by Martín Escardó (personal communication). It uses the representation of reals as a sequence of nested closed intervals whose end-points use the signed binary (mantissa, exponent) representation. Given a suitable Cauchy sequence for which we know the rate (or sometimes just the manner) of convergence to its limit, we can generate a sequence of nested intervals of this form. The algorithm allows us to then convert this sequence of nested intervals representation into a single numeral which representing the limit of the original Cauchy sequence. We give a description of this algorithm, and an proof of correctness.

The algorithm for the computation of the limits of Cauchy sequence forms the basis of the algorithms for transcendental functions. Many transcendental functions can be expressed very easily as the limits of Cauchy sequences, and once we have found such a sequence we can use the approach described above directly to implement an algorithm to compute the function. The set of transcendental functions which we implement in the calculator includes the basic trigonometric and logarithmic functions, but this set of available functions could be extended with very little additional effort to include more transcendental functions using this approach.

Also implemented here are Alex Simpson's algorithms for the functional operations integration and function minimum and maximum [26]. In particular, this includes the first implementation of these algorithms on the whole real line.

We implement the calculator using the functional programming language Haskell. The implementation can be loosely divided into the modules containing the core algorithms, and a simple text based user interface which is used to demonstrate them.

Finally we analyse the algorithms and implementation. This allows us to evaluate the approach and discuss the performance of the algorithms. Experimentation is used to test hypotheses made in the theoretical analysis, and allows us to assess the performance of the implementation as a whole.

We examine the basic algorithms to determine the lookahead (the number of digits of input are required to obtain n digits of output), whether the algorithm branches (spawns additional stream operations as it generates output digits), and whether the size of the representation of dyadic rational style digits swells in algorithms which output dyadic digit streams. These results can then be used to examine expressions involving sequences of operations. We make observations about the lookahead required for the iterated logistic map and the effect of changing the structure of expressions can have on the lookahead.

The experimentation confirms some of the theoretical predications and demonstrates the real behaviour of the algorithms for which we can determine a minimum and maximum lookahead, but for which these are different and the maximum lookahead is not always required. It also illustrates the problem of the dyadic digit representation size swelling and its effect on performance. Lastly we show that although the performance of this implementation is adequate for simple computations, this particular implementation is probably too slow to be applied to applications involving more complex computations. We discuss why this might be so.

1.2 Related Work

The inadequacy of standard digit representations for exact real number computation was first recognised by Brouwer [6] in 1920. Section 2.2.1 shows why this is the case. In 1937 Alan Turing introduced the notion of a computable function [28, 29], and since then, a number of different approaches for exact real arithmetic have been proposed. A brief introduction to some of this related work is presented here. Section 2.2.3 discusses some of the representations used by these authors in more detail.

Avizienis [1] considers a signed digit representation, as does Wiedmer [33] when he discusses computation with infinite objects represented as words which may be of infinite length. Wiedmer presents an algorithm for the addition of two signed decimal numbers represented using an infinite signed decimal representation.

Gosper [13] discusses the use of continued fractions for numerical calculations, and this work is continued by Vuillemin [31] who uses this representation for exact real arithmetic. Kornerup and Matula [19] also use continued fractions for real arithmetic, and consider the use of redundant binary strings to representation the coefficients these fractions.

Nielsen and Kornerup [21] discuss a representation of numbers as potentially infinite strings of signed digits, and define various classes of computable functions on them. They also introduce a floating point style representation not dissimilar to the one used in this project (see section 3.4).

Boehm *et al* [3] discuss a number of representations, including the signed digit and continued fraction approaches, and there and in [5] consider a representation of reals as functions mapping an integer specifying the precision required as n digits in base b to a rational correct to the specified precision. Boehm and Cartwright [4] implement algorithms using this and other representations, and discuss their relative performance.

Valérie Ménissier-Morain [20] gives some extremely striking examples of the problems with floating point arithmetic, and summarises approaches to solving these problems. She uses a representation of reals using a sequence of (dyadic) rational numbers which is different from the one used here, and gives algorithms for the usual arithmetic operations and transcendental functions.

Pietro di Gianantonio [8, 9] works extensively with digit representations, considering the signed digits used here in addition to a similar one which instead of using three digits in base two, uses two digits whose base is the golden ratio $\varphi = (\sqrt{5}+1)/2$. Algorithms for the usual arithmetic operations in the golden ration notation are given. He also considers an extension of the idealised functional programming language PCF representing reals using the compositions of the linear maps $\frac{x-1}{2}, \frac{x}{2}, \frac{x+1}{2}$.

Alex Simpson [26] uses the signed digits and digits formed from dyadic rationals (the two representations used here) to define algorithms for function integration and function minimum and maximum. These algorithms are implemented in this work.

Martín Escardó [11] uses a representation of reals as an infinite sequence of nested intervals with rational endpoints, and develops an extension to the language PCF which is related to the work of di Gianantonio. He also considers the signed digit representation in [12] and describes a surprising method by which a pair of numerals may be normalised such that their numerical and lexicographic orderings (which are normally unrelated) can be made to coincide.

Abbas Edalat, Peter Potts, Martín Escardó [23] consider another extension to the language PCF using Möbius transforms, a type of linear fractional transformation on extended real numbers. Such transformations have the property that their composition can be computed neatly using matrix multiplication. Edalat and Potts [10, 24] continue the work of Gosper, Vuillemin, and Nielsen and Kornerup, and develop algorithms for the usual arithmetic operations and transcendental functions

Reinhold Heckmann [14, 15] analyses the convergence of the algorithms of Edalat and Potts, and the problems of a swelling representation size.

Interestingly, an on-line calculator for exact real arithmetic demonstrating written by Peter John Potts demonstrating the Linear Fractional Transformation approach available. This calculator available through a Java interface on the World Wide Web at `http://theory.doc.ic.ac.uk/~pjp/RealClient`.

`html`, but appears to connect to a C++ server.

Philipp Sünderhauf [27] discusses the benefits of incremental and non-incremental representations, and the creation of hybrid representations to achieve the benefits of each.

Although several of these authors discuss the signed digit representation, none state or prove any operations other than addition using the signed digit stream representation adopted here. This, and the goal of implementing Alex Simpson's functional operations, motivates the work performed here.

1.3 Organisation

The report is structured as follows:

In chapter 2 we discuss real arithmetic in general terms, examine some approaches to performing it on a computer, and introduce exact real arithmetic.

Chapter 3 defines the the representations used, chapter 4 develops the algorithms for the basic operations, and chapter 5 those for higher level transcendental functions. Proofs of correctness are included for some algorithms.

In chapter 6, the implementation of the algorithms is discussed, and in chapter 7 some experimentation and analysis of the work is given.

In the conclusion, chapter 8, we draw together the work, make some concluding remarks, and discuss potential extensions and directions for future work.

2. Real Number Computation

This Chapter briefly introduces the field of exact real arithmetic and discusses in more detail some of the related work mentioned in the introduction. We also discuss different approaches to real arithmetic in more detail, and some of the problems associated with these approaches.

2.1 Approaches to Real Arithmetic

There are a number of approaches to performing computations involving real numbers using a computer. These include:

- Floating point arithmetic
- Floating point arithmetic with error analysis
- Interval arithmetic
- Stochastic Rounding
- Symbolic manipulation
- Exact Real Arithmetic

2.1.1 Floating Point Arithmetic

Conventionally, floating point arithmetic has been used to perform real arithmetic with computers, and floating point operations have been highly studied and optimised. The standard most commonly used is IEEE-754, which includes 32-bit 'single' precision and 64-bit 'double' precision representations.

Floating point represents a number as fixed length binary mantissa and an exponent of fixed size. This representation only allows numbers in a finite subinterval of the whole real line to be represented, and only a finite number of the elements within that interval to be represented exactly.

The major problem with floating point arithmetic is that it is inherently inaccurate. There are two reasons for this. Firstly because only a finite subset of real numbers may be represented exactly, it is necessary to approximate all other numbers by the nearest representable one. Secondly, every time a floating point operation is performed, rounding occurs, and the rounding error introduced can have very serious effect on the accuracy of the result.

If several floating point operations are performed, carrying the result from one to the next, the rounding error is propagated. In some cases, this can lead to the computed result being entirely 'noise' and bearing no apparent relation to the actual one.

Other problems with floating point arithmetic include the fact that the floating point arithmetic operations and transcendental functions do not correspond directly to the mathematical ones, in addition to which certain mathematical laws such as associativity do not hold.

One dramatic demonstration of the potential inadequacy of floating point arithmetic is to observe the effects of computing successive iterations of the so called logistic map using floating point arithmetic, and then observing how the computed result differs from the correct one.

The logistic map is defined as $f(x) = Ax(1 - x)$ where A is a real constant. The iterated logistic map, which is also known as the Verhulst Model after it was published in 1845 by the Belgian mathematician Pierre Verhulst as a model of population growth, exhibits chaotic behaviour when certain values for the constant A are used.

We let $A = 4$, and x to be the following simple (arbitrary but machine representable) number. The subscripts here denote the base of the representation.

$$x = (0.671875)_{10} \qquad = (0.101011)_2$$

The logistic map function is now iterated repeatedly using single and double precision arithmetic. The correct result (computed using the exact real arithmetic functions implemented as part of this work) is shown in the right hand column. All correct digits are underlined.

It.	Single Precision	Double Precision	Correct Result
1	$\underline{0.881836}$	$\underline{0.881836}$	$\underline{0.881836}$
5	$\underline{0.384327}$	$\underline{0.384327}$	$\underline{0.384327}$
10	$\underline{0.31303}4$	$\underline{0.313037}$	$\underline{0.313037}$
15	$\underline{0.0227}02$	$\underline{0.022736}$	$\underline{0.022736}$
20	$\underline{0.98}3813$	$\underline{0.982892}$	$\underline{0.982892}$
25	0.652837	$\underline{0.757549}$	$\underline{0.757549}$
30	0.934927	$\underline{0.481445}$	$\underline{0.481445}$
40	0.057696	$\underline{0.02400}8$	$\underline{0.024009}$
50	0.042174	$\underline{0.62}9402$	$\underline{0.625028}$
60	0.934518	0.757154	$\underline{0.315445}$

One can clearly see how very rapidly the results computed using floating point arithmetic become incorrect. Increasing the size of the mantissa clearly improves matters, but the same problems occur, and simply increasing the size of the mantissa does not necessarily guarantee that any given computation is correct. Valérie Ménissier-Morain [20] explores these problems further.

2.1.2 Floating Point arithmetic with error analysis

One approach to dealing with the problems of accuracy when using floating point arithmetic is to perform error analysis. This involves doing calculations to obtain a bound on the error of a particular expression.

One approach is to use the assumption that a real number x is approximated by the number $\hat{x}$, where $\hat{x} = x(1+\epsilon)$. In this equation, ϵ is the relative error in the representation. Hence:

$$\epsilon = \frac{\hat{x} - x}{x}$$

It is now possible to calculate the effect that certain operations will have on the relative error of a floating point computation. Operations such as floating point multiplication will affect the relative error, but not significantly. Let $\hat{x} = x(1+\epsilon) = \hat{x}_1 \cdot \hat{x}_2$ where the desired result is $x_1 \cdot x_2$:

$$\hat{x}_1 \cdot \hat{x}_2 \quad = \quad x_1(1+\epsilon_1) \times x_2(1+\epsilon_2)$$

$$\begin{aligned} &= x_1x_2 \cdot \Big((1+\epsilon_1)\cdot(1+\epsilon_2)\Big) \\ &= x_1x_2 \cdot \Big(1 + (\epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2)\Big) \\ &\Rightarrow \epsilon = (\epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2) \end{aligned}$$

Operations such as addition or subtraction, however, can have a much more significant effect on the relative error in certain cases. Consider the subtraction $\hat{x} = x(1+\epsilon) = \hat{x}_1 - \hat{x}_2$:

$$\begin{aligned} \hat{x}_1 - \hat{x}_2 &= x_1(1+\epsilon_1) - x_2(1+\epsilon_2) \\ &= (x_1 - x_2) + (x_1\epsilon_1 - x_2\epsilon_2) \\ &= (x_1 - x_2)\cdot\left(1 + \frac{x_1\epsilon_1 - x_2\epsilon_2}{x_1 - x_2}\right) \\ &\Rightarrow \epsilon = \frac{x_1\epsilon_1 - x_2\epsilon_2}{x_1 - x_2} \end{aligned}$$

It is now clear that if x_2 is nearly equal to x_1, the relative error will be greatly magnified.

The use of simple methods like this, or more sophisticated approaches, allow the accuracy of a given computation to be examined. This may allow the user to have faith in the results of a floating point computation. Knuth [17] describes the approach illustrated here and gives a more detailed discussion of the problem.

The major problems with such methods are that firstly the error analysis may simply tell the user that he or she should have no faith whatsoever in the correctness of the result produced, and secondly that the error analysis must be performed for every computation and is not general.

2.1.3 Interval Arithmetic

Interval arithmetic involves expressing a real as a pair of numbers which represent an interval containing the number. A representation such as floating point arithmetic may be used to express the lower and upper bounds.

Interval arithmetic operations compute new upper and lower bounds on the result after the operation has been performed. This may be performed using floating point arithmetic, but rounding strictly upwards for the upper bounds and strictly downwards for the lower bounds.

Interval arithmetic is extremely useful. Once a suitable interval arithmetic package is available, no further analysis need be performed on specific computations themselves. Combining the results of the upper and lower bounds allows the result to be expressed to an appropriate number of correct significant digits. In addition, when an input is not known exactly but only to some small number of digits (eg. a physical measurement of some kind), this fact can be represented and is reflected in the tightness of the bound on the output result.

The major problem with interval arithmetic is that, like floating point arithmetic performed with error analysis, it does not make the computations any more exact. Although the user may have faith in the bounds of the result, the bounds may not be sufficiently close to provide a useful answer.

2.1.4 Stochastic Rounding

Unlike the previous approaches, which either round to the nearest number, or strictly up and down to obtain bounds, the stochastic approach rounds numbers at each stage randomly. The final result is obtained by performing the desired computation several times using this stochastic rounding technique, and then using probability theory to estimate the true result.

Clearly this approach does not guarantee the accuracy of the result in the way that interval analysis might, but in general it will give a better result than ordinary floating point arithmetic. The stochastic rounding technique also gives probabilistic information about the reliability of the calculation.

2.1.5 Symbolic Approaches

Using a symbolic approach to real arithmetic involves manipulating expressions in terms of symbols representing variables, constants, and functions, rather than performing computations with the numbers themselves.

Whilst an expression is manipulated solely in terms of its symbols, it represents exactly the correct result. For example, integration performed using floating point arithmetic will only represent an approximation to the correct result, but if the integration can be performed symbolically, the resulting expression will represent the answer exactly.

Symbolic approaches to arithmetic are extremely useful for certain tasks because they can provide information about results that a numeric approach would not be able to give. For example, simplifying an expression symbolically may reveal some useful property not apparent from a numerical answer. Consider

$$4 \arctan \left(\sin^2 \left(\frac{1}{23} \right) + \cos^2 \left(\frac{1}{23} \right) \right)$$

Simplification reveals that this is equal to π. This is probably more useful than the result which would be achieved by floating point approximation. It is worth noting that there are many complicated symbolic expressions for which we can find no useful simplification.

Unfortunately, although the symbolic approach is extremely useful in certain applications, it is not general enough to replace other approaches. In particular, although an expression may represent the correct result, when it becomes necessary to evaluate the expression to get a numeric answer, an approach such as floating point or interval arithmetic will need to be used. The accuracy of the evaluated result will still suffer from the problems which apply to to the arithmetic operations used to compute it.

There is no reason in principle why exact real arithmetic techniques should not be used in conjunction with the symbolic approach.

2.1.6 Exact Real Arithmetic

Exact real arithmetic is a numerical approach to real number computation based on potentially infinite data structures such as streams. Its main feature is that it solves the problems of inaccuracy and uncertainty in floating point and interval arithmetic, and is applicable in some cases in which a symbolic approach would not be appropriate.

Although exact real arithmetic can be used to calculate results which can be guaranteed to be completely accurate, it does so at the expense of the efficiency afforded by more conventional methods. One reason for this is that in practice, infinite data structures such as streams are inherently expensive to manage when compared with the type of fixed sized data structure used to represent floating point numbers. In addition, during exact computations it is often found that a small portion of the result of a computation may depend upon a large amount of the input. These inefficiencies assoociated with exact real arithmetic may be an acceptable trade-off against the benefits. For example which do not consider a sufficient number of digits to determine the correct result will be inherently inaccurate. These issues are discussed further in chapter 7.

2.2 Approaches to Exact Real Arithmetic

In section 2.1.6 we mention a data structure used to represent infinite objects called a *stream*, which is central to many approaches to exact real arithmetic. Streams are infinite sequences, and are explored in more detail in section 3.2. Importantly, we will see that working with streams is fundamentally different from working with finite data structures like the mantissa of a floating point number. When using a data structure of finite length, it is possible to examine any element of the structure at any point. When using a stream, however, only a finite number of elements at the head of the stream may be accessed at a given time.

2.2.1 Representation Problems

When we perform exact real number computations, we may require a representation of infinite size. However when performing computations with structures such as infinite sequences of digits, one problem which is encountered is that there is no 'least significant' digit. This is because as the representation extends infinitely far to the right. This fact means that arithmetic operations on such sequences must be computed from left (most significant digit) to right (least significant digit). Most operations on floating point numbers, by contrast, require computation from right to left. Floating point addition, for example, is performed by starting with the least significant digits of the input numbers and working towards the most significant.

Given that we must implement arithmetic operations from left to right on streams, it quickly becomes apparent that representations such as binary and decimal are not adequate. Surprisingly, using these representations even simple operations such as addition and subtraction are not computable. For example, suppose we wish to compute the result $(a+b)$ of adding two numbers which start as follows:

$$a = 0.3333333\ldots \qquad \text{and} \qquad b = 0.6666666\ldots$$

It is not clear whether the first digit of the result here should be a one or a zero. If, for example, the numbers continued as shown here until 100^{th} digit of a, but at this point the next digit of a was greater than three then the result of the calculation would be greater than one. If the same digit in fact turned out to be less than three, the result would be less than one. In general it is not possible to compute an output digit without examining an infinite number of digits. Similar examples show that all integer bases suffer from the same problem.

In order to find algorithms for exact real number computation using infinite objects, we must use different representations. The representations used in this work are described next, and we also mention a number of other possible representations. Note that all the representations used for exact real arithmetic are equivalent to one another in the sense that it is possible to convert between them in a computable way. Some representations may be more efficient than others.

It is only possible to represent an countable subset of the real numbers in a computer using streams—the so called "computable reals".

2.2.2 Two Suitable Digit Representations

The two digit representations for exact real arithmetic we will use in this work are introduced here. These representations are defined fully in chapter 3. These are suitable representations which allow us to define algorithms to compute basic arithmetic operations, and also higher level transcendental functions.

The representations used require a high degree of redundancy, and in general we cannot compute equality or relational operators on them (although infinite sequences of digits using the standard representations already rejected are slightly redundant, and relational operators are not computable on them either).

The first representation uses one of the most intuitive ways to introduce further redundancy into the representation, which is by allowing negative digits. One way of doing this with the integer base B would be to allow the digits $\{-B+1, \ldots, -1, 0, 1, \ldots, B-1\}$ rather than just the postive ones. The signed binary representation, which is one of those used here, uses digits $\{-1, 0, 1\}$

The second representation used consists of a sequence of dyadic rationals, each of which is in the interval $[-1, 1]$ and is treated as 'pseudo-digit'. Notice that unlike in the first representation, however, there are an infinite number of these 'pseudo-digits'.

One of the motivations for these choices of representation is that they are the ones required to implement Alex Simpson's functional operations [26], which we want to implement and analyse. They also have the advantage that they are relatively easy to represent and understand because of their similarity to more usual unsigned digit representations. Many authors mention these approaches, especially the signed digit one, but as far as we know none develop the full set of arithmetic operations and transcendental functions for them.

2.2.3 Other Possible Representations

The digit representations mentioned in section 2.2.2 are by no means the only suitable ones. Section 1.2 lists the related work of a number of other authors, and a few of their representations are described here by way of comparison.

One interesting feature of the digit representations, including all the representations used in this work, is that they are *incremental*. An incremental representation is one in which extending the precision of a result can be performed without recomputing it from scratch. With the digit representations, we can extend the precision simply by computing more digits. Not all representations used for exact real arithmetic have this property.

Philipp Sünderhauf [27] observes that whilst incrementality would appear to be an intuitively useful property, this may not always be the case, even in situations where the incrementality is directly used. Some non-incremental representations report surprisingly high performance. Sünderhauf discusses this, and develops a "hybrid" representation from a non-incremental one which combines benefits of both incrementality and non-incrementality.

Continued Fractions

The continued fraction representation of a real r is a stream $[s_0, s_1, s_2, \ldots, s_i, \ldots]$ of integers such that:

$$r = \lim_{i \to \infty} s_0 + \cfrac{1}{s_1 + \cfrac{1}{s_2 + \cfrac{1}{\ddots + \cfrac{1}{s_i}}}}$$

One of the benefits of the continued fraction is that some numbers which are complicated to represent using a digit notation have remarkably simple continued fraction representations. For example:

$$\begin{aligned} \phi = \tfrac{1+\sqrt{5}}{2} \quad &= [1, 1, 1, 1, 1, 1, \ldots] \\ e \quad &= [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, \ldots] \end{aligned}$$

The continued fraction representation is incremental. Jean Vuillemin [31] discusses the performance of an implementation using this representation.

B-adic Number Stream

Valérie Ménissier-Morain [20] uses a representation of reals as streams of integers. A real r is represented by a stream of integers $(c_n)_{n \in \mathbb{N}}$ with the property that $|r - c_n B^{-n}| < B^{-n}$.

$$r = \lim_{n \to \infty} \left(\frac{c_n}{B^{-n}} \right)$$

This representation is not incremental. If we evaluate a result, and subsequently require the same result to a higher precision, we must examine a later element in the stream. This is computed from scratch, unlike the dyadic digit representation used in this work in which later digits simply refine earlier results. Despite its non-incremental nature, this representation reportedly performs well in practice.

Linear Fractional Transformations

The linear fractional transformation representation for exact real numbers is described in [10, 23, 24]. Linear fractional transformations—also known as Möbius transformations—are transformations which are of the following forms:

One dimensional linear fractional transformationos (lft's) are generalisations of linear maps of the form:

$$t(x) = \frac{ax + c}{bx + d}$$

Two dimensional lft's are transformations of the following form:

$$t(x, y) = \frac{axy + cx + ey + g}{bxy + dx + fy + h}$$

One useful property of linear fractional transformations is that they can be represented as matrices, and their composition computed using matrix multiplication.

It is possible to use lft's to represent a non negative extended real number as the intersection of an infinite composition of matrices with integer co-efficients applied to the interval $[0, \infty]$.

$$\bigcap_{n \geq 0} M_0 \cdot M_1 \cdot M_2 \cdot M_3 \cdot M_4 \cdot M_5 \cdots ([0, \infty])$$

Algorithms for arithmetic operations and transcendental functions can then be developed using this representation.

One advantage with using the linear fractional transformation representation, especially over representations such as the ones using an infinite stream of digits, is that many of the elegant continued fractions for mathematical constants and functions can be used almost directly. It is claimed that this representation is the most efficient for exact real arithmetic, although this claim is as yet unsubstantiated.

3. Representations

In chapter 2 we discussed real arithmetic in general and the inadequacy of the usual binary or decimal representations for real number computation. In this chapter we define the representations that will be used in this work and some of their general properties. We also briefly discuss the issues that arise when we develop algorithms for these representations, show how to convert between the two basic stream representations, how to convert a decimal numeral into the signed binary representation, and how to convert a finite portion of a signed binary stream back into decimal.

Two basic representations are used for reals in the closed interval $[-1, 1]$; streams of signed binary digits and streams of dyadic rationals treated as digits. These representations are then used to derive a representation for the whole real line, and another which simplifies the implementation of transcendental functions later on.

The signed binary stream representation is used as the main representation for the calculator. However the use of dyadic rationals as digits greatly simplifies certain algorithms, in particular division and the functional operations, and is used as an intermediate representation in these cases. In chapter 7, we show how algorithms for the same operation using these two superficially similar representations can perform very differently, and the observations from the analyses performed here form one of the justifications for not using the dyadic stream representation more extensively.

3.1 Terminology and Notation

The following terminology and notation is used here.

A line above a number is used to denote its negation. For example the digit minus one (-1) is usually written $\overline{1}$.

An arrow above a digit is used to denote an infinite stream of that digit. For example $\overrightarrow{1}$ denotes an infinite stream of ones.

A double colon (::) is used to separate elements of a list or stream. The last element printed represents the tail of the list or stream unless stated otherwise. Hence $(1 :: \overline{1} :: x)$ is a stream whose first two elements are the digits one and minus one, and the remainder of which is a stream which would be referred to as x.

The term *numeral* is used to refer to the symbol or group of symbols representing a *number*. A number is a mathematical object, whereas a numeral is a syntactic object. When working with numerals and numbers in equations where the potential for confusion exists, we use semantic brackets $[\![x]\!]$ to denote the number or interval represented by the numeral x.

If x is one of the representations of reals defined in this chapter, $[\![x]\!]$ is simply the real number it represents. When working with exact reals as infinite streams, however, we often know only a finite portion of the stream. We define the *range* of a finite portion of a stream representing a real number to be the set of all possible reals that may be represented by a stream starting with the elements known already, but with all possible suffixes.

The symbol $\oplus$ is used to represent the average operation, $x \oplus y = (x + y)/2$. Average is a mathematical operation, but the symbol is also used to denote the result of applying an average algorithm to two numerals. In the latter case, the actual algorithm required will depend on whether the numerals are dyadic digits (see section A.2), a digit and a stream (section 3.6.2), or two streams (sections 4.2.1 and 4.3.1).

Multiplication will be represented using the symbol $\times$, or as a single dot $(\cdot)$. In general we will use $\times$ when both operands are numerals in stream representation, and the dot notation for the multiplication of two numbers, or when the operands are numerals and one or both are digits.

3.2 Streams

Section 2.1.6 briefly introduced the notion of a *stream*, which is used to implement the two basic representations considered in this work. We discuss streams in more detail here.

A stream is simply an infinite sequence. Clearly a machine with finite resources cannot store an infinite sequence in memory. However we can represent some infinite sequences as programs which successively generate elements of the sequence. We refer to these sequences as *computable streams*, and the programs which generate them are used extensively when performing exact real arithmetic on a computer.

As mentioned in section 2.2, the important point to note about streams is that because the elements are evaluated successively, we can only examine a finite initial sub-sequence of the stream. Because computers are finite and we can only store a finite number of digits in memory at any one time, we must decide each output digit on the basis of a finite number of input digits from the start of the sequence representing the number. This is quite different from the way arithmetic is usually performed on finite strings of digits in which the entire numeral can be accessed.

A computable stream is normally implemented as a function which takes no parameters, but which, when evaluated, returns the first element *(head)* of the sequence and a new function computing the remainder *(tail)* of the infinite sequence. The language ML could be used to define a stream of elements of type `'a` as follows:

```
datatype 'a Stream = Stream of unit ('a * ('a Stream))
```

One feature of this particular datatype is that it requires explicit evaluation of the function to obtain each element of the sequence. Lazy functional languages such as Haskell [22] allow streams to be handled more neatly as lazy lists. This is one of the motivations for using Haskell to implement the calculator (see section 6.2.1). For example, in Haskell one could implement a function to produce a stream counting from some number n as follows:

```
from n = (n : from (n+1))

> from 0
[ 0, 1, 2, 3, 4, ... <ctrl-c>
```

The `from` function generates the digit at the head of the sequence, and uses a function to generate the remainder of the sequence by recursively calling itself. When evaluated the function will generate digits from the sequence indefinitely unless interrupted.

An operation on this stream might be defined as follows:

```
add (a:x) (b:y) = (a + b : add x y)

> add (from 0) (from 1)
[ 1, 3, 5, 7, 9, ... <ctrl-c>
```

This style of programming is used extensively in the implementation, as described in chapter 6.

3.3 Representing Reals as Streams

In section 2.2.1, we showed why an infinite sequence of binary or decimal digits would be inadequate for the purposes of exact real arithmetic. We now define the alternative representations used here using streams.

These representations are of reals in a closed interval. In section 3.4 we show how to extend these representations to the whole real line.

3.3.1 Signed Binary Digit Streams

The first representation makes use of a stream of signed binary digits $\{\overline{1}, 0, 1\}$. This representation is also known as the *redundant binary representation* on $\mathbb{R}$ [17]. The real number represented by the stream $x = (d_1 :: d_2 :: d_3 :: ...)$ is defined as:

$$[\![x]\!] = \sum_{i=1}^{\infty} d_i \cdot 2^{-i} \qquad = \frac{d_1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + \frac{d_4}{2^4} + \ldots$$

Hence $[\![x]\!] \in [-1, 1]$. There is a unique representation for the numbers minus one ($\overrightarrow{-1}$) and one ($\overrightarrow{1}$), and infinitely many representations for all other numbers in this range.

We also define the range (see section 3.1) of a finite portion of the stream as follows: Let x' be a list containing the first m digits of x, $[\![x']\!]$ is the range of x':

$$\begin{aligned} x' &= d_1 :: d_1 :: d_2 :: \ldots :: d_m \\ [\![x]\!] &= \left[\left(\sum_{i=1}^{m} d_i . 2^{-i} \right) - 2^{-m}, \left(\sum_{i=1}^{m} d_i . 2^{-i} \right) + 2^{-m} \right] \end{aligned}$$

Figure 3.1 shows the range of values represented by the signed binary streams starting with each of the three possible digits.

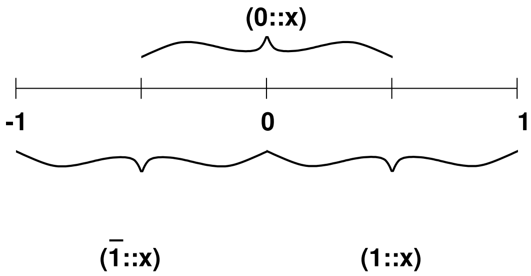

Figure 3.1: The range of streams starting with different digits

There are two further interesting observations to make. The first is that the following identities exist. We use them frequently when developing algorithms using signed binary streams:

$$\begin{aligned} 1 :: \overline{1} :: x &= 0 :: 1 :: x \\ \overline{1} :: 1 :: x &= 0 :: \overline{1} :: x \end{aligned}$$

The second observation is that the signed binary representation has a fixed rate of convergence. In other words, every new digit in the stream refines the range of possible values taken by the stream at a fixed rate.

3.3.2 Dyadic Rational Streams

The second basic representation used is a stream of dyadic rationals in the range $[-1, 1]$:

$$\frac{a}{2^b} \cap [-1, 1] \qquad \text{where} \quad a, b \in \mathbb{Z}$$

Observe that there are a infinitely many such numbers. We can define algorithms for a number of simple operations and relationships on these dyadic rationals.

We treat these dyadic rationals as if they were digits, and from now on term them *dyadic digits*. Dyadic digits may be represented in several ways. One possibility is to use a datatype which is either a symbol representing one, minus one, or is an arbitrary but finite length string or binary digits representing a binary fraction. ie:

$$\begin{aligned} d &= (a_1, a_2, a_3, ...a_n) \\ [\![d]\!] &= [\![(a_1, a_2, a_3, ...a_n)]\!] = \textstyle\sum_{i=1}^{n} a_i \cdot 2^{-i} \end{aligned}$$

Another alternative is to represent the digit as a pair of arbitrary length integers. ie.

$$\begin{aligned} d &= (a, b) \\ [\![d]\!] &= [\![(a, b)]\!] = a/2^b \end{aligned}$$

In the latter case, any given dyadic rational may be represented in several ways (eg. $\frac{1}{2^1} = \frac{2}{2^2} = 72057594037927936/2^{57}$). One solution to the problem of having an unnecessarily large representation for a dyadic rational is to force it into its lowest terms by ensuring that zero is represented as $(0, 0)$, and that in all other cases the numerator a is odd and the denominator positive. This prevents the integers representing the dyadic digit swelling more than is strictly necessary.

The choice of internal representation used in the implementation is discussed in section 6.2.5, and we give algorithms for the basic operations using these digits alluded to above in appendix A.

We now define the dyadic digit stream representation in exactly the same way as we did the signed binary digit streams, except that instead of just using three digits $-1, 0, 1$, we use instead the dyadic digits described above.

Observe that like signed binary streams, the dyadic stream representation has a fixed rate of convergence. Dyadic digits, however, can incorporate significantly more information than signed binary digits. This can be extremely useful for simplifying certain algorithms, and can both improve or destroy the performance of certain calculations. The performance of the algorithms is discussed in detail in chapter 7.

3.4 (Mantissa, Exponent) Representation

The signed binary and dyadic digit streams described in sections 3.3.1 and 3.3.2 represent reals in the range $[-1, 1]$. In fact the choice of $[-1, 1]$ as endpoints is arbitrary, as we could easily interpret a given stream as a real in any closed interval.

To illustrate, suppose we have a numeral x which we would interpreted as the number $[\![x]\!] \in [-1, 1]$. We could just as easily interpret x as a number in the range $[a, b]$ as

$$a + \frac{(b-a)([\![x]\!]+1)}{2}$$

When it is necessary to represent reals on the whole real line, however, the stream representation alone is not sufficient. Perhaps the simplest solution to this problem is to introduce a (mantissa, exponent) representation similar to a floating point number. We do this using a mantissa which is a stream of either signed binary or dyadic digits, and and exponent is an integer of unrestricted size. For simplicity, the exponent represents a power of two, although powers of other numbers could also be used.

Hence, given a numeral x with mantissa m and exponent e, we have:

$$[\![x]\!] = [\![m]\!] \times 2^e$$

Most algorithms for operations on reals represented using a mantissa and exponent in this way reduce to two simple and independent operations on the mantissa and exponent.

3.5 Nested interval Representation

The digit stream representations described in sections 3.3.1 and 3.3.2, and the (mantissa, exponent) representations derived from them, have the property that as we evaluate more digits, the range of possible numbers represented by the stream converges at a constant rate.

When computing the limits of Cauchy sequences as we do in section 5.2, it is useful to use a representation which does not have this property (ie. has a variable rate of convergence). One such representation a stream of nested intervals.

When implementing such a representation, the end points of the intervals may be either rationals or other exact real numbers. The the latter approach has been adopted here for simplicity (because we have already implemented algorithms for exact reals). Each element of the stream x is a nested interval expressed as a pair of numbers as follows:

$$x = [a_1, b_1] :: [a_2, b_2] :: [a_3, b_3] :: \ldots$$

where $[a_i, b_i] \supseteq [a_{i+1}, b_{i+1}]$ for all values of $i \geq 1$, and $(b_i - a_i) \to 0$ as $i \to \infty$. The semantics of x are defined as

$$\begin{aligned} [\![x]\!] &= \lim_{i\to\infty} a_i = \lim_{i\to\infty} b_i \\ \{[\![x]\!]\} &= \bigcap_{i=0}^{\infty} [a_i, b_i] = \Big[[\![x]\!], [\![x]\!]\Big] \end{aligned}$$

For the purposes of implementing the algorithm to compute the limits of Cauchy sequences, we use the signed binary (mantissa, exponent) representation to hold the end-points of each interval which we can then convert into a single numeral in signed binary (mantissa, exponent) representation as described in section 5.1. An exponent is stored at every end-point of every interval for simplicity, although because of the convergent properties described above, every number in each interval could be represented using the highest exponent seen in the interval at the head of the sequence.

3.6 Algorithm Design

We now discuss a number of considerations which apply when developing algorithms using the representations defined in this chapter, and some useful techniques and properties of the representations which we can use when developing these algorithms.

3.6.1 Considerations

When designing algorithms for arithmetic and related operations, there are a number of considerations:

Convergence: It must be possible to determine a new output digit after examining a finite number of digits from the input streams. If we need to examine a potentially infinite number of input digits in order to determine an output digit, then we do not have an algorithm. There are a number of subtle ways in which such a situation can arise, and care must be taken.

Correctness: The algorithm must compute the correct result.

Efficiency: The efficiency of the algorithm is also a consideration. Ideally we would like an algorithm that computes the result using as few digits of input as possible and which does not require manipulation of excessively large data structures. In addition, we would rather the algorithm did not branch excessively. For example, the operation $(x \oplus 0 \cdot y)$, where $\oplus$ is the average operation, if näively implemented, would require the evaluation of each digit of y, a multiplication by zero on each digit, and a stream average operation. In fact this could be more efficiently implemented by using the 'cons' algorithm to prefix a zero to the stream representing x (ie. return $(0 :: x)$—see section 3.6.2). Different algorithms for the same operation can have very different levels of performance. The issue of the efficiency of the algorithms is discussed in more detail in chapter 7.

3.6.2 Useful Properties

In order to construct the algorithms using these representations we also make use of a number of general techniques and useful properties.

Range Analysis

It is often possible to analyse the range of a finite portion of given stream, and use this information to determine output digits for an algorithm computing some operation applied to this stream. For example, suppose we are computing $(1-x)$, and we know that x is of the form $(1 :: x')$, we can say that $[\![x]\!] \in [0,1]$, and hence $[\![1-x]\!] \in [0,1]$. We can represent all reals in this range using a stream starting with the digit one, and so we can output this digit without examining further digits of x. It was thus possible to generate one output digit by examining only one input digit. The example given is a specific case, but this general principle may be applied in numerous ways.

In some cases we can extend this technique as follows. Suppose we know by range analysis that $x \in [-\frac{1}{2}, \frac{1}{2}]$, but that it need not necessarily start with the digit zero. If we wish to compute $2x$, a legal operation given the range of x, the simplest way is to perform a left shift by removing a zero from the head of the stream representing x. Obviously, however, this method will not work if the first digit of x is not a zero. Fortunately we can define a function which will re-represent the stream starting with a specified digit, in this case zero, which we can then remove. This function for signed binary streams is described in section 4.1.4, and is referred to in this work as the 'p' function.

Relationship between 'cons' and average

The 'cons' operation is a primitive operation on lists and streams. Operations on exact real numbers, however, are defined in terms of mathematical operations. Fortunately the semantics of the stream of digits representation of reals leads to a useful relationship between 'cons' and average which simplifies later definitions.

We define average as one would expect, using the symbol $\oplus$ so that

$$a \oplus b = \frac{a+b}{2}$$

Now, suppose we wish to compute the average $(a \oplus x)$ of the digit a and the stream x. It is useful to observe that

$$a \oplus [\![x]\!] = [\![a :: x]\!]$$

because $a \oplus [\![x]\!] = \frac{a}{2} + \frac{[\![x]\!]}{2} = \frac{a}{2} + [\![0 :: x]\!] = [\![a :: x]\!]$. Hence the result of averaging a digit and a stream may be performed by generating a new stream using the 'cons' operation to attach the digit at the head of the original stream.

Properties of average

It is also useful to observe the following properties of average.

$$\begin{array}{rcll}
x \oplus x & = & x & \text{Idempotency} \\
x \oplus y & = & y \oplus x & \text{Commutativity} \\
(a \oplus x) \oplus (b \oplus y) & = & (a \oplus b) \oplus (x \oplus y) & \text{Exchange law} \\
a \cdot (x \oplus y) & = & (a \cdot x) \oplus (a \cdot y) & \text{Distributivity} \\
a \oplus (x \oplus y) & = & (a \oplus x) \oplus (a \oplus y) & \text{Self Distributivity}
\end{array}$$

Notice that average is **not** associative:

$$x \oplus (y \oplus z) \neq (x \oplus y) \oplus z$$

3.7 Conversion between Stream Representations

The signed binary representation is generally more appropriate than the dyadic representation for implementing most operations. This is because firstly there are a small finite number of digits, which means we can examine different combinations of digits at the start of input streams and treat them as individual cases, and also each signed binary digit occupies a fixed amount of space and cannot swell in size like the dyadic digit representation. Certain algorithms, however, are easier to define using the dyadic digit representation, either as an intermediate representation, or for both input and output.

The fact that we use two representations means that it is sometimes necessary to convert between them. We show here how this is performed for the stream representations. The (mantissa, exponent) representations are converted by performing the appropriate conversion on the mantissa and leaving the exponent unchanged.

Converting signed binary streams to dyadic streams is trivial, and is performed by generating a stream with dyadic ones, zeros, and minus ones in place of the corresponding signed binary digits.

Converting dyadic streams into signed binary representations is more complex. It is necessary to examine two digits of the dyadic input stream to decide one digit of output. The conversion function d_to_s is defined as follows:

$$\mathrm{d\_to\_s}(a :: b :: x) = \begin{cases} 1 :: \mathrm{d\_to\_s}(4(a' - \frac{1}{4}) :: x) & \text{if } a' >= \frac{1}{4} \\ 0 :: \mathrm{d\_to\_s}(4a' :: x) & \text{if } -\frac{1}{4} < a' < \frac{1}{4} \\ \overline{1} :: \mathrm{d\_to\_s}(4(a' + \frac{1}{4}) :: x) & \text{if } a' \leq -\frac{1}{4} \end{cases}$$
$$\text{where } a' = a \oplus \frac{b}{2}$$

The conversion function works because the value attached to the remainder of the stream x using the 'cons' operation in the recursive call to d_to_s is always in the range $[-1, 1]$, and as such can be represented by a single dyadic digit.

We prove of correctness as follows. The value a' represents the value of the first two digits of input. The number represented by the whole input stream is $a' \pm \frac{1}{4}$ because the remainder of the stream (x), which represents a value in the range $[-\frac{1}{4}, \frac{1}{4}]$. Now, if a' is greater or equal to a quarter, the entire input stream is in the range $[0, 1]$ and can be represented by the signed binary stream whose first digit is one. Similarly if a' is a value between minus a quarter and a quarter the whole stream is in the range $(-\frac{1}{2}, \frac{1}{2})$ and can be represented by an output stream whose first digit is zero, and if a' is a less than or equal to minus a quarter the whole stream is in the range $[-1, 0]$ and can be represented by an output stream whose first digit is minus one. We output the appropriate digit in each case, and convert the remainder of the dyadic input stream by attaching the remainder of the output digit subtracted from a' to x and calling the conversion function recursively.

3.8 Conversion to and from decimal

A human user of a calculator is likely to want to input numbers in decimal, and to view results in decimal. For this reason we include conversions to and from decimal in the calculator.

Converting a decimal input into signed binary (mantissa, exponent) representation is possible, but unfortunately converting from a signed binary input back to decimal is not. This is because it is not necessarily possible to determine even the first digit of the decimal representation of the number without examining a potentially infinite amount of the signed binary input (see section 2.2.1). The solution adopted is to convert only a finite portion of the input stream to decimal. This approximates the result to a known accuracy. Printing exact decimal lower and upper bounds would be a trivial extension.

The operations described here are to and from the signed binary representations only. Conversions between dyadic digit representations and decimal can be performed by going via the signed binary representations (see section 3.7).

3.8.1 Decimal to Signed Binary

The algorithm used here is fairly straightforward conversion from decimal to binary, and is described in [17]. In order to convert the input, we separate the integer and fractional parts of the input, convert each one individually, and then combine them using addition. The decimal inputs are assumed to be of finite length.

Decimal integers to signed binary

Conversion from decimal (base 10) can be performed using binary (base 2) arithmetic as follows: Suppose we have the decimal integer $(u_m \ldots u_1 u_0)$. We compute the binary representation U as

$$U = ((\cdots(u_m \cdot 10 + u_{m-1}) \cdot 10 + \cdots) \cdot 10 + u_1) \cdot 10 + u_0$$

This can be easily computed using the signed binary (mantissa, exponent) operations which we develop algorithms for in chapter 4. We extract the finite decimal integer, reverse the digits, and present them to a recursive algorithm which has a base case which returns zero when the input is exhausted, but otherwise computes number represented by the tail of the list, multiplies this by ten, and adds the number at the head of the list:

$$\begin{aligned} \mathrm{decSB\_int}([\,]) &= 0 \\ \mathrm{decSB\_int}(a :: x) &= 10 \cdot \mathrm{decSB\_int}(x) + a \end{aligned}$$

Decimal fractions to signed binary

Suppose we wish to compute the signed binary representation U of the decimal fraction $(0.u_{-1}u_{-2}u_{-3} \ldots u_{-m})$ using signed binary operations, we simply evaluate

$$U = u_{-1} \cdot 10^{-1} + u_{-2} \cdot 10^{-2} + u_{-3} \cdot 10^{-3} + \cdots + u_{-m} \cdot 10^{-m}$$

To compute this more efficiently, we observe

$$U = (((u_{-m}/10 + u_{-(m-1)})/10 + \cdots + u_{-2})/10 + u_{-1})/10$$

We can now use a simple recursive algorithm using the average and division by an integer operations which evaluates to zero on an empty input, but otherwise computes:

$$\text{decSB\_frac}(a :: x) = \frac{a + \text{decSB\_frac}(x)}{10} = \frac{a \oplus \text{decSB\_frac}(x)}{5}$$

Because we cannot represent decimal digits in signed binary streams, we replace a in the right hand side of the expression by a stream which represents $a/2^4$, and multiply the entire result by 2^4. Multiplication and division by powers of two can be performed easily using shift operations.

3.8.2 Signed Binary to Decimal

In order to convert signed binary numerals into decimal we proceed as follows. First of all we separate the integer and fractional parts of the signed binary input using the exponent of the (mantissa, exponent) representation. The integer and fractional part are handled separately. The integer part of the signed binary representation is finite, and can be converted into decimal relatively easily. In order to convert the fractional part we first convert it from a signed binary stream into a signed decimal stream. Then we take a finite portion of this and change the signed decimal into decimal. Lastly we combine the integer and fractional part, handling and overflow from the fractional to the integer part, and generate the complete decimal output.

The algorithms used here are modifications of those given by Knuth [17] for converting between bases using unsigned digit representations.

We use the notation $\text{int}(x)$ to denote the operation which integer part of a real x, and $\text{frac}(x)$ the operation which returns its fractional part.

Signed binary integers to decimal

We can easily extract the integer part of a signed binary (mantissa, exponent) represented real by examining the exponent, and if it is positive, extracting that number of digits from the head of the stream. The remainder of the stream is the fractional part and is handled separately.

Converting the signed binary integer u to its decimal representation $U_n \ldots U_3U_2U_1U_0$ is performed as follows:

$$\begin{aligned}
U_0 &= u \bmod 10 \\
U_1 &= \mathrm{int}(u \text{ div } 10) \bmod 10 \\
U_2 &= \mathrm{int}(\mathrm{int}(u \text{ div } 10) \text{ div } 10) \bmod 10 \\
&\vdots \\
U_n &= \ldots
\end{aligned}$$

This differs slightly from the algorithm presented in [17] which uses the floor function instead of taking the integer part at each stage. The reason for this is that Knuth's algorithm only deals with unsigned digits. The 'int' function behaves like the floor function for positive digits, but the ceiling function for negative digits (taking the integer obtained by rounding towards zero). This is the correct behaviour because converting negative digits is equivalent to negating these digits to make them positive and then negating the output digit at each stage.

This part of the algorithms requires 'div' and 'mod' operations to be performed on a finite signed binary list. This can be performed using a modification of the division of a stream by an integer algorithm described in section 4.3.3. It is changed so that the input list 'div' the denominator is the output which would be returned from the original algorithm, and the 'mod' portion is the remainder to be carried which is found when the end of the input list is reached.

Signed binary fractions to signed decimal fractions

Conversion to signed decimal is performed as follows. Given the signed binary input u, we compute the decimal result $0.U_{-1}U_{-2}U_{-3}\ldots$ as:

$$\begin{aligned}
U_{-1} &= \mathrm{int}(10 \cdot u) \\
U_{-2} &= \mathrm{int}(10 \cdot \mathrm{frac}(10 \cdot u)) \\
U_{-2} &= \mathrm{int}(10 \cdot \mathrm{frac}(10 \cdot \mathrm{frac}(10 \cdot u))) \\
U_{-4} &= \ldots
\end{aligned}$$

We implement this using a signed binary stream, representing a number in the interval $[-1, 1]$. This means we cannot represent the value ten to perform multiplication by ten using the signed binary stream multiplication algorithm. It is possible, however, to multiply instead by the value $10/2^4$, which is in the interval $[-1, 1]$ and can be represented as a signed binary stream. This produces the correct result provided we interpret the output as being multiplied by 2^4.

The algorithm for converting from signed binary into signed decimal is therefore as follows:

$$\begin{aligned} \mathrm{sbDec\_s}(x) &= (8 \cdot a + 4 \cdot b + 2 \cdot c + d) :: \mathrm{sbDec\_s}(x') \\ &\quad \text{where } (a :: b :: c :: d :: x') = x \times (1 :: 0 :: 1 :: \overrightarrow{0}) \end{aligned}$$

In fact the output stream is in fact not true signed decimal, because we can in fact get the digits -10 and 10 appearing at the first digit if there is an overflow. However this is not a problem because we can easily check these digits in the first output digit when the conversion is complete, and handle the overflow appropriately if necessary.

Signed decimal fractions to decimal

In order to convert the signed decimal stream into decimal, we take a finite portion of the stream and find its sign by searching for the first non-zero digit. If the number is negative, we print a minus sign and negate all digits in the signed decimal stream.

We then examine the last digit of the list. If it is negative, we add 10 to the digit and pass a negative carry back by subtracting one from the digit to the left. The process is then repeated.

4. Basic Operations

In chapter 3 we defined two basic representations for reals in a closed interval using streams of signed binary digits and streams of dyadic digits. We then used these to define (mantissa, exponent) representations for numbers on the whole real line and a representation of reals as a stream of nested intervals. We also discussed some properties of these representations, and considerations which apply when we define algorithms on them.

In this chapter we give algorithms for the basic arithmetic operations and some useful auxiliary functions used to implement them. The algorithms given here are used to implement the basic arithmetic operations in the calculator, and form the basis for higher level transcendental functions and the functional operations in chapter 5.

The main representation used to implement operations within the calculator is the signed binary (mantissa, exponent) representation. The (mantissa, exponent) representation is required because the stream representations are not closed under addition, subtraction, or division. The signed binary representation is chosen because it is simpler to use when developing certain algorithms, and because it does not suffer from the problem of dyadic digit swell discussed in chapter 7.

In order to implement the (mantissa, exponent) operations addition, subtraction, and multiplication, we use the stream operations average and multiplication (as the stream representations are closed under average and multiplication). These stream operations are also used to implement other algorithms including division.

Although the signed binary stream operation is the main representation used here, the dyadic digit stream representation is also extremely useful as an intermediate representation to simplify the division algorithm, and it is used in Alex Simpson's algorithms for functional operations [26]. In addition, the average and multiplication algorithms for the dyadic stream representation are considerably simpler than those for the signed binary representation, and it is helpful to understand these before developing equivalent algorithms for the signed binary representation.

The chapter is structured as follows. We first develop algorithms for the auxiliary operations, some of which are used with both dyadic and signed binary stream representations, and some of which are only developed for the signed binary streams. Next we give the algorithms for average and multiplication using the dyadic stream representation, and then for the signed binary stream representation. We also give and algorithm for division of a signed binary stream by an integer. Lastly we give the (mantissa, exponent) algorithms for the basic operations addition, subtraction, multiplication, and division.

The algorithms for the primitive operations on dyadic digits using the representation adopted in the implementation, which are required for the algorithms for auxiliary functions and basic operations on dyadic streams, are given in appendix A.

Presented here are algorithms for division of two numbers in the signed binary (mantissa, exponent) representation, and for the direct multiplication of two streams in signed binary representation without going via the dyadic stream representation. These two algorithms are entirely my own work. The remaining algorithms are already known, and the majority of these were described, at least in some a general form, in personal communications with Alex Simpson and Martín Escardó.

4.1 Auxiliary operations

4.1.1 Negation

Negation of a signed binary stream is achieved by simply negating each individual digit in the stream. If $[\![x]\!] = \sum_{i=1}^{\infty} \frac{d_i}{2^i}$:

$$-[\![x]\!] = -\sum_{i=1}^{\infty} \frac{d_i}{2^i} = \sum_{i=1}^{\infty} \frac{-d_i}{2^i}$$

4.1.2 Multiplication of a stream by a digit

Multiplication of a stream x by a digit a is performed by simply multiplying each digit of the stream x by the specified digit a.

If the digit is a signed binary digit, the result will be either x, the zero stream ($\overrightarrow{0}$), or $-x$. If dyadic streams and digits are used, a dyadic digit multiplication algorithm must be used (see section A.3).

4.1.3 Addition or subtraction of one from stream

It is possible to define two simple functions to add or subtract the number one from the number represented by a from a stream of signed binary digits (provided the stream represents a number in a suitable interval). Let us define these functions as follows:

$$
\begin{aligned}
f(x) &= \min(x+1,1) = \begin{cases} 1 & \text{if } x > 0 \\ x+1 & \text{if } x \leq 0 \end{cases} \\
g(x) &= \max(x-1,-1) = \begin{cases} -1 & \text{if } x < 0 \\ x-1 & \text{if } x \geq 0 \end{cases}
\end{aligned}
$$

These functions may be implemented as follows:

$$
\begin{aligned}
f(1 :: x') &= \overrightarrow{1} & \qquad g(1 :: x') &= \overrightarrow{-1} \\
f(0 :: x') &= 1 :: f(x') & \qquad g(0 :: x') &= \overline{1} :: g(x') \\
f(\overline{1} :: x') &= 1 :: x' & \qquad g(\overline{1} :: x') &= \overline{1} :: x'
\end{aligned}
$$

The proof of correctness for such definitions is as follows. Let us consider the function f applied to a numeral x. The proof for the function g is similar. We use $[\![x]\!]$ to denote the number represented by the numeral x. We examine the three cases in turn:

Input x starts with the digit 1: If the input x starts with a one we know that $[\![x]\!] \geq 0$. Hence we can output an infinite stream of ones ($\overrightarrow{1}$) by the function definition.

Input x starts with the digit 0: If the input is x is of the form $(0 :: x')$, we know that x represents the value $[\![x']\!]/2$. Now, using the properties described in section 3.6.2:

$$\begin{aligned} f(x) &= \min\left(1 + \frac{[\![x']\!]}{2}, \overrightarrow{1}\right) = \min\left(1 \oplus (1 + [\![x']\!]), 1 \oplus \overrightarrow{1}\right) \\ &= 1 \oplus \min(1 + [\![x']\!], \overrightarrow{1}) \\ &= 1 :: f(x') \end{aligned}$$

Input x starts with the digit $\overline{1}$: If the input x is of the form $(\overline{1} :: x')$, we can simply add the number one at the first digit and return $(1 :: x')$:

$$\begin{aligned} x &= (\overline{1} :: x') = -\frac{1}{2} + \frac{[\![x']\!]}{2} \\ f(x) &= \min\left(1 + \left(-\frac{1}{2} + \frac{[\![x']\!]}{2}\right), \overrightarrow{1}\right) = \min\left(\frac{1}{2} + \frac{[\![x']\!]}{2}, \overrightarrow{1}\right) = (1 :: x') \end{aligned}$$

4.1.4 The 'p' function

We now give another useful algorithm for a function which we will dub the 'p' function. It is used extensively in situations where you can perform an analysis to determine the range of the output of some operation using the method described in section 3.6.2, but do not know what the first digit of a resulting stream will be. It is used in a number of algorithms, including for example the division algorithm (section 4.4.4), and the algorithm used to compute the limits of Cauchy sequence described in chapter 5).

The 'p' function re-expresses a stream starting with an unknown digit as a stream which, when appended to a specified digit, represents the same number as the original stream. In other words, the 'p' function of the digit a and stream x works as follows:

$$\mathrm{p}(a, x) = x' \qquad \text{where} \quad [\![a :: x']\!] = [\![x]\!]$$

The caveat associated with using the 'p' function is that if it is impossible to re-express the stream starting with the specified digit, the result will be undefined. The 'p' function removing the digit a from the stream x may be defined as follows:

$$\mathrm{p}(a,x) = \begin{cases} -1 & \text{if } 2\cdot[\![x]\!] - a \leq -1 \\ 2\cdot[\![x]\!] - a & \text{if } -1 \leq 2\cdot[\![x]\!] - a \leq 1 \\ 1 & \text{if } 2\cdot[\![x]\!] - a \geq 1 \end{cases}$$

The algorithm may be implemented simply by examining cases. Where the start digit of the stream is equal to the digit being removed, the result is simply the tail of the input stream. If the digit at the head of the stream differs by one from the digit to be removed, the functions to compute $(x-1)$ and $(x+1)$ defined in section 4.1.3 may be used. If the digits differ by two (ie. 1 and $\overline{1}$), the resulting stream will be either minus one (-1) or one (1). Hence:

$$\begin{array}{lcllcllcl} \mathrm{p}(1, 1 :: x') & = & x' & \mathrm{p}(0, 1 :: x') & = & 1 - x' & \mathrm{p}(\overline{1}, 1 :: x') & = & \overrightarrow{1} \\ \mathrm{p}(1, 0 :: x') & = & x' - 1 & \mathrm{p}(0, 0 :: x') & = & x' & \mathrm{p}(\overline{1}, 0 :: x') & = & 1 - x' \\ \mathrm{p}(1, \overline{1} :: x') & = & \overrightarrow{-1} & \mathrm{p}(0, \overline{1} :: x') & = & x' - 1 & \mathrm{p}(\overline{1}, \overline{1} :: x') & = & x' \end{array}$$

4.2 Dyadic Stream Operations

We give algorithms for computing the average of two streams of dyadic digits, and the result of multiplying two streams of dyadic digits.

4.2.1 Average

Computing the average two streams of dyadic digits x and y can be performed using a relatively simple recursive algorithm. Here we examine the first digit of each of these two input streams, and then use the properties given in section 3.6.2 to derive such an algorithm which returns their average as a new dyadic digit stream.

Suppose x is of the form $(a :: x')$ and y is of the form $(b :: y')$. We use the observation that a prefixing a digit to a stream using the 'cons' operation may be interpreted as the average of the numbers represented by the digit and the stream, and then apply the exchange law to say:

$$\begin{aligned} x \oplus y &= (a :: x') \oplus (b :: y') \\ &= (a \oplus x') \oplus (b \oplus y') \\ &= (a \oplus b) \oplus (x' \oplus y') \end{aligned}$$

Note that in the third equation the term $(a \oplus b)$ is the average of two dyadic digits, and $(x' \oplus y')$ is the average of two streams. Dyadic digits are closed under average, and hence the number $a \oplus b$ can represented in a single dyadic digit. If we do this, the average of these two terms will be of a single dyadic digit and a dyadic digit stream. We use this fact, and the property that a digit averaged with a stream can be implemented by using the 'cons' operation to prefix the digit to the stream, to give us the following:

$$x \oplus y = (a \oplus b) :: (x' \oplus y')$$

This leads to a simple recursive algorithm, expressed here as the function average, for computing dyadic stream average. It makes use of the algorithm for computing the average of two dyadic digits, digit_av, which is described in section A.2.

$$\text{average}(a :: x', b :: y') = \text{digit\_av}(a, b) :: \text{average}(x', y')$$

The algorithm is convergent because it need examine only one digit of each input stream to generate a digit of output. The performance of this algorithm is examined further in section 7.1.2.

4.2.2 Multiplication

Multiplying two streams of dyadic digits is also a relatively simple operation, Suppose we wish to multiply two streams of the form $(a :: x)$ and $(b :: y)$. Using the properties described in section 3.6.2 we can say

$$\begin{aligned}
(a :: x) \times (b :: y) &= (a \oplus x) \;\times\; (b \oplus y) \\
&= (a{\cdot}b \oplus a{\cdot}y) \;\oplus\; (b{\cdot}x \oplus x \times y) \\
&= (a{\cdot}b \oplus (x \times y)) \;\oplus\; (b{\cdot}x \oplus a{\cdot}y) \\
&= (a{\cdot}b :: x \times y) \;\oplus\; (b{\cdot}x \oplus a{\cdot}y)
\end{aligned}$$

From this we can now obtain a simple recursive algorithm:

$$\begin{aligned}
&\text{multiply}(a :: x, b :: y) = \text{average}(\text{p}, \text{q}) \\
&\qquad \text{where} \quad p = (\text{digit_mul}(a, b) :: \text{multiply}(x, y)) \\
&\qquad \phantom{\text{where}} \quad q = \text{average}(\text{digit_stream_mul}(b, x), \text{digit_stream_mul}(b, x))
\end{aligned}$$

This uses the operations to average two streams (section 4.2.1), multiply two digits (digit_mul, see section A.3), multiply a stream by a digit (digit_stream_mul, see sections 4.1.2 and A.3), and a recursive call to itself to multiply two streams.

Observe that the $(b{\cdot} x \oplus a \cdot y)$ term is directly computable from the input, we know one digit or $(a \cdot b :: x \times y)$ without needing to make a recursive call to this dyadic stream multiplication algorithm, and the dyadic stream average operation requires only one digit of each input to generate one digit of output. Hence an output digit is can be determined without making a recursive call and thus depends on a finite portion of the input. This argument also applies to the recursive call, and so it can be seen that this method is indeed convergent and we have an algorithm.

4.3 Signed Binary Stream Operations

We now develop algorithms for the signed binary stream operations for average and multiplication, and division of a signed binary stream by an integer. These are algorithms are used to implement the basic operations within the calculator, and are also required by other algorithms including division (section 4.4.4) and conversion to and from decimal (section 3.8).

The signed binary multiplication algorithm presented in this section is original.

4.3.1 Average

The algorithm for computing the average of two signed binary streams is more complicated than the one for dyadic streams described in section 4.2.1. This is because unlike dyadic digits. individual signed binary digits are not closed under average.

The algorithm described here may require two digits of the input streams to generate one digit of output. A 'carry' is used, but the carry is from left (most significant digit) to right (least significant digit), unlike conventional addition algorithms in which the carry occurs from right to left.

We describe the algorithm by first developing it from scratch and the proving its correctness. We then summarise the development by expressing the algorithm as a recursive function which computes the average of two inputs.

Suppose we wish to compute $(x \oplus y)$ where x is of the form $(a_0 :: a_1 :: x'')$ and y is of the form $(b_0 :: b_1 :: y'')$. We use signed binary carry digit c, which adds $c/2$ to the number represented by the result:

$$(a_0 :: a_1 :: x'') \oplus (b_0 :: b_1 :: y'') + \frac{c}{2}$$

Using the properties described in section 3.6.2, this is equal to:

$$\begin{aligned}
&\Big(a_0 \oplus (a_1 \oplus x'')\Big) \oplus \Big(b_0 \oplus (b_1 \oplus y'')\Big) + \frac{c}{2} \\
= \quad &(a_0 \oplus b_0) \oplus \Big((a_1 \oplus b_1) \oplus (x'' \oplus y'')\Big) + \frac{c}{2}
\end{aligned}$$

We now wish to generate one signed binary digit of output, and express the remainder of the number as a recursive call. Let e be the digit which is to be output, and the amount $c'/2$ be the amount which cannot be represented exactly in the digit e and is to be added to the remainder of the stream by passing the digit c' right as the new carry.

$$= e \oplus \left((a_1 \oplus b_1) \oplus (x'' \oplus y'') + \frac{c'}{2} \right)$$

These equations may be expanded to give:

$$\frac{c}{2} + \frac{a_0 + b_0}{4} + \frac{a_1 + b_1 + x'' + y''}{8} = \frac{e}{2} + \frac{c'}{4} + \frac{a_1 + b_1 + x'' + y''}{8}$$

By re-arranging this we get $c' = a_0 + b_0 + 2c - 2e$.

Now let us assume $-2 \leq (a_0 + b_0 + 2c) \leq 2$. We now show how to compute e and c' so that this assumption will hold when we make the recursive call to the stream average operation. Observe that as $c = 0$ initially, the assumption holds. Now find e, c' such that

$$\begin{aligned} & -2 \leq a_1 + b_1 + 2c' \leq 2 \\ \Rightarrow \quad & -2 \leq a_1 + b_1 + 2(a_0 + b_0 + 2c - 2e) \leq 2 \\ \Rightarrow \quad & -2 - 4e \leq a_1 + b_1 + 2(a_0 + b_0 + 2c) \leq 2 - 4e \end{aligned}$$

As $-2 \leq (a_0+b_0+2c) \leq 2$, we find e as follows. Let $d = 2(a_0+b_0+2c)+a_1+b_1$:

$$e = \begin{cases} \overline{1} & \text{if } (-6 \leq d < -2) \\ 0 & \text{if } (-2 \leq d \leq 2) \\ 1 & \text{if } (2 < d \leq 6) \end{cases}$$

These equations lead to an algorithm for average, although it is more complex than dyadic average. In certain situations, however, it is not necessary to examine two digits of each input stream as the average of the first input digits and carry digit may be expressed exactly as a signed binary digit with a carry of zero.

The algorithm developed and proved above amounts to the following recursive definition.

$$\text{average}(x, y) = \text{average}'(x, y, 0)$$

$$\text{average}'(a_0 :: x', b_0 :: y', \frac{c}{2}) = \begin{cases} \text{sign}(d') :: \text{average}'(x', y', 0) & \text{if } d' \text{ even} \\ \text{average}''(x', y', d') & \text{otherwise.} \end{cases}$$
$$\text{where} \quad d' = a_0 + b_0 + c$$

$$\text{average}''(a_1 :: x'', b_1 :: y'', d') = e :: \text{average}'(x'', y'', c')$$
$$\text{where} \quad d = 2d' + a_1 + b_1$$
$$e = \begin{cases} 1 & \text{if } (2 < d \leq 6) \\ 0 & \text{if } (-2 \leq d \leq 2) \\ -1 & \text{if } (-6 \leq d < -2) \end{cases}$$

The sign function is defined as follows:

$$\text{sign}(x) = \begin{cases} -1 & \text{if } (x < 0) \\ 0 & \text{if } (x = 0) \\ 1 & \text{if } (x > 0) \end{cases}$$

The implementation of this algorithm is described in section 6.3, and its behaviour is examined further in section 7.1.2.

4.3.2 Multiplication

The algorithm for signed binary stream multiplication is more complex than the dyadic algorithm described in section 4.2.2. We briefly explain why this is so, and show how we can obtain a suitable algorithm. We were unable to find an algorithm for multiplication using this representation in the literature, and so this algorithm is original work.

The algorithm for signed binary stream multiplication is more complex than the dyadic one. This is because the algorithm which averages two signed binary streams requires potentially two digits of each input stream to generate one digit of output, whereas the dyadic stream average algorithm only requires one. The effect of this is that if we attempted to use the same approach for signed binary stream multiplication as for dyadic stream multiplication, we would no longer have an algorithm because the function would be able to recursively call itself indefinitely without ever generating an output digit.

The solution to this problem is to develop an algorithm which uses a lookahead of two on the input streams. Suppose we wish to multiply two streams of the form $(a_0 :: a_1 :: x)$ and $(b_0 :: b_1 :: y)$. We can re-arrange this to show that $(a_0 :: a_1 :: x) \times (b_0 :: b_1 :: y)$ is in fact equal to:

$$\underbrace{((a_0 \cdot b_1 :: (b_1 \cdot x \oplus a_1 \cdot y)) \oplus (b_0 \cdot x \oplus a_0 \cdot y))}_{\mathbf{A}} \oplus \underbrace{(a_0 \cdot b_0 :: a_1 \cdot b_0 :: a_1 \cdot b_1 :: x \times y)}_{\mathbf{B}}$$

This may be easily verified by expanding the expressions into their semantic forms using the observations about the 'cons' operation and properties of average described in section 3.6.2

We can use this expression to obtain a recursive algorithm. This is because expression **A** is directly computable using the inputs, three digits of **B** are available directly without making a recursive call to multiply the streams x and y. We know that average of the streams obtained from terms **A** and **B** will require at most $(n+1)$ digits of each to determine n output digits (see sections 4.3.1 and 7.1.2), so in fact two digits may be determined before making a recursive call to the multiplication function.

$$
\begin{aligned}
&\text{multiply}(a_0 :: a_1 :: x,\ b_0 :: b_1 :: y) = \text{average}(p,\ q) \\
&\quad \text{where} \quad p = \text{average}(\ a_0 \cdot b_1 :: \text{average}(\ \text{digit\_stream\_mul}(b_1, x), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{digit\_stream\_mul}(a_1, y)), \\
&\qquad\qquad\qquad\qquad\qquad\quad \text{average}(\ \text{digit\_stream\_mul}(b_0, x), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{digit\_stream\_mul}(a_0, y))) \\
&\qquad\qquad\quad q = (a \cdot c :: b \cdot c :: b \cdot d :: \text{multiply}(x, y))
\end{aligned}
$$

An alternative formulation for which we could also give a recursive algorithm would be:

$$(a_0 :: a_1 :: x) \times (b_0 :: y) = (a_0 \cdot y \oplus (b_0 \cdot x \oplus a_1 \cdot y)) \ \oplus \ (a_0 \cdot b_0 :: a_1 \cdot b_0 :: x \times y)$$

Although superficially simpler, this second algorithm has worse lookahead properties. In fact a better lookahead for signed binary multiplication is achieved by multiplying signed binary digits to dyadics, as described in section 4.2.2, and then converting the resulting dyadic stream back to signed binary (see section 3.7), although we do not use this because it is affected by the problem of swelling dyadic digit representations. These issues are discussed in more detail in chapter 7.

4.3.3 Division by an integer

A third primitive operation on streams of signed binary digits is division of such a stream by an integer. Although it is not in general possible to divide a stream representing a number in the interval $[-1, 1]$ by another stream representing a number in the interval $[-1, 1]$ (as this interval is not closed under division), division by an integer value may be performed quite simply. An algorithm for division of arbitrary numbers in the signed binary (mantissa, exponent) representation is described in section 4.4.4.

The method used is similar to the school long division method, although both the input and output streams may contain negative digits. Digits are examined at the head of the numerator stream and evaluated as if they formed a signed binary integer. If this sum exceeds the denominator a one is output and the sum decremented by the denominator. Similarly if this sum becomes less than the negated denominator a minus one is output and the sum incremented by the denominator. If the absolute value of the sum is smaller than the denominator, we generate a zero digit.

A function int_div which divides a stream of the form $(a :: x)$ by the integer n may be defined as follows:

$$\text{int\_div}(a :: x, n) = \text{int\_div}'(a :: x, n, 0)$$

$$\text{int\_div}'(a :: x, n) = \begin{cases} 1 \;\; :: \text{int\_div}(x, n, s' - n) & \text{if } (s' \geq n) \\ 0 \;\; :: \text{int\_div}(x, n, s') & \text{if } (-n < s' < n) \\ \overline{1} \;\; :: \text{int\_div}(x, n, s' + n) & \text{if } (s' \leq -n) \end{cases}$$
$$\text{where } s' = 2s + a$$

4.4 (Mantissa, Exponent) Operations

In order to define operations on the whole real line, a (mantissa,exponent) representation is used. The mantissa is either a dyadic or signed binary stream, and the exponent is an integer of unrestricted size. Addition, subtraction, and multiplication can all be implemented using a stream operation and modifying the exponent separately. The full division algorithm presented in this section is more complicated.

4.4.1 Addition and Subtraction

Addition and subtraction of (mantissa, exponent) represented reals is performed using the average and negation operations on the mantissa streams as follows:

$$\begin{aligned} (m_a, e_a) + (m_b, e_b) &= (m_a \cdot 2^{(e_a - e_{max})} \oplus m_b \cdot 2^{(e_b - e_{max})}, e_{max} + 1) \\ (m_a, e_a) - (m_b, e_b) &= (m_a \cdot 2^{(e_a - e_{max})} \oplus (-m_b \cdot 2^{(e_b - e_{max})}), e_{max} + 1) \\ &\quad \text{where } e_{max} = \max(e_a, e_b) \end{aligned}$$

Notice that the powers of two are always zero or negative, and hence multiplication of a stream by 2^{-n} may be implemented using a simple shift right (prefixing n zeros to the stream).

4.4.2 Multiplication

Multiplication of (mantissa, exponent) represented reals is very simple indeed. One simply multiplies the mantissas and adds the exponents.

$$(m_a, e_a) \times (m_b, e_b) = (m_a \times m_b, e_a + e_b)$$

4.4.3 'Normalisation'

During the course of calculations using the (mantissa, exponent) representations, the exponents tend to become large and the mantissas small. This reduces the efficiency of computations using such representations significantly. The 'normalisation' processes described here is extremely useful for avoiding such problems.

The name 'normalisation' is used because this process is analogous to normalising a floating point number. This is slightly misleading, however, as a true normalisation process would normalise any representation of zero to a single unique representation. The process described here does not do that. As an aside, it would be possible to create an operation which modified each stream such that if it were zero the representation would be (say) an exponent of zero and a mantissa which was an infinite stream of the zero digits, but unfortunately we would not still not be able to test such a representation to determine if it actually represented the number zero (see section 2.2.2).

Signed binary (mantissa, exponent) 'normalisation'

Observe that using the signed binary (mantissa, exponent) representation:

$$[\![(0 :: x, n)]\!] = [\![(x, n-1)]\!]$$

Therefore if a number in (mantissa, exponent) representation is observed which starts with a zero (0) digit, it is possible to remove the zero and represent the original real by decrementing the exponent.

The 'normalisation' function examines the head of the mantissa stream and if the first digit is a zero then the real is re-represented as shown. The function then calls itself recursively so that it continues reducing the exponent until it can go no further.

There are two other identities which we can also use to make the algorithm more effective:

$$[\![(1 :: \overline{1} :: x, n)]\!] = [\![(1 :: x, n-1)]\!]$$

$$[\![(\overline{1} :: 1 :: x, n)]\!] = [\![(\overline{1} :: x, n-1)]\!]$$

Observe that typically such a function will not terminate if applied to a representation of zero. If the values being normalised could potentially represent zero, potential solutions include restricting the function to a maximum number of recursive calls, or only applying it until the exponent is less than or equal to zero.

Dyadic (mantissa, exponent) 'normalisation'

Dyadic (mantissa, exponent) 'normalisation' is more complex than the signed binary case. This is because the dyadic digits at the start of the stream can become very small indeed without ever being exactly zero. There are two obvious approaches:

The simplest method is to just remove dyadic zero digits from the head of the mantissa. It is possible, however, that a mantissa representing a very small value which could be 'normalised' is ignored because one of the early digits are small but non-zero.

The second approach is to add the first digit of the mantissa stream with half the second. If the absolute value of this sum is less than or equal to a half, the first two digits of the mantissa may be replaced by the digit representing twice this sum and the exponent decremented.

$$[\![(a :: b :: x, n)]\!] = [\![((2a+b) :: x, n-1)]\!] \qquad \text{if} \quad |2a+b| < 1$$

4.4.4 Division

We have already described an algorithm for division of a signed binary stream by an integer in section 4.3.3. Here we give an algorithm for division of two arbitrary reals. This algorithm is entirely my own work.

Suppose we wish to compute (a/b), where a and b are reals represented in signed binary (mantissa, exponent) representation by the numerals (x, e_x) and $(y', e_{y'})$ respectively. The approach is as follows. First we modify the mantissa y' of the numeral b to find a new numeral (y, e_y) which also represents b. We can perform division by this modified mantissa to obtain a new mantissa which represents $4x/y$ using the dyadic stream representation. Finally we convert this mantissa back into the signed binary representation, and use it and the exponents e_x and e_y to construct a numeral which represents a/b.

Note that division is undefined when the denominator is zero. It is impossible to test for equality using these representations (see section 2.2.2). Attempts to divide by zero will cause the algorithm to loop forever without returning digits.

Modification of mantissas We first need to modify the mantissas x and y' of the numerator and denominator in such a way as to allow us to perform division on the streams. Using the naïve approach of simply using the mantissas directly this would be impossible:

$$x \in [-1, 1] \text{ and } y' \in [-1, 1] - \{0\} \;\Rightarrow\; \frac{x}{y} \in (-\infty, \infty)$$

However we can use shift operations and the identities from section 3.3.1 to find an exponent $e \in \mathbb{Z}$ and a numeral y' such that:

$$[\![y]\!] = [\![y']\!] \times 2^e \qquad \text{such that } \; |y| \in \left[\frac{1}{4}, 1\right]$$

Essentially we insist that the stream y starts with a non-zero digit, and that the first two digits are not $(1 :: \overline{1})$ or $(\overline{1} :: 1)$. Now we can say:

$$x \in [-1, 1] \text{ and } |y| \in \left[\frac{1}{4}, 1\right] \Rightarrow \frac{x}{y} \in [-4, 4]$$

The algorithm will now compute $x/4y = (x/y') \times 2^{e-2}$. For simplicity we now assume that y always starts with the digit one. In order to compute division by a negative y, it is sufficient to negate y to make it positive, and then negate either the numerator x or the result of the whole division. Hence from now on:

$$y \in \left[\frac{1}{4}, 1\right]$$

We describe the algorithm and prove its correctness by examining cases. We look at all possible combinations of digits at the head of the input streams, and for each case show how we can find a dyadic digit which we can output, and express the remainder of the result as a recursive call to the division algorithm.

The algorithm is broken up into the cases which produce the digit zero as output, those which produce a positive digit, and those which produce a negative digit.

Output zero case: If the numerator input starts with a zero, or two digits which could be rewritten as a zero and another digit, the algorithm is simple.

$$\begin{aligned}
\text{divide}(0 :: x', \quad y) &= \ 0 :: \text{divide}(x',\ y) \\
\text{divide}(1 :: \overline{1} :: x',\ y) &= \ 0 :: \text{divide}(1 :: x',\ y) \\
\text{divide}(\overline{1} :: 1 :: x',\ y) &= \ 0 :: \text{divide}(\overline{1} :: x',\ y)
\end{aligned}$$

The proof is as follows. Here, one can say:

$$x \in \left[-\frac{1}{2}, \frac{1}{2}\right], \qquad y \in \left[\frac{1}{4}, 1\right]$$
$$\Rightarrow \frac{x}{4y} \in \left[-\frac{1}{2}, \frac{1}{2}\right]$$

Hence we can output a zero. Now to express the remainder of the result as a recursive call, we want to find a stream x'' such that

$$\frac{x}{4y} = 0 :: \text{divide}(x'',\ 4y) \qquad = 0 \oplus \frac{x''}{4y}$$

This is achieved by setting $(x'' = 2x)$. We can represent $2x$ using a signed binary stream as the initial two digits of x scanned indicate that it be expressed in the form $(0 :: x')$.

Output positive digits Positive digits are output if the numerator x is of the form $(1 :: 0 :: x')$ or $(1 :: 1 :: x')$. If so, we can immediately say

$$x \in \left[\frac{1}{4}, 1\right] \qquad \frac{x}{4y} \in \left[\frac{1}{16}, 1\right]$$

The obvious strategy here would be to generate the digit one (1) in the output stream. Unfortunately if we do so it apparently becomes very complicated to express the remainder of the result as a recursive call (doing so would probably require examination of more digits of the input streams x and y). An algorithm which outputs signed binary digits must exist as the representations used are all equivalent, but would probably be significantly more complex than the one presented here and require a larger lookahead.

The solution is to output a dyadic digit instead of a signed binary one, and to determine which digit by considering the result r of subtracting the denominator y from the numerator x. Let

$$r = y - x$$

Now, we examine the first two digits of r. There are three different possible cases to consider:

Case 1: The result r starts $(1 :: 1)$ or $(1 :: 0)$. Hence

$$r \in \left[\frac{1}{4}, 1\right]$$

We can deduce further that as x can be no greater than 1, and y is no less than $\frac{1}{4}$, r can be no greater than $\frac{3}{4}$. Similarly,

$$x \in \left[\frac{1}{2}, 1\right] \qquad y \in \left[\frac{1}{4}, \frac{3}{4}\right]$$

$$\Rightarrow \frac{x}{4y} \in \left[\frac{1}{6}, 1\right] \qquad \text{and} \qquad r \in \left[\frac{1}{4}, \frac{3}{4}\right]$$

Observe now that

$$r \in \Big[\frac{1}{4}, \frac{3}{4}\Big] \qquad \Rightarrow \qquad (r-y) \in \Big[-\frac{1}{2}, \frac{1}{2}\Big]$$

So we can represent the value $r' = 2(r-y) = 2(x-2y)$ in a stream. Now:

$$\begin{aligned}\frac{x}{4y} &= \frac{1}{2} + \Big(\frac{1}{2} \times \frac{2(x-2y)}{4y}\Big) \\ &= 1 \oplus \frac{r'}{4y} = 1 :: \text{divide}(r',\ y)\end{aligned}$$

Case 2: The result r starts $(\overline{1} :: 1)$, 0, or $(1 :: \overline{1})$. Therefore:

$$r \in \Big[-\frac{1}{2}, \frac{1}{2}\Big]$$

And we can represent $r' = 2r = 2(x-y)$ in a stream. Now:

$$\begin{aligned}\frac{x}{4y} &= \frac{1}{4} + \Big(\frac{1}{2} \times \frac{2(x-y)}{4y}\Big) \\ &= \frac{1}{2} \oplus \frac{r'}{4y} = \frac{1}{2} :: \text{divide}(r',\ y)\end{aligned}$$

Case 3: The result r starts $(\overline{1} :: \overline{1})$ or $(\overline{1} :: 0)$. Hence

$$r \in \Big[-1, -\frac{1}{4}\Big]$$

We can deduce further that as x can be no less than $\frac{1}{4}$, and y is no greater than 1, and therefore r can be no less than $-\frac{3}{4}$. Furthermore we can deduce:

$$\begin{gathered} x \in \Big[\frac{1}{4}, \frac{3}{4}\Big] \qquad y \in \Big[\frac{1}{2}, 1\Big] \\ \Rightarrow \frac{x}{4y} \in \Big[\frac{1}{16}, \frac{3}{8}\Big] \qquad \text{and} \qquad r \in \Big[-\frac{3}{4}, -\frac{1}{4}\Big]\end{gathered}$$

We cannot use r to express the result in terms of a digit and a recursive call directly, so we define r' as

$$r' = x - \frac{y}{2} = r + \frac{y}{2} \qquad \in \Big[-\frac{1}{2}, \frac{1}{2}\Big]$$

As $2r'$ is in $[-1, 1]$, we can represent $r'' = 2r' = 2x - y$ in a stream. Now, we can find a recursive call as follows:

$$\begin{aligned} \frac{x}{4y} &= \frac{1}{8} + \Big(\frac{1}{2} \times \frac{2x - y}{4y}\Big) \\ &= \frac{1}{4} \oplus \frac{r''}{4y} = \frac{1}{4} :: \mathrm{divide}(r'',\ y) \end{aligned}$$

Output negative digits Negative digits are generated in the output if the numerator starts $(\bar{1} :: \bar{1})$ or $(\bar{1} :: 0)$, which means that:

$$x \in \Big[-1, -\frac{1}{4}\Big]$$

The strategy here is identical to the one used when we output positive digits. We require three cases as before, and output either -1, $-\frac{1}{2}$, or $-\frac{1}{4}$. The derivations are not given here, but exactly the same method is used and the results are all of the same form with a few additions and subtractions swapped, and some minor sign changes present.

Extending division to the whole real line

Extending the division algorithm to work on the whole real line is relatively trivial. We now have an algorithm which will compute

$$\frac{x}{4y} \qquad \text{where } x \in [-1, 1], |y| \in \Big[\frac{1}{4}, 1\Big]$$

with the precondition that y is of the form $(1 :: 0 :: y')$ or $(1 :: 1 :: y')$. We can now implement the full division algorithm using the following observation:

$$\frac{(x, e)}{(y, f)} = \frac{x \times 2^e}{y \times 2^f} = \frac{x}{4y} \times 2^{2+e-f}$$

Appendix B gives the Haskell implementation of the entire division algorithm. The structure closely follows the approach taken in describing the algorithm here, and gives a succinct recursive definition for the algorithm.

5. High-Level Operations

In Chapter 4 we developed algorithms for the basic arithmetic operations. This chapter uses these operations as building blocks to implement higher level transcendental functions.

In order to compute the transcendental functions, we first define a function which converts a real represented as a sequence of nested intervals (as described in section 3.5) into a single signed binary numeral. This allows us to compute the limits of Cauchy sequences very simply, and hence many transcendental functions which can be defined as the limits of such sequences.

This general approach used to compute limits was originally proposed by Martín Escardó (personal communication). We develop this idea and give an algorithm and a proof of correctness and convergence here.

5.1 Computing Limits

In order to compute limits, we first define a function which takes three signed binary streams (x, y, z) as inputs with the property $[\![x]\!] \leq [\![y]\!] \leq [\![z]\!]$. The function examines the streams x and z and generates a (possibly empty) string of digits whose range (see section 3.1) contains all possible values for y and then uses the stream y directly to return the remainder of the output such that the whole output stream represents the number $[\![y]\!]$. Importantly, this function determines information about y from the bounds x and z before examining y itself.

In order to obtain an algorithm for computing limits, we require that this function has the property that if x and z are closer than some finite amount a digit is generated, and that if x and z are equal, y is not examined at all.

The function may be used recursively to convert a real represented as a stream of nested intervals (section 3.5) into a single signed binary stream by trying to find common digits from the first interval in the stream of nested intervals when possible, and then using the remaining intervals to determine the rest of the signed binary stream. When we have determined digits from the first interval (or determined that they end-points of the stream are not equal), we examine the next intervals. We can use the 'p' function described in section 4.1.4 to re-express the endpoints of the next interval starting with the digits already generated, and then remove these known digits from the endpoints. This allows us to then use the original function again to determine the further digits in the output stream.

We know that because the nested intervals converge to a single value, we can get arbitrarily many output digits by examining a finite number of intervals.

The motivation for defining such a function is that now, in order to compute the limit of a given Cauchy sequence, we can express the sequence as a stream of nested lower and upper bounds on the limit and then simply convert this into a signed binary stream as described. Many useful functions can be defined as the limits of Cauchy sequences.

5.2 Limits within [-1,1]

Determining output digits

Suppose we have inputs (x, y, z) where

$$[\![x]\!] \leq [\![y]\!] \leq [\![z]\!]$$

We examine the first digits of x and z, and use simple analysis of the range of possible reals represented, as described in section 3.6.2 to determine whether or not we can determine digits of a stream representing $[\![y]\!]$ using only the known digits of x and z.

Having examined a some finite number of digits from x and z, we will encounter on of three qualitatively distinct cases; either we can say that $[\![x]\!] \neq [\![z]\!]$, or we can find a digit or digits whose range encloses all possible values of $[\![y]\!]$, or we cannot find such a digit but cannot say that $[\![x]\!] \neq [\![z]\!]$. In the first case it is safe to cease examination of x and z and use y to return the remainder of the result. In the second and third cases we must continue to examine x and z, generating digits where possible, until we can say for certain that $[\![x]\!] \neq [\![z]\!]$.

Suppose the ranges $r = [r_{min}, r_{max}]$ and $s = [s_{min}, s_{max}]$ contain the lower and upper bounds x and z of y. There are a number of possible situations that may arise. Figure 5.1 illustrates with some examples.

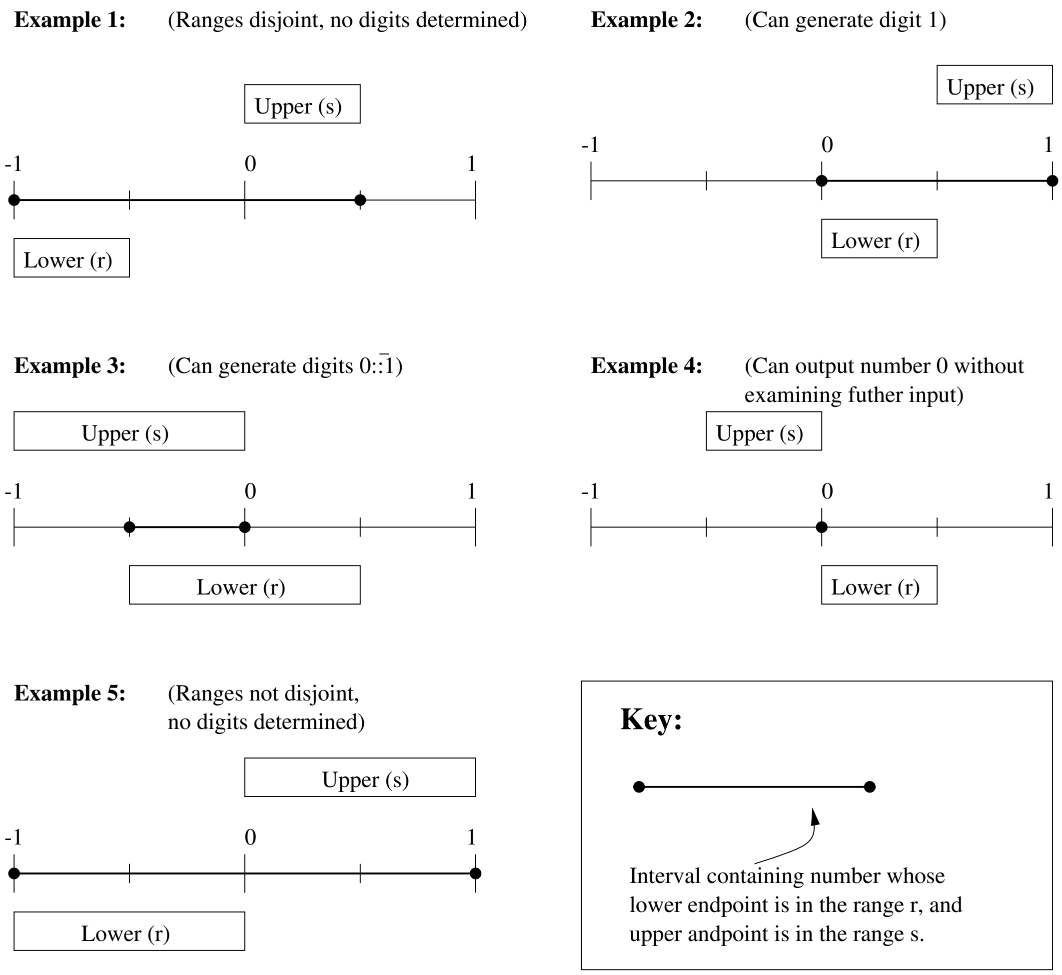

Figure 5.1: Limit cases

Case 1: The lower and upper bounds are too far apart to determine a digit, and ranges are disjoint. Therefore lower and upper bounds not equal and we can safely return y.

Example 1 illustrates such a situation.

Case 2: We can find a digit such whose range encloses $[r_{min}, s_{max}]$. Output this digit and examine the inputs further.

In example 2, we can deduce that the range of y is $[0, 1]$, so we can generate the digit 1 as the first digit of y. Similarly in example 3, the range of y must be $[-\frac{1}{2}, 0]$, so we can generate the digits $(0 :: \overline{1})$ (or equivalently $(\overline{1} :: 1)$).

Example 4 is an interesting possibility which would be handled by here. We can see that the least possible value of the lower bound r_{min} is equal to the greatest possible value of the upper bound s_{max}. We can deduce that this is the only possible value for y and output it directly without examining further inputs or intervals.

Case 3: We cannot find a digit such whose range encloses $[r_{min}, s_{max}]$, but we cannot say for certain that the lower and upper end-points are not equal. Therefore we must examine more digits of the end-points.

Example 5 falls into this category. Suppose the numeral $x = (\overline{1} :: \overrightarrow{1})$ and $z = (1 :: \overrightarrow{-1})$, we can see that $[\![x]\!] = [\![z]\!] = 0$. If this were the case, we would have to output the number zero (eg. the numeral $\overrightarrow{0}$) without examining y or any further intervals by the condition stated in section 5.1. We must therefore examine further digits of x and z until either we can ascertain that they are both equal, or we can generate a digit as in examples 2, 3, or 4.

Input digits required

We now show that we need to examine at most three input digits from x and z in order to either determine at least one additional output digit, or to determine that the lower and upper bounds are definitely not equal. In doing so we satisfy the conditions given in section 5.1.

After examining three digits of the input, we know that the range of both the lower and upper bounds is of width $\frac{1}{4}$. If the lower and upper bounds are not disjoint, the width of the range of possible numbers which fall between the least possible lower bound and the greatest possible upper bound is not greater than $\frac{1}{2}$.

Given any interval whose width is less than or equal to $\frac{1}{2}$, we can find a single digit whose range encloses the interval. We can also say that if there exists no signed binary digit containing both intervals of width $\frac{1}{4}$, then the intervals are definitely disjoint.

A recursive definition

We now show how one might use the approach described above to give a recursive definition for this algorithm. The approach is as follows; we use an auxiliary function f' to do the work hard work, and call it with the correct parameters by the function f, which is the function we ultimately want to use. The function f' takes a parameter sig which determines the number of digits examined since the last digit was decided (or rather the amount by which the further digits examined can modify the range of the finite number of digits examined at the head of x and z so far). Two parameters r and s determine the range of the lower and upper bounds given the first few digits of x and z seen so far, all multiplied by eight. Parameters out_x and out_z are used to store the digits already seen in streams x and z.

$$f(x, y, z) = f'(x, y, z, sig, (-8, 8), (-8, 8), [\,], [\,])$$

$$\begin{array}{l}
f'(a :: x', y, c :: z', sig, (r_{min}, r_{max}), (s_{min}, s_{max}), out_x, out_z) \\
\quad = \left\{ \begin{array}{lll}
1 :: f(\ \mathrm{p}(1, out_x +\!\!+ x),\ \mathrm{p}(1, y),\ \mathrm{p}(1, out_z +\!\!+ z)) & \text{if} & (r'_{min} \geq 0) \\
0 :: f(\ \mathrm{p}(0, out_x +\!\!+ x),\ \mathrm{p}(0, y),\ \mathrm{p}(0, out_z +\!\!+ z)) & \text{if} & (r'_{min} \geq -4) \\
 & \text{and} & (s_{max} \leq 4) \\
\overline{1} :: f(\ \mathrm{p}(\overline{1}, out_x +\!\!+ x),\ \mathrm{p}(\overline{1}, y),\ \mathrm{p}(\overline{1}, out_z +\!\!+ z)) & \text{if} & (r'_{min} \leq 0) \\
y & \text{if} & (r'_{max} < s'_{min}) \\
\multicolumn{3}{l}{f'(x', y, z', sig \text{ div } 2, (r'_{min}, r'_{max}), (s'_{min}, s'_{max}), out_x +\!\!+ [a], out_z +\!\!+ [c])} \\
 & \multicolumn{2}{l}{\text{otherwise}}
\end{array} \right. \\
\quad \text{where} \quad (r'_{min}, r'_{max}) = (r_{min} + (a+1) \cdot sig, r_{max} + (a-1) \cdot sig) \\
\qquad\qquad\quad\ (s'_{min}, s'_{max}) = (s_{min} + (c+1) \cdot sig, s_{max} + (c-1) \cdot sig)
\end{array}$$

In this definition, the function p is the 'p' function defined in section 4.1.4, and a double plus (++) is used to denote concatentation of two lists.

The implementation of this function in Haskell is given in appendix B.

Recursive calls to compute the limits

Suppose we have a stream of nested intervals represented as $(x_1, z_1) :: (x_2, z_2) :: \ldots$, whose end points are represented as signed binary streams (section 3.5), and with the following property:

$$[x_1, z_1] \supseteq [x_2, z_2] \supseteq [x_3, z_3] \supseteq \ldots$$

We also have a function f which takes three signed binary stream x, y, and z as inputs, and outputs a signed binary stream representing $[\![y]\!]$ by examining x and z, determining common digits from them if it can using the method described above, and then using y directly to determine the rest of the output.

We can now compute the signed binary stream represented by the stream of nested intervals y as the result of $\text{limit}(y)$, where the function limit function is defined as:

$$\text{limit}((x, z) :: y) = f(x, \text{limit}(y), z)$$

5.3 Limits within $[-\infty, \infty]$

Section 5.2 shows how we can convert a stream of nested intervals in the range $[-1, 1]$ into a signed binary stream. It would be more useful to be able to convert streams of nested intervals from the whole real line.

The problem here is that using the (mantissa, exponent) representation for end-points, the numeral for each end-point can have different a exponent and we cannot use the mantissa's directly as we could in section 5.2. Fortunately, we can easily choose an exponent with which can represent the final output, and all other end-points in the stream of nested intervals, by simply taking the maximum exponent required at the first interval of the stream. Because the intervals are strictly nested, we can then re-represent all subsequent end-points using this exponent and operate solely on the mantissas using the previous method, and outputting a result using the chosen exponent.

5.4 Transcendental Functions

Computing transcendental functions is performed as follows. Suppose we wish to compute the value $f(x)$, we find a sequence or series in terms of x which either which tends towards a limit $f(x)$ with a known rate of convergence, or converges to but oscillates round the limit $f(x)$. Once we have this we can generate a sequence of upper and lower bounds on this limit at each term of the original sequence. We know that the sequence converges, so we can use this fact to generate an infinite and strictly nested stream of intervals containing the limit of the sequence. Once we have this stream, we can then converted it into a signed binary representation of the desired result using the method described in section 5.1.

We now give some sequences satisfying these conditions for a number of trigonometric and logarithmic transcendental functions.

5.4.1 Trigonometric Functions

We discuss the trigonometric functions sine, cosine, arctan, and the computation of the constant π.

Sine

The sine function is relatively simple to implement using the following approximation:

$$\sin(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} \cdot x^{2n-1}}{(2n-1)!}$$

Using this sum, we can easily construct a sequence $S(x)$ such that $S_n(x) \rightarrow \sin(x)$ as $x \rightarrow \infty$. We by also computing a second sequence $s(x)$, the current value of which can be computed as we generate the sequence when implementing this:

$$
\begin{array}{ll}
s_0(x) = x & s_n(x) = \dfrac{(-1)^n \cdot x^2 \cdot s_{n-1}(x)}{2n \cdot (2n-1)} \\
S_0(x) = s_0 & S_n(x) = S_{n-1}(x) + s_n(x)
\end{array}
$$

Observe that if $x = 0$, all terms of the sequence are zero. If $x > 0$, all even terms are greater than $\sin(x)$, and all odd terms less than it, and if $x < 0$, all even terms are less than $\sin(x)$, and all odd terms greater than it.

Using this information, it is extremely easy to generate a stream of nested intervals using two consecutive terms of the sequence of lower and upper bounds. Notice that the division is always by in integer. The integer division algorithm is simpler and quicker than algorithm for division of a real number by another real.

Cosine

The cosine function is implemented in exactly the same manner as the sine function using the approximation:

$$\cos(x) = \sum_{i=1}^{\infty} \frac{(-1)^n \cdot x^{2n-1}}{(2n)!}$$

Arctan

Similarly, the arctan function can be easily computed using:

$$\arctan(x) = \sum_{i=0}^{\infty} \frac{(-1)^n \cdot x^{2n+1}}{2n+1}$$

The convergence of this sequence is extremely slow as $|x|$ approaches one. This has particular relevance when we compute π.

Pi

The computation of π has a long and colourful history (at least by mathematical standards), and has been much studied. The method chosen to implement an algorithm here is to use some identities due to the the mathematician John Machin in 1706. A fuller discussion of approaches to computing π is available in [2].

Perhaps the simplest identity (by James Gregory and Gottfried Wilhelm Leibniz) is:

$$\frac{\pi}{4} = \arctan(1)$$

However this identity is impractical because the sequence for arctan converges far too slowly with $x = 1$ to be of value. Over 300 terms of the expansion of arctan given in section 5.4.1 would be required for two decimal places, and to achieve 100 correct digits would require a massive 10^{50} terms.

Machin's variation uses the tan double angle angle formula

$$\tan(2\theta) = \frac{2\tan(\theta)}{1 - \tan^2(\theta)}$$

to derive

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

There are other similar identities. The one used in this implementation is:

$$\pi = 24\arctan\left(\frac{1}{8}\right) + 8\arctan\left(\frac{1}{57}\right) + 4\arctan\left(\frac{1}{239}\right)$$

These identities give a much faster of convergence because the sequence which approximates arctan is generated with inputs much closer to zero.

These identities are not the most effective method of computing π. In fact more recent approaches allow the computation of an arbitrary hexadecimal digit of π without any computation of the preceding digits.

5.4.2 Logarithmic Functions

We discuss the exponential function, natural logarithm, and computing exponentiation (ie. x^y).

Exponential Function

The exponential function is usually defined as follows:

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

We can define a sequence $S(x)$ such that $S_n(x) \to \exp(x)$ as $n \to \infty$ as follows, using a second sequence $s(x)$, whose current value is used to simplify the implementation:

$$\begin{aligned} s_0(x) &= 1 & s_n(x) &= \frac{x.s_{n-1}(x)}{n} \\ S_0(x) &= s_0(x) & S_n(x) &= S_{n-1} + s_n(x) \end{aligned}$$

If $(x < 0)$, the sequence S oscillates about the limit. We can easily use two consecutive elements of the sequence to compute a lower and upper bound, and hence compute the required stream of nested intervals.

If $(x > 0)$, the sequence no longer oscillates around the sequence, but tends to the limit from below. We can observe that

$$\left|\frac{x^n}{n!}\right| \leq 1 \quad \Rightarrow \quad S_n(x) \leq \exp(x) \leq S_n(x) + \frac{x^n}{n!}$$

Therefore to create a sequence of nested intervals bounding the required limit, we simply find N such that

$$n > N \Rightarrow \left|\frac{x^n}{n!}\right| \leq 1$$

Now we simply compute nested intervals using the bounds above.

There is a problem with the above approach, which is that we cannot determine the sign of x without examining a potentially infinite number of digits, and yet we need to know the sign to determine the method to use. One solution to this is to 'normalise' the input, but stopping after a finite number of steps if no non-zero digit is found. If we cannot determine the sign, we simply compute S_{n-1}, S_n, and $S_n(x) + \frac{x^n}{n!}$, and then take the minimum and maximum of these numbers as lower and upper bounds respectively. This is the way the algorithm has been implemented in the calculator, and despite the extra cost, the method is very fast as both sequences converge extremely quickly for small values of $|x|$.

Natural Logarithm Function

Computing the natural logarithm function is performed using the approximation

$$\ln(1+x) = \sum_{i=0}^{\infty} \frac{x^i}{i} \qquad (-1 < x \leq 1)$$

Using this we can construct a sequence $S(x)$ with the property that $S_n(x) \to \ln(1+x)$ as $x \to \infty$ as follows:

$$S_0(x) = x \qquad S_n(x) = S_{n-1} + \frac{x^n}{n}$$

We can prove that if $x \in [-\frac{1}{2}, \frac{1}{2}]$, this converges to the limit with a rate of convergence which ensures $S_n(x) - 2^{-n} < \ln(1+x) < S_n(x) + 2^{-n}$. In order to get an upper and lower bound on the value of $\ln(1+x)$, we need only take the first n digits of the (corrected to take account of the exponent), and 'cons' this list onto an infinite sequence of $\overline{1}$'s for the lower bound or of 1's for the upper bound.

This allows us to compute $\ln(1+x)$ given that $x \in [\frac{1}{2}, \frac{3}{2}]$. More generally, however, we would like to be able to compute $\ln(x)$ for all values of $(x > 0)$. In order to do this, we use the following properties of natural logarithm:

$$\begin{aligned} \ln(x \cdot e) &= \ln(x) + 1 \\ \ln\left(\frac{x}{e}\right) &= \ln(x) - 1 \end{aligned}$$

To calculate the natural logarithm of a value x, we multiply or divide x by the constant e to obtain x' which in the range $[\frac{1}{2}, \frac{3}{2}]$. We can then apply the algorithm to compute $\ln(1 + x')$, and use this result to work $\ln(x)$. The complicated part is that we do not have relational operators which allow us to test whether x' is in the required range (see section 2.2.2). To solve this problem we use a subtle trick which allows us to implement the algorithm using these properties.

Although we cannot actually test whether a given stream x is in the specified range, we can examine a finite portion of it and narrow the range reals it could represent until we can deduce one of the following facts:

$$x \in \left[\frac{1}{2}, \frac{3}{2}\right] \qquad x.e \in \left[\frac{1}{2}, \frac{3}{2}\right] \qquad \frac{x}{e} \in \left[\frac{1}{2}, \frac{3}{2}\right]$$

We then repeatedly multiply or divide x by e until we can observe one of these facts and proceed as before. In fact we need to examine at most five signed binary digits of x at each stage to determine whether we need to multiply or divide again or whether we can proceed to the next stage of the calculation.

At this point we can compute the limit of the sequence defined above generated by the new x after multiplication or division by e as this new x is in the required range, and use this to obtain the desired result. It is necessary to examine sufficient digits and cases to ensure that it is not possible to 'miss' the desired range by perpetually multiplying and dividing x by e without ever being able to definitely say that it lies in the desired range.

Exponentiation: x^y

A function to compute arbitrary powers of x can be implemented using the exponential and natural logarithm using the following identity:

$$x^y = \exp(y \cdot \ln(x))$$

5.5 Functional Operations

Algorithms for Alex Simpson's functional operations integration and function minimum and maximum are also implemented in the calculator. These algorithms are interesting, not least of all because it is quite surprising that they can be developed at all

A discussion of these algorithms is given in [26]. We do not undertake a full discussion of these algorithms here for two reasons. Firstly, their explanation is extremely involved; the paper cited is roughly ten pages long and even this is only an extended abstract. Secondly, although time was spent working on the implementation of these algorithms and linking this to the user interface of the calculator, the algorithms were implemented to see the algorithms working in a real implementation and to test their performance. The implementation of these algorithms here does not reflect any intellectual effort spent developing the algorithms.

The functional operations themselves take as input some closed interval, and a function which is continuous over the interval. The function maximum and minimum functions return the greatest or least value attained by the input function over the interval, and the integration algorithm computes the definite integral of the function in the specified interval.

The algorithms are defined first using the stream representations of reals in a closed interval. They take input in the signed binary stream representation, make extensive use of the dyadic representation for computation, and then give an output in the dyadic stream representation which can be coerced back into the signed binary stream representation.

The algorithms are then extended to functions which operate on the whole real line by scaling the closed interval over which the algorithm operates onto the interval $[0, 1]$, computing the largest exponent achieved by the function over the closed interval, and then scaling the output appropriately.

The implementation of the functional algorithms adapts directly an implementation by Reinhold Heckmann of the algorithms in the closed interval. This adaption means that they may be applied to functions using the full range of arithmetic operations and transcendental functions. We then extend the algorithms using the approach described above to operate on the whole real line as described. This extension is the first implementation of these algorithms on the whole real line.

6. Design and Implementation

In the previous chapters we developed algorithms for the basic arithmetic operations and transcendental functions. In this chapter we now discuss the design and implementation of a calculator which uses them. We describe the design structure and the user interface, the choice of the implementation language Haskell, and give a brief coverage of some general issues that arise during implementation. We also show how we convert the algorithms described in chapters 4 and 5 into Haskell using the signed binary stream average and division by an integer algorithms as examples.

6.1 Design

The design attempts to separate the functionality of the calculator from the interface and to construct the calculator in a modular fashion. In so doing we simplify the use of the core code outside the user interface if necessary. In addition, it allows us to modify modules internally without affecting others; for example the internal representation and primitives for dyadic digits could be modified—and indeed were—without affecting any of the higher level algorithms using them.

6.1.1 Modular Structure

Figure 6.1 shows the conceptual design. At the lowest level the basic representations and operations on signed binary and dyadic streams and their (mantissa, exponent) extensions are implemented. These are used to implement conversions between representations, algorithms which use both representations, limits of sequences and transcendental functions, functional operations, and conversions between signed binary and decimal. At the top level, the calculator requires a user interface to obtain and parse input. The other modules provide the functionality to compute the result of the expression and generate an output string which can be returned to the user through the user interface.

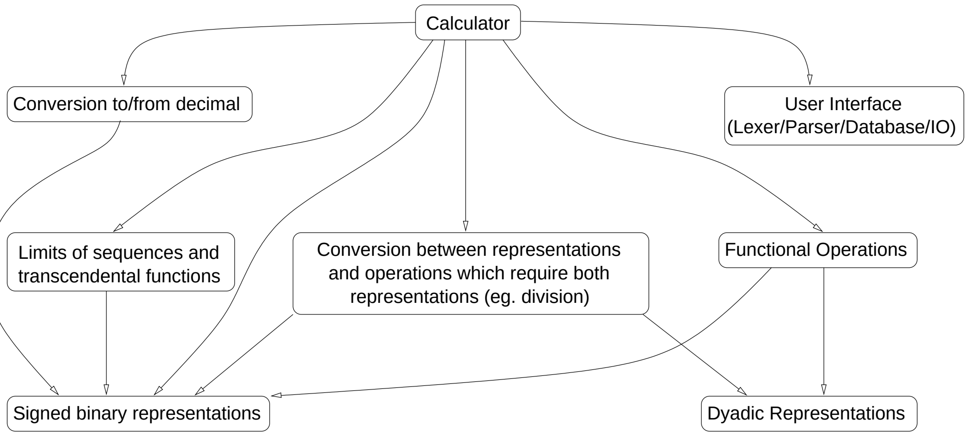

Figure 6.1: Conceptual structure of the design

6.1.2 User Interface

The user interface is fairly simple. Its purpose is to provide a vehicle for demonstrating the functionality of the calculator without requiring a direct knowledge of the implementation.

The interface itself consists of text prompt at which the user enters expressions using a straightforward expression language. The calculator computes the result and presents the user with a signed decimal stream as output as the computation proceeds and a decimal output when the result has been computed to a user specified output precision.

A version of the calculator which includes the functional operations but only gives output in signed binary form has also been created. This is because in general the functional operations are extremely slow to compute. The signed binary output is more useful because individual decimal digits require four signed binary digits to generate and does not show range of possible values represented by the computed portion of the stream as precisely.

The expression language itself includes decimal numbers, all the basic arithmetic and transcendental functions, the constant π, and simple unscoped variables which may be assigned to or evaluated. A graphical representation of the structure of the expression language and a sample session are given in appendix C.

6.2 Implementation

The chosen implementation language was the functional programming language "Haskell". We briefly justify the use of a functional language instead of an imperative one and the choice of the language Haskell. We talk about the approach used to implement the user interface, the tools used are described, and give a brief account of the stages the implementation went through during its evolution. Lastly a few issues which arise during implementation are discussed.

6.2.1 Language

It was decided to implement the calculator using a functional programming language, although it would been equally possible to implement it using an imperative one. Functional and imperative programming languages each have their merits and disadvantages. As a general rule one would expect an imperative implementation to be faster and more memory efficient, but a functional one would probably be simpler to program, modify, and experiment with. It is also likely that a functional implementation would have less source code and be less likely to contain bugs.

Imperative languages usually give the programmer almost direct control over the machine's representation of data and low level control over the instructions executed. As such it is possible to choose highly efficient representations and optimise the code which is executed frequently for performance. On the other hand the programmer is exposed to such issues and changing early decisions later on in the development can be complicated and leave the program unstable if not performed with extreme care.

Functional programming languages typically offer a higher level of abstraction, so the programmer is one step further removed from the actual representations and execution. As a result a functional language compiler or interpreter will often need to add its own memory and processing overheads which affect performance.

In return for this, however, the programmer does not need to worry about issues like memory management, garbage collection, data structures, execution order, and numerous other such tasks which make the code simpler, easier to develop, and less prone to bugs.

The implementation of this calculator is meant as a tool to help develop, experiment with, and demonstrate the representations and algorithms we have described. Using a functional programming language makes these tasks simpler, and as performance is not a priority it was decided that this would be preferable to implement using an imperative one.

Had we chosen to implement the calculator using an imperative programming language, one data structure which it would have been necessary to implement would have been the stream. This would be a complex task, particularly if we required efficient memory management and high performance. A functional language which incorporates lazy lists already addresses these problems, thus sparing us this effort. In addition, a good compiler for a functional programming language will probably use sophisticated techniques to manage these data structures and would probably perform better than an attempt programmed from scratch in an imperative language without some fair degree of thought.

The choice of a functional language was justified during the course of the implementation and analysis, as some quite fundamental changes were made to the implementation which were facilitated by this choice. For example, in chapter 7.2 were refer to an implementation of the algorithms which keeps track of the lookahead in the input streams required. Using Haskell, it was possible to re-implement the stream representations such each digit also held lookahead information and operations on digits tracked the maximum lookahead, and then use these streams in the algorithms already implemented, without significant change to the existing code. Although it would have been possible to make the same changes in an imperative language, they would have required either significant modification of the code, or a quite remarkable degree of foresight in the early stages design and coding to make the code sufficiently modular and flexible that the changes could have been made easily.

Hughes [16] gives an interesting discussion of these and other benefits of functional programming.

Haskell: An appropriate choice of functional language is also important. Haskell, the language chosen, is a pure, non-strict functional language[22]. Most importantly Haskell incorporates lazy evaluation and lazy lists, which simplify operations on streams by allowing them to be manipulated as lists (see section 3.2). Operations such as pattern matching at the head of a stream or generating streams are possible in a very natural fashion, simplifying the code and making it more readable. In addition, Haskell avoids re-evaluating expressions which have already been evaluated, which undoubtedly contributes to the performance of the system.

Haskell also incorporates arbitrary length integers, modules, and has a good set of I/O libraries. There are a number of implementations of Haskell available. The ones used are described in section 6.2.3.

6.2.2 User Interface

The user interface is implemented entirely in Haskell. Although a functional language is perhaps not the most obvious choice for implementing the user interface, techniques and tools exist which make the task not only possible, but even relatively straightforward. The advantage of implementing entirely in one language is that we do not need to worry about interfacing with another. This would be generally complicated and make testing with the user interface harder, especially because it would prevent us using a Haskell interpreter. As we only require a simple user interface for demonstration purposes, using another language to implement it is an entirely unnecessary complication.

The user interface operates by using Haskell I/O facilities to obtain a string. A lexer, which is automatically generated using the tool "Alex" (see section 6.2.3), is used to convert the string into a list of tokens. Next a parser, which is automatically generated using the tool "Happy" (see section 6.2.3) converts the list of tokens into an expression tree. Because Haskell is a pure functional language, we require the use of *Monads* [32] to perform error checking and to simulate exceptions.

The calculator itself takes the expression tree and uses this to generate a function which will return the desired result in signed binary (mantissa, exponent). Conversions from input strings of decimal numbers are performed to generate streams representing constants in the expression, and a simple database is accessed to obtain expressions which are substituted for the variables in the expression. Finally at the top level this function is evaluated to an appropriate precision, its result converted into signed decimal and decimal to an appropriate precision, and the output from this displayed to the user.

In some cases, such as with division or exponentiation by an integer, there are two possible algorithms that could be used. Either we treat the integer constant as an integer and use the simpler algorithm, or compute it using the full division algorithm. In this implementation we have adopted the more general (but slower) approach. Performance of the calculator as accessed through the user interface could be improved my making the parser slightly more sophisticated and making these and similar optimisations.

6.2.3 Implementation Tools

The following tools were used to implement the calculator

Gofer: Gofer is a small, freely available interpreted language which is closely related to Haskell 1.1, written by Mark P. Jones.

HUGS: HUGS is the Nottingham and Yale Haskell User's (Gofer) System. It is an interpreted version of Haskell 1.4, the successor of Gofer, and freely available for personal or educational purposes from `http://www.haskell.org/hugs`.

Glasgow Haskell Compiler: The Glasgow Haskell Compiler (GHC) is a compiler for Haskell 1.4. Available with the Glasgow Haskell Compiler is a profiling system and the parsing tool "Happy". The Glasgow Haskell Compiler is freely available from `http://www.dcs.gla.ac.uk/fp/software/ghc/`. The binaries produced by the GHC are significantly faster than the same code executing within HUGS, although the compilation process can be extremely time consuming.

Happy: Happy is a parser generator similar to the unix Tool `yacc`. It automatically generates code to parse a specified language.

Alex: Alex is a `lex` like package for generating Haskell scanners. It is available from `http://www.cs.ucc.ie/~dornan/alex.html`, and its release is covered by a GNU public licence.

6.2.4 Implementation History

Initial experimentation was performed using the language ML. It was found, however, that it was complicated to manipulate streams having to explicitly evaluate digits when they were required using the datatype described in section 3.2.

The language Gofer was then chosen. The early ML code was converted into Gofer and much of the early implementation work was performed using it. However Gofer lacks features such as arbitrary length integers and a module system, and later on HUGS (Haskell 1.4) was used instead. The Gofer code executable within HUGS with a minimum of alteration.

During the later stages of the project, the full Glasgow Haskell Compiler for Haskell 1.4 was obtained. This allowed tools such as "Alex" and "Happy" to be compiled, the creation of binaries giving improved performance, and profiling of the implementation to be performed.

Although the performance of the compiled code was significantly better than the interpreted code, the compilation process was time-consuming and the resulting binaries large. The interpreter was much better for development, debugging, and experimentation purposes.

6.2.5 General Issues

There are a couple of general issues which arose during implementation not relevant to the algorithms themselves and independent of any chosen implementation language.

Use of fixed and arbitrary length integers There are many functions which appear in the implementation which take integers as arguments and where the integer may in theory be arbitrarily large. If, for example, we need a function which prints out a specified number of digits, the integer specifying the number of digits might be arbitrarily large. For the purposes of implementation, however, there are a large number of cases when a fixed length (eg. word length, 32-bit) integer will suffice. For example it is highly unlikely that the user would ever require more than say $2147483648 (= 2^{31})$ digits, and if they did the time and space requirements would be so enormous that the program would never be able to supply them anyway.

The advantage of avoiding arbitrary precision integers where possible is that the code becomes slightly more efficient. The decision as to whether to use a 32-bit integer instead of an arbitrary length one must be made on a function by function basis and is not always an acceptable optimisation.

Dyadic digit representation There are two obvious representations for the dyadic digits described in section 3.3.2.

1. As a string of bits representing a binary fraction. This is similar to the representation of an integer, except that the integer is aligned round the least significant digit but a binary fraction around its most significant digit.

2. As a pair of arbitrary length integers

The choice of representation is arbitrary in the sense that once the primitive operations are defined the choice should the dyadic stream or other algorithms. It may be preferable, however, to use one representation over another.

The first representation has the advantage that it would theoretically require less space if implemented efficiently. Unlike the second representation, there is also no problem with keeping the first representation in its lowest terms as there is only one possible representation of each dyadic rational using the first representation (excluding trailing zeros), but infinitely many for each using the second. We overcome this in the second representation by ensuring that the denominator is always positive and the numerator always odd or zero.

Although the first representation would in theory be more efficient, the second representation was chosen to implement the algorithms. There are two good reasons for this. Firstly, arbitrary length integers are supplied for free in Haskell, making it extremely easy to implement operations using the second representation. In order to use a string of bits to represent dyadic digits, one would have to implement the arithmetic operations from scratch. This is both difficult in a functional language, and less efficient in real terms because it is no longer possible to take advantage of the optimised internal operations on arbitrary length integers. Secondly, it is easier to design, describe, and understand algorithms written using the second representation than using the first.

In actual fact as early implementations of the calculator were written in Gofer (see section 6.2.3), in which arbitrary length integers were not available, and it was therefore necessary to write primitive dyadic digit operations from scratch using the first representation. The algorithms required are modifications of those used for arbitrary length integer arithmetic which are described in detail in [17]. These were subsequently replaced by versions using the arbitrary length integer package later in the development.

6.3 Coding an algorithm in Haskell

Chapter 4 describes the algorithms for the basic operations. Once we have developed and these algorithms, proving correctness and convergence, coding them in Haskell is a relatively straightforward procedure. The descriptions of the algorithms suggest obvious recursive definitions which are easily expressed in the functional language.

In order to illustrate the procedure we examine the implementation of two algorithms for performing division of a signed binary stream by an integer (section 4.3.3), and averaging two signed binary streams (section 4.3.1). The Haskell code used to implement the algorithms for division and the computation of limits of sequences described in sections 4.4.4 and 5.1 is given in appendix B.

6.3.1 Signed Binary Division by an Integer

In section 4.3.3, the algorithm for the division of a signed binary stream by an integer was defined as follows:

$$\text{int\_div}(a :: x, n) \;=\; \text{int\_div}'(a :: x, n, 0)$$

$$\text{int\_div}'(a :: x, n) \;=\; \begin{cases} 1 \;\; :: \text{int\_div}(x, n, s' - n) & \text{if } \; (s' \geq n) \\ 0 \;\; :: \text{int\_div}(x, n, s') & \text{if } \; (-n < s' < n) \\ \overline{1} \;\; :: \text{int\_div}(x, n, s' + n) & \text{if } \; (s' \leq -n) \end{cases}$$

where $s' = 2s + a$

The Haskell implementation of this algorithm is as follows:

```
-- sbIntDiv : Division of a signed binary stream by an integer
sbIntDiv :: SBinStream -> Int -> SBinStream
sbIntDiv _ 0    = undefined
sbIntDiv x 1    = x
sbIntDiv x (-1) = sbNegate x
sbIntDiv x n    = sbIntDiv' x n 0

-- sbIntDiv' : Auxiliary to sbIntDiv
sbIntDiv' :: SBinStream -> Int -> Int -> SBinStream
sbIntDiv' (a:x) n s = if (s' >= n)  then (  1:sbIntDiv' x n (s'-n)) else
                      if (s' <= -n) then ( -1:sbIntDiv' x n (s'+n))
                                    else (  0:sbIntDiv' x n s')
        where s' = 2*s+a
```

The algorithm and implementation are essentially the same. The only changes required are syntactic.

The lines starting with a double minus signed (`--`) are comments. The lines starting with a function name and followed by a double colon (`::`) are type definitions, and the remaining lines are the function definitions. The first three cases for `sbIntDiv` catch division by zero and avoid trivial divisions by one or minus one. The keyword `where` is a kind of postfix version of ML's `let ... in ...`

statement. It allows us to define variables which are used in the main expression, and makes the code extremely readable.

The only remaining differences are that 'cons' is represented using a single colon rather than the double one used in the algorithm description, and the three cases in the algorithm description are implemented using an if then statement.

6.3.2 Signed Binary Average

The implementation of the signed binary average algorithm is also extremely simple once we have the algorithm description. In section 4.3.1, we describe the algorithm as follows:

$$\mathrm{average}(x, y) = \mathrm{average}'(x, y, 0)$$

$$\mathrm{average}'(a_0 :: x', b_0 :: y', \frac{c}{2}) = \begin{cases} \mathrm{sign}(d') :: \mathrm{average}'(x', y', 0) & \text{if } d' \text{ even} \\ \mathrm{average}''(x', y', d') & \text{otherwise.} \end{cases}$$
$$\text{where} \quad d' = a_0 + b_0 + c$$

$$\mathrm{average}''(a_1 :: x'', b_1 :: y'', d') = e :: \mathrm{average}'(x'', y'', c')$$
$$\text{where} \quad d = 2d' + a_1 + b_1$$
$$e = \begin{cases} 1 & \text{if } (2 < d \leq 6) \\ 0 & \text{if } (-2 \leq d \leq 2) \\ -1 & \text{if } (-6 \leq d < -2) \end{cases}$$

The sign function is defined as follows:

$$\mathrm{sign}(x) = \begin{cases} -1 & \text{if } (x < 0) \\ 0 & \text{if } (x = 0) \\ 1 & \text{if } (x > 0) \end{cases}$$

This is implemented in Haskell as follows.

```
-- sbAv : Signed binary stream average operation
```

```
sbAv :: SBinStream -> SBinStream -> SBinStream
sbAv x y = sbAv' x y 0

-- sbAv' : Auxiliary function for sbAv
sbAv' :: SBinStream -> SBinStream -> Int -> SBinStream
sbAv' (a0:x) (b0:y) c = if (d' `mod` 2 == 0) then
                            ((signum d'):(sbAv' x y 0))
                        else sbAv'' x y d'
        where d' = (a0 + b0 + 2*c)


-- sbAv'' : Auxiliary function for sbAv
sbAv'' :: SBinStream -> SBinStream -> Int -> SBinStream
sbAv'' (a1:x') (b1:y') d' = (e : sbAv' (a1:x') (b1:y') c')
        where {d = (2*d' + a1 + b1);
               e = if (d >  2) then  1 else
                   if (d < -2) then -1 else 0;
               c' = d' - (2*e)}
```

7. Analysis

Chapter 3 gave a number of possible representations for exact real number computation, chapters 4, chapter 5 gave algorithms for useful operations using these representations, and chapter 6 discussed how the algorithms have been implemented using a functional language.

In this chapter, the performance of the algorithms is analysed. The fact that we are using several different representations means that there is often more than one way of performing a given operation, and these different approaches are compared. For example, to average several numbers using the dyadic representation requires fewer input digits from the numbers being averaged than performing the same operation using the signed binary representation, and hence less individual operations are performed. Unfortunately, the size of the dyadic digits can swell enormously during such calculations, making the dyadic operations themselves much slower than signed binary operations.

The implementation shows exact real arithmetic to be noticeably slower than floating point arithmetic. This is not entirely unexpected, but we show why this may be to an extent unavoidable.

By performing a theoretical analysis of the algorithms, we highlight some potential areas of interest. The implementation is then used as a tool to experiment with the algorithms and confirm the predictions, and show how their properties affect real performance.

7.1 Theoretical Analysis

The theoretical analysis begins with the simple operations such as converting between representations and averaging streams, and moves onto more complex operations and sequences of them. Features of particular interest include:

Lookahead: The number of digits of the input streams required to generate a specified number of output digits.

Branching: The number of stream operations required at each step of the algorithm. Some operations (eg. average) require an operation on the digits, the result of which is prefixed to the stream resulting from a single recursive call using the 'cons' operation. Others (eg. multiplication) may make several stream operations for each digit generated.

Digit size: In many of the operations using dyadic digits, there is the possibility of the size of the size of the representation of the digits swelling. It may interesting to compare similar work by Heckmann [15] [14] which analyses the way the size required for the representation of reals using linear fractional transformations behaves during certain computations.

These points are of interest because they affect the performance of the algorithms and the amount of memory they require. An algorithm with high lookahead requires many input digits to generate output. If the input is a complex expression itself, computing even a a few extra digits of this input may affect performance.

The branching will also affect the performance. An algorithm which branches will slow down over time as the number of stream operations required accumulates. An algorithm with no branching will continue generating new digits at a fairly constant rate (with respect to rate at which input digits are generated).

Dyadic digit swell will affect performance because the time required for primitive digit operations will dramatically increase.

7.1.1 Conversions between Representations

The conversion algorithms described in section 3.7 convert between the dyadic and signed binary stream representations.

Conversion from signed binary to dyadic streams is a trivial operation. We require one input digit from the signed binary stream to determine one dyadic output digit, or n input digits to generate n output digits.

Conversion from dyadic to signed binary streams is marginally more complex. We require two input digits from the dyadic stream to determine one output digit in the signed stream. More generally, in order to obtain n signed binary digits, we require $(n+1)$ dyadic digits. The algorithm itself requires three dyadic digit operations for each signed binary digit, but does not branch.

7.1.2 Average

The algorithms for averaging streams are described in sections 4.3.1 and 4.2.1.

Note that the lookaheads for addition and subtraction using (mantissa, exponent) representation are the same as for average, although the exponent also increases by one each time an addition or subtraction operation is performed.

Dyadic Stream Average

The dyadic stream average operation (section 4.2.1) is extremely simple; n input digits are required to generate n output digits, and each stage requires one dyadic digit operation. The size of the dyadic digits will not grow substantially as each digit of the output stream is simply the average of the two input digits. There is no branching as the algorithm proceeds.

Signed Binary Stream Average

The algorithm for computing the average of two signed binary streams (section 4.3.1) theoretically requires $(n+1)$ input digits to generate n output digits, although in certain cases (when the combination of input and carry digits allow the algorithm to determine the required output digit without examining further input), only n input digits will be required to generate n output digits. The algorithm does not branch as it proceeds.

7.1.3 Multiplication

The algorithms for dyadic stream and signed binary stream are described in sections 4.2.2 and 4.3.2.

Dyadic Stream Multiplication

Dyadic stream multiplication (section 4.2.2) requires $(n+1)$ input digits to generate n digits of output. This is clear from the algorithm which computes the average of two streams. The first digit of the first of these is determined directly from the first digits of the input, but the first digit of the second stream comes from the second digits of the two input streams.

The dyadic multiplication algorithm branches. Although only one recursive call to the multiplication function is made for each new digit, two stream average operations are also required. This means that n output digits will result in $(2n-2)$ average operations running concurrently. There is also the possibility of the size of the dyadic digits swelling as individual digits are multiplied at each step.

Consider $(a_0 :: a_1 :: a_2 :: x) \times (b_0 :: b_1 :: b_2 :: y)$. Using the algorithm we can expand this and get the first two digits as follows:

$$\begin{aligned}(a_0 \cdot b_0 ::\ & ((a_1 \cdot b_1 :: ((a_2 \cdot b_2 :: x \cdot y) \oplus (a_2 \cdot y \oplus b_2 \cdot x))) \\ & \oplus\ (a_1 \cdot (b_2 :: y) \oplus b_1 \cdot (a_2 :: x)))) \\ \oplus\ & a_0 \cdot (b_1 :: b_2 :: y) \oplus b_0 \cdot (a_0 :: a_2 :: x))\end{aligned}$$

Which gives us a result whose first two digits are as follows:

$$(a_0 \cdot b_0 \oplus (a_1 \cdot b_0 \oplus a_0 \cdot b_1)) \ :: \ ((a_1 \cdot b_1 \oplus (a_1 \cdot b_2 \oplus a_2 \cdot b_1)) \oplus (a_2 \cdot b_0 \oplus a_0 \cdot b_2)) \ :: \ \ldots$$

As we proceed, there are more and more average operations have been performed for each digit of output. In most cases where the input digits are random, a large numerator and denominator will be required to represent the result.

Signed Binary Stream Multiplication

The signed binary stream multiplication algorithm (section 4.3.2) is more complex than the dyadic version. Although there is no possibility of the digits swelling, the lookahead properties are worse and the branching different. The operation of the algorithm used to compute $(a :: b :: x) \times (c :: d :: y)$ is illustrated in figure 7.1.

The figure uses the fact that the average operation requires at most two digits of the input streams to compute one digit of output. The part of the tree which governs the lookahead is marked. To obtain one digit of output, a maximum of four digits is required from x and y, a total lookahead of six digits. Four two output digits, seven digits of the inputs are required. To obtain three output digits we also examine the recursive call. This requires a lookahead of six on the operands it was called with x and y, which is a total lookahead of eight. We can see that the maximum lookahead required for n digits is $(n + 5)$, and also that the minimum is $(n + 2)$ digits of the input streams. This range of possible lookaheads exists because the signed binary average algorithm may only require one digit of input to determine one digit of output in certain situations.

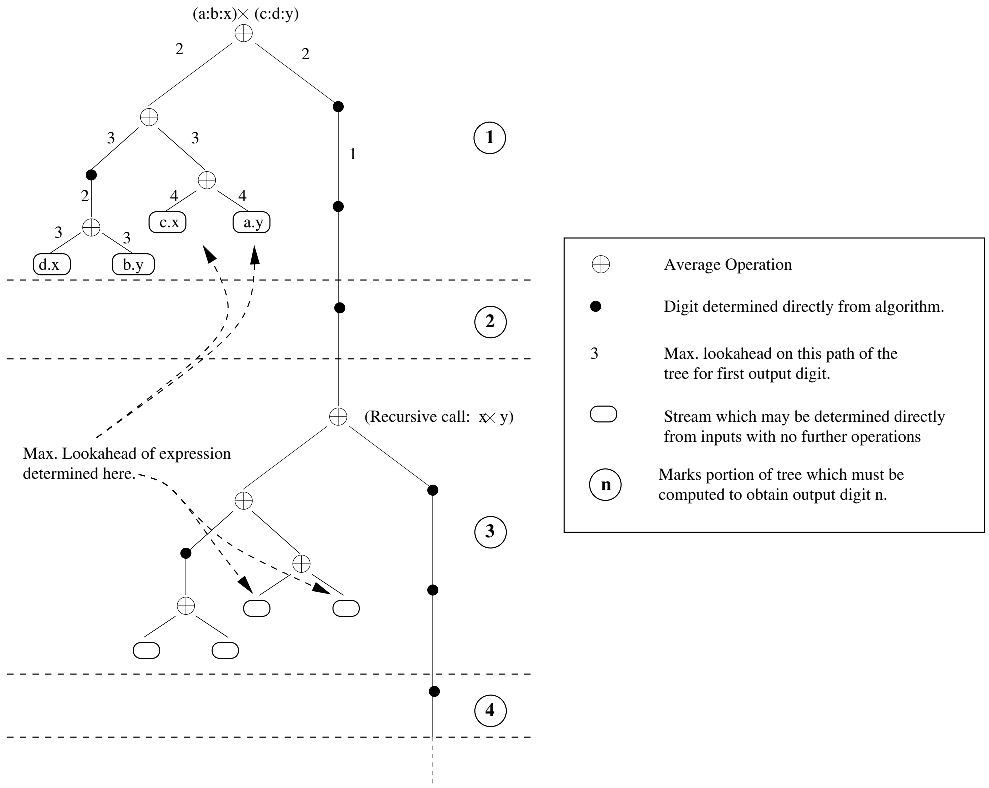

Figure 7.1: Behaviour of the signed binary multiplication algorithm used to compute $(a :: b :: x) \times (c :: d :: y)$.

The branching which will occur is also clear from the figure. The recursive call is made once for ever two digits of output. At each recursive call we introduce another four average operations. Hence obtaining n digits of output will result in $4\lfloor n/2 \rfloor$ average operations. In fact by optimising the algorithm to fix cases where the we are averaging streams with one input zero using 'cons' operations, etc., this becomes an upper bound on the number of average operations.

It is unlikely that this is the most efficient algorithm with regards lookahead requirements as we could get an algorithm from signed binary to signed binary with a constant of lookahead $(n+2)$ via the dyadic digits, albeit one which suffers from the problems of swelling dyadic digit sizes to some extent.

7.1.4 Division

The algorithms described so far all have all had lookaheads of the form $(n + c)$. Here we examine the division algorithm, whose lookahead behaviour is worse than this.

We first observe that there is no danger of the size of the dyadic digits swelling as we only generate one of seven possible digits at each step and that these are not used in other operations except in conversion back into signed binary digits.

Suppose we are computing the division x/y. In order to examine the lookahead we concentrate on the numerator x. It is possible to see from the way that the denominator is used that the lookahead in the denominator stream will always be less than or equal to the lookahead in the numerator stream.

The algorithm first removes all 0, $(1 :: \overline{1})$, and $(\overline{1} :: 1)$ combinations from the start of the numerator and adds an appropriate value to the exponent to leave the number represented unchanged. This is a fixed overhead and is not expensive except in as much as the numerator may itself be an expression which must be calculated digit by digit. The algorithm now proceeds by examining cases. The conversion of the dyadic result back into signed binary requires $(n + 1)$ dyadic digits of input to obtain n of output. We examine each case in turn and show how it affects the lookahead and branching.

Digits of x	Digits of $(x-y)$	Digits required to obtain n digits of numerator of the recursive call (in general).	Branching
$0 :: \ldots$	$\ldots$	$n+1$	None
$1 :: \overline{1} :: \ldots$	$\ldots$	$n+1$	None
$1 :: (0 \text{ or } 1) :: \ldots$	$\overline{1} :: 1 :: \ldots$	$n+3$	1 average
$1 :: (0 \text{ or } 1) :: \ldots$	$\overline{1} :: 1 :: \ldots$	$n+3$	1 average
$1 :: (0 \text{ or } 1) :: \ldots$	$\overline{1} :: (0 \text{ or } 1) :: \ldots$	$n+3$	1 average
$1 :: (0 \text{ or } 1) :: \ldots$	$0 :: \ldots$	$n+3$	1 average
$1 :: (0 \text{ or } 1) :: \ldots$	$1 :: \ldots$	$n+4$	2 averages
$\overline{1} :: 1 :: \ldots$	$\ldots$	$n+1$	None
$\overline{1} :: (0 \text{ or } 1) :: \ldots$	$\overline{1} :: 1 :: \ldots$	$n+3$	1 average
$\overline{1} :: (0 \text{ or } 1) :: \ldots$	$\overline{1} :: 1 :: \ldots$	$n+3$	1 average
$\overline{1} :: (0 \text{ or } 1) :: \ldots$	$\overline{1} :: (0 \text{ or } 1) :: \ldots$	$n+3$	1 average
$\overline{1} :: (0 \text{ or } 1) :: \ldots$	$0 :: \ldots$	$n+3$	1 average
$\overline{1} :: (0 \text{ or } 1) :: \ldots$	$1 :: \ldots$	$n+4$	2 averages

We now use these observations to work out how the lookahead for division behaves. To obtain one digit of output will require at most four digits of the numerator. In order to obtain two digits will require no more than four digits of the numerator of the recursive call, or eight digits of the original input. It is clear that the lookahead is at most $(4n + c)$, although for most cases the lookahead at each step is less than this, so for an average operation in which each case is equally likely a lookahead of $3n + c$ would be more likely. The lower bound is $(n + c)$. The algorithm branches. For n digits of output we would expect roughly n concurrent average operations to be required.

7.1.5 Logistic Map

The logistic map is a quadratic map defined as $f(x) = Ax(1-x)$ where A is a real constant. For some values of A, iterating the function produce chaotic behaviour (see [7] p.130-139). For the value 4, the behaviour is chaotic, and the function maps the interval $[0, 1]$ onto itself. The chaotic nature of the system produced has the effect that when computed using floating point arithmetic we rapidly get erroneous results. This is illustrated in section 2.1.1.

We examine the lookahead required to compute this using our implementation of exact real arithmetic supposing a is exactly equal to 4. Let $f(x) = 4x(1-x)$. Observe that

$$x \in [0, 1] \Rightarrow f(x) \in [0, 1]$$

so we could compute this using either the stream representation alone or the (mantissa, exponent). In fact, the lookahead can be smaller for the (mantissa, exponent) case if we implement multiplication by four by simply adding two to the exponent. This is a hollow saving, however, because in order to obtain the same precision we must evaluate more of the exponent. For example, if the exponent is n, and we require m signed binary digits of the result past the binary point, we must in fact evaluate $m + n$ digits of the mantissa.

Dyadic Arithmetic

Suppose we restrict ourselves to dyadic streams. In order to obtain n digits of $f(x)$, we compute $(1-x)$ which has a lookahead of $(n+1)$ (because the subtraction operation uses a shift), multiply this by x which gives $(n+2)$, and shift the result left to achieve multiplication by 4. This means the function will require $(n+4)$ lookahead. If this is now iterated, at the second iteration we will require $(n+4)$ digits of $f(x)$ which is $((n+4)+4)$ digits of x. Hence at the i^{th} iteration the lookahead will be $(n+4i)$.

If we use the (mantissa, exponent) representation in which a multiplication by four can be accomplished by simple modification of the exponent, the lookahead required will be $(n+i)$ as the only increase comes from the multiplication. Note that if the exponent increases, however, we need more digits of output to obtain the same precision.

Signed Binary Arithmetic

When we compute the logistic map using signed binary stream arithmetic, we observe that the maximum lookahead required is $(n+2)$ for subtraction, $(n+5)$ for multiplication and $(n+2)$ for the shift left (multiplication by 4). This gives us a maximum lookahead of $(n+9)$ for computation of $f(x)$, and $(n+9i)$ for $f(x)$ iterated i times. Note that this is an upper bound as multiplication may be able to compute digits with as little as $(n+2)$ lookahead and subtraction with $(n+1)$, which would give lookahead $(n+5i)$ as a lower bound.

The effect of this is that in order to compute 60 iterations of the logistic map to a precision of roughly 6 decimal digits (after this number of iterations our earlier test using double precision floating point arithmetic got no digits correct (see section 2.1.1), signed binary stream arithmetic would need to examine between 324 and 564 digits of input. In fact an experiment using the value `a` in section 7.2.2 as input showed the actual number of digits to be on the low side of this value at 382 digits. The actual lookahead will vary with the input used.

If we used signed binary (mantissa, exponent) arithmetic, the required upper bound would be only $(n+6i)$.

7.1.6 Sequences of operations

One interesting consideration is the following. Suppose we wish to compute

$$(\pi + (1 + (1 + (1 + \ldots (1 + 1) \ldots))))$$

Using signed binary (mantissa, exponent) representation we would require two digits of π to obtain one digit of output, although many more digits of the later numerals representing the number one. Suppose on the other hand we compute:

$$(1 + (1 + (1 + (1 + \ldots (1 + \pi) \ldots))))$$

Suppose we had 100 additions, we may require 101 digits from π, although the result of the computation is the same. This implies that applying the operations sensibly, and computing expressions in a sort of tree form would be sensible. For example, although both results are the same,

$$(((a+a)+(a+a))+((a+a)+(a+a)))+(((a+a)+(a+a))+((a+a)+(a+a)))$$

requires less digits of a than

$$(a+(a+(a+(a+(a+(a+(a+(a+(a+(a+(a+(a+(a+(a+(a+a)))))))))))))))$$

The first example requires only four digits of a to obtain one digit of output, whereas the second requires sixteen. In general if the second form requires n digits of a, one would expect the first form to require $\lceil ln(x) \rceil$ digits.

7.1.7 Summary of Theoretical Analysis

We have examined the two basic stream operations, average and multiplication, for the two basic stream representations. We have also examined conversions between these representations, the division algorithm, and sequences of operations.

The operations for the dyadic representation in general have a lower lookahead requirement than the corresponding signed binary ones, but also suffer from the problem of dyadic digit swell—especially the dyadic multiplication algorithm.

We have also seen that the division algorithm has a higher lookahead requirement, that a sequence of operations can rapidly lead to a high lookahead requirement, and that the order in which operations are performed can affect the lookahead complexity.

7.2 Experimental Analysis

In section 7.1 we examine some of the basic algorithms. In addition to this we perform various experiments in order to test the theoretical predictions, and to obtain statistics for the actual performance of the calculator implemented.

7.2.1 Strategies

Three strategies are employed to test the performance of the system. The first involves tracking the number of input digits required to obtain each output digit. The second is to use the Glasgow Haskell compilers profiling tool to discover which operations are contributing to the time and memory usage. The third is to simply find suitably complex operations and time them approximately to give a 'feel' for the overall performance of the system.

Lookahead

In order to track the input digits required to obtain certain digits of output we modify the data types used to store the digits to contain an additional 'lookahead' field. We number the lookahead of the digits in the input stream (ie. $1, 2, 3, \ldots$), and then we can extract the lookahead of the digits in the output stream.

The implementation of this is not entirely trivial. The readability and performance of the parts of code is seriously affected, so the lookahead experiments were implemented on a copy of the basic code and are separate from the calculator. It is achieved by modifying the basic operations so that when two digits are operated upon, the lookahead of the resulting digit is the maximum of the lookaheads of the operands. Similarly, if we introduce constants or pattern match in order to decide digits, we must extract the lookaheads from the digits used and ensure that the lookahead of the output digit is the maximum of its current lookahead and the lookahead of other digits used to decide it.

Profiling

The profiling tool records two types of information; memory usage and time requirements (see [25]). The results can be plotted graphically and illustrate roughly what portion of the time or space used is attributable to functions marked as being of interest.

Timing

Although the profiling reports give a timing measurement, these appear to be inaccurate and includes overheads due to the profiling itself. In fact we simply time some sample calculations performed using a compiled program using the unix `time` command to get measure the amount of CPU time required for the calculation. Note that these timings include the time required to execute the program and to compute the result back into decimal. The results give us an approximate measure of the real time requirements of the system.

7.2.2 Lookahead Results

In general, the lookahead experiments performed confirm the predictions made in section 7.1. When analysing the algorithms for dyadic stream average and multiplication, we observe that the lookahead is constant, and this was reflected in the results of the experimentation. The algorithms for signed binary average and multiplication and the division algorithm are more interesting because the lower and upper bounds on lookahead are not the same.

We show the results of experimentation with the lookahead of the signed binary multiplication and the division algorithms. These show how the average lookahead required might be related to the predicted lower and upper bounds. To illustrate the results we use two streams a and b are defined as follows:

```
a = [1, 0, -1, 0, 0, -1, 1, 0, 0, -1, 0, 1, 0, 1, 0, 1, -1, 0, 1, 0, 0, -1, 0, 1, 0,
     -1, 1, 0, -1, 0, 0, 0, 0, 0, 0, 1, 0, -1, 1, 0, -1, 0, 0, 0, 0, -1] ++ a

b = [1, 1, 0, -1, 0, -1, 0, 1, 0, 1, 0, -1, 0, 0, 1, -1, 0, 0, 0, 1, 0, 0, 0, 0, 0, -1,
     1, 0, -1, 0, -1, -1, 1, 0, -1, 1, -1, 0, 0, 0, -1, 0, 1, 0, 0, 0, 1, 0] ++ b
```

These numerals are random, have a roughly even mix of signed binary digits, and cycle with (different) periods of roughly 50 digits. The experimentation is not extensive or systematic, and only two of the results are given here.

Signed Binary Multiplication

The multiplication $a \times b$ has a predicted lookahead of between $(n+2)$ and $(n+5)$. Figure 7.2 illustrates this by plotting the lookahead at each digit. A lookahead of $(n+3)$ or $(n+4)$ appears to be more usual, although the minimum and maximum lookaheads are reached occasionally.

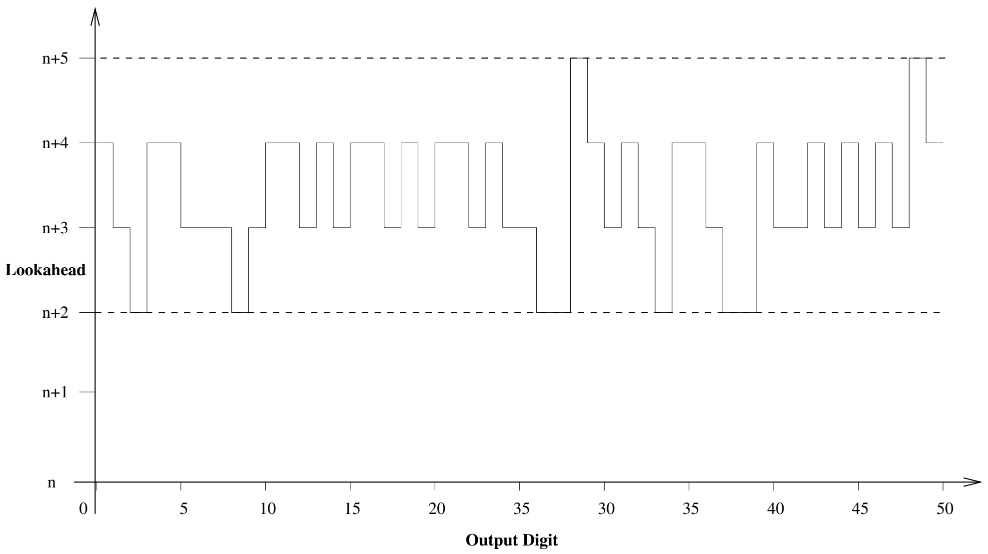

Figure 7.2: Lookahead required at each output digit when computing $a \times b$

Division

The division a/b shows a lookahead of roughly $2n$ (see figure 7.3). This is above the minimum possible (n), but well below the predicted maximum $4n$. Other experiments show this to be fairly typical behaviour unless the numerator and denominator are equal, in which case a lookahead of n will occur.

This experiment avoids the constant factor in the lookahead due to modification of the denominator mantissa and conversion back from dyadic and signed binary digits. In general a lookahead of $(2n + c)$ for some constant c appears probable using the division algorithm.

Note that the analysis given in this section is based upon the rate that the algorithm consumes digits from the input. It is possible that jumps in the algorithm steeper than the $4n$ predicted maximum may occur if part of the algorithm examines more digits without consuming them (and indeed this can be observed in figure 7.3). However this will never add more than a small constant to the lookahead and digits cannot be consumed faster than $4n$.

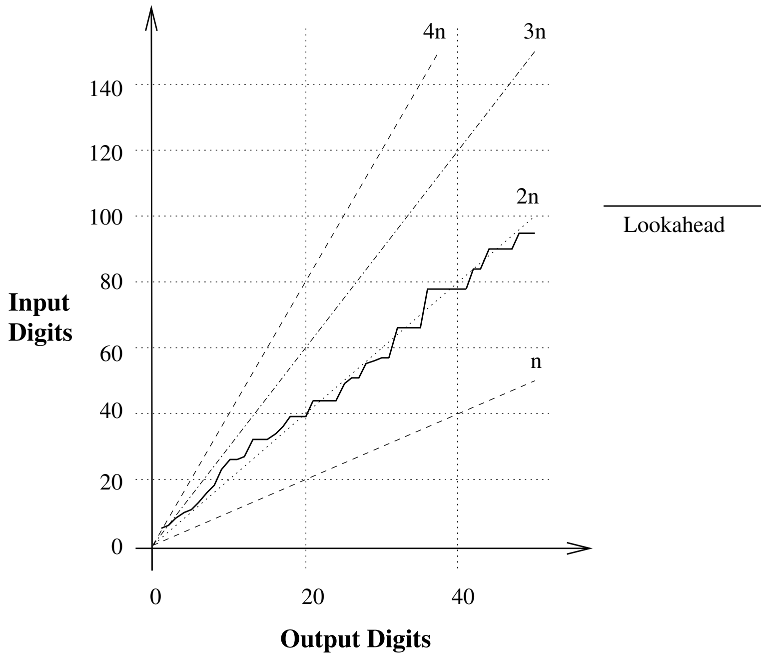

Figure 7.3: Input digits required to generate output digits of a/b

7.2.3 Dyadic digit swell

The problem of the size of the dyadic digit representations swelling during the course of a computation (see section 7.1 can be a serious problem. Experimentation shows that if we were to perform a series of computations involving dyadic stream multiplication such as computing iterations of the logistic map, for example, the problem can cause calculation to slow down very quickly. It is easy to see why by examining the first digit of output when computing ten iterations of the logistic map with the input $\frac{1}{2}$, which is the following large number.

```
388724570240705552142776682659458947805233224971859047752053338671439429888249069717757239313788490855334773356668182913613076126
552062213403564490610115613879629448773326009906579395235728097731596429190636911575213790785968204525040785029732203576471005350
951634165803606107538538116079472135738129467210780105351961615961185396421655274104732599578476551971538700680019225274588464976
895373596656654737071019377572902620812208450390621338292169853029681069459203464648330292157739569298813293253272386475249342402
851477594014885182650701889533684366050366443636678080968237909546976725623467384695888847956243028042083901779843462335576089740
372112324289481823120689109773824095307343395206347702861682801738167529775146670579837304392703285117954644610416233226 25/2^2558
```

This problem is one of the reasons the dyadic stream representation is not the main representation used by the calculator.

7.2.4 Profiling Results

Profiling experiments were performed with a variety of computations for both time and memory usage. The graph in figure 7.4 is an automatically generated summary of a profiling report generated using the Glasgow Haskell Compiler. The profiler also produces textual reports.

The graph shows the amount memory used by different functions against time. The function `sbIntDiv` performs division of a signed binary stream by an integer and `sbAv` is the signed binary stream average operation. The calculation performed here is computation of $ln(2)$ to a precision of 20 decimal digits.

In this example, the profiler shows that the majority of the memory is consumed during the signed binary stream average and division by an integer operations.

This and other experiments show that the majority of the time and memory are consumed in functions like the signed binary average, division of a signed binary stream by an integer, and a few other basic operations. This is not in itself surprising as this operation is used extensively in almost all other algorithms. It suggests, however, that the majority of the execution time is spent in a very small portion of the code, and that significant performance gains might be obtained by optimising a relatively small amount of the implementation.

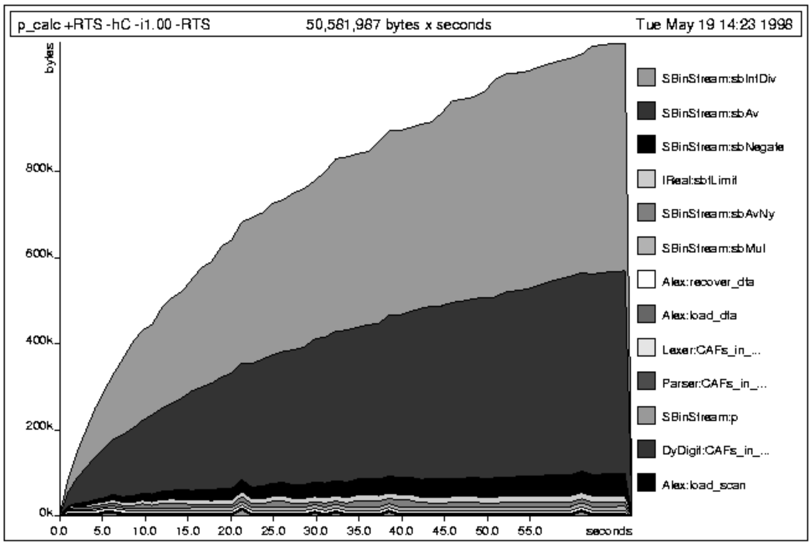

Figure 7.4: Example memory usage profiling graph, generated by the GHC profiler applied to the implemented calculator computing the expression ln(2) to 20 decimal digits

7.2.5 Timing Results

Timings are approximate and were conducted using a version of the calculator compiled with optimisation flags on a Sparc Ultra 1 model 140 with 64Mb of main memory using the unix command `time`. The precision is specified in decimal digits after the decimal point (or binary digits past the binary point for integrations or function maximum calculations).

Operation	Precision	Time (approx.)
exp(1)	50	0.66s
ln(2)	10	1.54s
π	50	3.88s
Logistic Map (Dyadic, 10 iterations)	10	16 mins 10s
Logistic Map (Signed Binary, 10 iterations)	10	0.10s
Logistic Map (Signed Binary, 50 iterations)	10	3.27s
ln(2)	30	28.78s
π	200	6 mins 43s
$\mathrm{fnmax}(0.23 + 1.1x - x^2, 0, 1)$	5 (binary)	34s
$\mathrm{fnmax}(0.23 + 1.1x - x^2, 0, 1)$	6 (binary)	5 mins 15s
$\int_0^1 x^2 dx$	4 (binary)	8 mins 41s

The timings show that performance varies significantly with different operations. The performance of different transcendental functions is in part due to the rate of convergence of the sequences used to compute them. The timings also show that the performance of the functional operations is terrible.

7.2.6 Summary of Experimental Analysis

The experimental analysis serves to confirm many of the predictions about lookahead made in section 7.1, and demonstrates how the lookahead requirement might behave in algorithms where we were unable to give more than a lower and upper bound on the lookahead requirement.

The experimental analysis also shows how the dyadic stream multiplication operation can cause massive swelling in the size of the digit representation and can cripple the performance. This problem is one of the reasons this representation is not used as the main one in the calculator.

The profiling experiments show that a significant portion of the execution time and the memory usage comes from just a few basic operations, suggesting that a high performance implementation could be achieved by using highly optimised code for the basic data structures and operations, and that most of the rest of the code has far less of an effect.

The timing experiments show generally poor performance. There are a number of reasons why this might be the case. Firstly that the algorithms themselves are more complex than corresponding floating point ones and examine more of the input than one would usually use in a floating point computation (as shown in the theoretical analysis in section 7.1. Secondly, infinite structures like streams are inherently more complex to manage than fixed sized data structures. Thirdly the functional implementation is not written for performance, and there are likely to be overheads imposed by the compiler used to generate the code used (see chapter 6).

8. Conclusion

In this work we have developed a calculator for exact real number computation and performed a theoretical analysis of the algorithms and experimented on their implementation.

We began by defining two representations of reals in the range $[-1, 1]$ using streams of digits. These were then extended to represent real numbers on the whole real line using a (mantissa, exponent) style representation. We showed how to convert between these representations, how to convert decimal numbers into a signed binary representation, and how to convert a finite portion of the signed binary representation back into decimal.

Algorithms for the basic arithmetic operations were implemented for these representations. A number of different techniques are used to obtain these algorithms, including exploiting the relationship between the list operation 'cons' and numerical average, using certain identities, and analysing of the range of possible values of a stream starting with a given digit or sequence of digits.

We developed an algorithm for the direct multiplication of two streams of signed binary digit. This is a more complex operation than the multiplication of dyadic digit streams, but avoids the problem of dyadic digit swell which can be observed if the dyadic digit multiplication is used to compute iterations of the logistic map, for example, and which can cripple performance.

We also developed an algorithm for the division of two signed binary numbers. This is significantly more complicated than the familiar school long division method because in the school long division method we know the whole denominator, whereas when dividing by a number with an infinite representation this is not the case. Other approaches such as Fourier's cross-division method (a description of which can be found in [30] p.159-164) which examine the denominator from right to left are also unsuitable because they allow digits which have already been output to be 'corrected' later on, and hence each output digit depends on an infinite number of digits of the numerator and denominator.

We also developed an algorithm to compute the limits of Cauchy sequences.

As far as we are aware, no other such algorithm has been developed in the literature. This algorithm significantly extends the range of computations which can be performed, and makes the calculator much more flexible. This is because of the many useful functions can be expressed as the limits of Cauchy sequences.

Finding algorithms for the transcendental functions is relatively straightforward once we have the algorithm for computing limits of Cauchy sequences. We simply find a method of generating a suitable Cauchy sequence using the inputs to the function whose limit is the output of the function, and about which we know either the rate, or in some cases just the manner in which the algorithm converges to its limit. The set of transcendental functions implemented here is not exhaustive, but it would be easy to extend the number of functions available using the technique described.

The functional operations integration and function minimum and maximum have also been implemented. The operations on streams (representing reals in the range $[-1, 1]$) are a modification of a Gofer implementation by Reinhold Heckmann. This work has been extended to the whole real line and integrated into the calculator allowing the operations to be applied to functions which use the complete set of arithmetic operations and transcendental functions. This is the first implementation of these algorithms on the whole real line.

A simple user interface has been created to demonstrate the calculator. The interface addresses the problem of allowing the user to view a (potentially time consuming) calculation as it proceeds but also giving an answer which can be used or checked, by outputting signed digits as they are computed and then converting the result into decimal when some specified precision is reached.

8.1 Summary of Analysis

Chapter 7 provides an analysis of some of the algorithms and the performance of the implementation. We can draw a number of conclusions from this.

Firstly, operations may be performed using either the dyadic or signed binary representations. Algorithms performing the same operation using different representations have different behaviours. In general the signed binary algorithms are more complex and have greater lookahead requirements, but on the other hand perform better than the corresponding dyadic digit operations which generally suffer if the size of the dyadic digits involved swell.

Secondly, computing certain expressions exactly (eg. the iterated logistic map) necessarily involves examining many more digits of input and performing a significantly greater number of manipulations than would normally be performed with floating point arithmetic.

Thirdly, the order in which operations are performed can greatly affect the lookahead required. Rearranging the same expression can significantly reduce the computation time of complex expressions.

Lastly, the present implementation is slow when compared to the floating point arithmetic packages commonly used, even when those operations are performed to a high precision in a package such as Maple. This is in part be due to the fact that the algorithms themselves are more complex than the floating point operations one might otherwise use (c.f. Chapter 7, especially section 7.1.5). However the fact that the implementation uses a functional language and most floating point arithmetic is either written using an imperative language, assembler, or embedded in hardware makes it unreasonable to make a direct comparison.

8.2 Evaluation of Implementation

The implementation used the functional language Haskell. This was an ideal choice for this work because the availability of lazy lists made working with streams extremely easy, and features such as unbounded length integers, good I/O facilities, and a module system all simplified aspects of the work which simply enable the work on exact real arithmetic to continue without being of direct interest in and of themselves. In addition, the availability of both an interpreter and compiler gave the benefits of easier development, testing, and experimentation afforded by an interpreter with a higher performance end product from the compiler. Tools such as a profiler, the parser generator "Happy", and the lexer generator "Alex" aided the development. The choice of language was discussed further in section 6.2.1.

We have observed that the performance of the calculator appears poor when compared with floating point arithmetic. However one would expect to have to sacrifice some efficiency of other approaches for the benefits afforded by the use of this approach. In fact, an application involving real number computation would probably only use these exact approaches where it was strictly necessary, and would use optimised code to do so. It is important to remember that this implementation has been conducted with simplicity, readability, and flexibility in mind, and is not optimised for performance.

It may be possible to significantly improve the performance of the transcendental functions. There are two ways in which this might be performed. Firstly there may be cases in which it is possible to find a Cauchy sequence whose limit is the desired result and which converges more quickly than the ones used in this implementation. Alternatively specific algorithms could be found which avoid the use of nested sequences of intervals altogether. Early implementations of division, for example, used a iterative approximation to compute the reciprocal of the denominator in the same way as the transcendental functions, and this result was then multiplied by the numerator. The division algorithm developed in section 4.4.4 was significantly faster.

8.3 Possible Extensions and Future Work

In chapter 1 we described many approaches to exact real arithmetic. Unfortunately it is very difficult to compare these approaches. One direction for future work would be to perform a theoretical comparison of these approaches. One could also implement systems using the different approaches and compare their performance in some standardised way.

The performance achieved in the calculator implemented is poor when compared with commercial implementations of high precision floating point arithmetic. It would be interesting to implement an optimised version of these operations in an imperative language.

It may be possible to achieve improved performance by parallelising some of the algorithms. For example, many of the algorithms developed exhibit some form of branching. One simple approach to parallelising these would be to allocate evaluation of the stream on each branch of the expression to different processors.

Chapter 7 gives some theoretical analysis of the basic algorithms. An interesting idea would be to analyse these algorithms in more detail, possibly extending it to the transcendental functions too, and investigate their complexity. In particular it would be interesting to see what minimum bounds on the lookahead and complexity exist, and whether they are achieved here or are achievable using other algorithms on these representations.

Another observation about the calculator is that although some numerals must be represented using an infinite number of digits, we sometimes know that a number is represented exactly by a finite portion of the numeral. Suppose, for example, we wish to compute a simple expression (eg. $(2 + 3)$), because we know that in fact the numbers can be represented exactly by a finite number of digits, we need never perform more than a finite amount of the computation. The current approach will output correct result, but would then continue computing indefinitely. If we were to introduce a terminator and allow computations with finite length numerals as well as infinite length ones, we may be able to improve performance on rational computations and allow the system to stop when the result is computed exactly.

The use of signed binary digits may not be the most effective digit representation for arithmetic operations. It may turn out that similar algorithms applied to a digit in a larger (eg. 2^{15}) or smaller base (eg. the golden ratio [8, 9]) may be more efficient, either in a theoretical sense, or by exploiting the fact that all operations on integers which may be represented in one word are executed in the same time, regardless of their size. It would be interesting to investigate this possibility further.

Lastly, in section 7.1.6 we saw how the form of an expression could affect its computational complexity. One interesting area for development which might also be useful in other approaches to exact real arithmetic would be to develop a compiler for optimising exact real arithmetic which used some strategy (heuristic or otherwise) to manipulate or re-arrange expressions to make them simpler to compute. This might even involve techniques from computer algebra which could perform more complex simplifications than merely re-arranging of expressions.

Bibliography

[1] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:389–400, 1961.

[2] David H. Bailey, Jonathon M. Borwein, Peter B. Borwein, and Simon Plouffe. The quest for pi. `http://www.cecm.sfu.ca/~pborwein/PAPERS/P130.ps`, June 1996.

[3] H. J. Boehm, R. Cartwright, M. J. O'Donnel, and M. Riggle. Exact real arithmetic, a case study in higher order programming. In *Proceedings of the ACM conference on Lisp and functional programming*, pages 162–173, August 1986.

[4] Hans Boehm and Robert Cartwright. Exact real arithmetic, formulating real numbers as functions. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 3, pages 43–64. Addison-Wesley, 1990.

[5] Hans-Juergen Boehm. Constructive real interpretation of numerical programs. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 214–221. ACM Press, June 1987.

[6] L. E. J. Brouwer. Besitzt jede reelle Zahl eine Dezimalbruchentwicklung. *Math Ann*, pages 210–210, 1920.

[7] R. L. Devaney. *An Introduction to Chaotic Dynamical Systems*. Addison-Wesley, California, 2nd edition, 1989.

[8] Pietro di Gianantonio. *A functional approach to computability on real numbers*. PhD thesis, University of Pisa, 1993. Technical Report TD 6/93.

[9] Pietro di Gianantonio. A golden ratio notation for the real numbers. Technical report, CWI, 1997.

[10] Abbas Edalat and Peter John Potts. A new representation for exact real numbers. *Electronic Notes in Theoretical Computer Science*, 6, 1997. `http://www.elsevier.nl/locate/entcs/volume6.html`.

[11] Martín Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162:79–115, 1996.

[12] Martín Escardó. Effective and sequential definition by cases on the reals via infinite signed-digit numerals. Unpublished Note, April 1998.

[13] R. W. Gosper. Item 101b: Continued fraction arithmetic. *HAKMEM, MIT Artificial Intelligence Memo 239*, pages 39–44, February 1972.

[14] Reinhold Heckmann. The appearance of big integers in exact real arithmetic based on linear fractional transformations. In *Proc. Foundations of Software Science and Computation Structures*, Lisbon, 1998.

[15] Reinhold Heckmann. Contractivity of linear fractional transformations. In Chesneaux *et al*, editor, *Proceedings of the Third Real Numbers and Computers Conference*, pages 45–59, Université Pierre et Marie Curie, Paris, France, April 1998.

[16] John Hughes. Why functional programming matters. *Computer Journal*, 32(1), 1989.

[17] Donald E. Knuth. *The Art of Computer Programming, Seminumerical algorithms*, volume 2. Addison-Wesley, 2nd edition, 1981.

[18] Ker-I Ko. *Complexity Theory of Real Functions.* Birkhauser, Boston, 1991.

[19] Peter Kornerup and David Matula. An algorithm for redundant binary bit-pipelined rational arithmetic. *IEEE Transactions on Computers*, 39(8):1106–1115, August 1990.

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

[21] Asger Munk Nielsen and Peter Kornerup. MSB-first digit serial arithmetic. *Journal of Universal Computer Science*, 1(7):527–547, 1995.

[22] Peterson, Hammond, Augustsson, Boutel, Fasel, Gordon, Hughes, Hudak, Johnsson, Jones, Meijer, Peyton-Jones, Reid, and Wadler. Report on the

programming language Haskell - a non-strict purely functional language - version 1.4. `http://haskell.org/`, April 1997.

[23] P. J. Potts, A. Edalat, and M. H. Escardó. Semantics of exact computer arithmetic. In *Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 248–257, Warsaw, Poland, 1997. IEEE Computer Society Press.

[24] Peter John Potts and Abbas Edalat. Exact real computer arithmetic. Imperial College, March 1997.

[25] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. Research Report FP-1994-10, 1994. Dept. of Computing Science, University of Glasgow.

[26] Alex K. Simpson. Lazy functional algorithms for exact real functionals (extended abstract). Submitted, 1998.

[27] Philipp Sünderhauf. Incremental addition in exact real arithmetic. Technical report, The Department of Computing, Imperial College, London, December 1997.

[28] Alan Turing. A correction. *The London Mathematical Society*, 43:544–546, 1937.

[29] Alan Turing. On computable numbers, with an application to the Entscheindungproblem. *The London Mathematical Society*, 42:230–265, 1937.

[30] J. V. Uspensky. *Theory of Equations*. McGraw-Hill, New York, 1948.

[31] Jean E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, August 1990.

[32] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[33] E. Wiedmer. Computing with infinite objects. *Theoretical Computer Science*, 10:133–155, 1980.

Appendix A. Dyadic Digit Operations

This appendix develops primitive operations on the dyadic digits defined in chapter 3, and used extensively in chapter 4. The algorithms assume a representation of dyadic digits as pairs of integers (see section 6.2.5). Other representations will require different algorithms.

We show how to convert a dyadic rational into its lowest terms, perform primitive operations, and perform equality and other relational tests on pairs of dyadic digits. Division is not closed over the dyadic digits (unlike dyadic rationals), and is not required as a primitive operation.

A.1 Lowest Terms

When implementing some of the later operations on dyadic digits, we sometimes generate a numeral $\frac{a}{2^b}$, but cannot guarantee that (a, b) is a valid dyadic digit in its lowest term representation. Given a pair of integers (a, b) with the properties $|a| \leq 2^b$ and $b \geq 0$, we show how to generate the dyadic digit (a', b') such that $[\![(a', b')]\!] = \frac{a}{2^b}$.

The operation is performed by examining a. If a is zero or odd, the dyadic digit may be obtained directly as follows:

$$\begin{aligned} a = 0 \quad &\Rightarrow \quad \frac{a}{2^b} = [\![(0, 0)]\!] \\ a \text{ is odd} \quad &\Rightarrow \quad \frac{a}{2^b} = [\![(a, b)]\!] \end{aligned}$$

If a is even and not equal to zero, we observe that $(b > 0)$ because $(|a| \leq 2^b)$, and so:

$$a \text{ is even} \quad \Rightarrow \quad \frac{a}{2^b} = \frac{a \text{ div } 2}{2^{b-1}}$$

In this case we simply recursively call the function with the arguments $(a/2, b-1)$ in this case until a is zero or odd, at which point we can return the dyadic digit.

A.2 Average

In order to compute the average of two dyadic digits

$$(a, b) \oplus (c, d) = \frac{a}{2^b} \oplus \frac{c}{2^d}$$

it is necessary to compute the value

$$\frac{x}{2^y} = \frac{a \cdot 2^d + c \cdot 2^b}{2^{b+c+1}}$$

with the property that (x, y) is a dyadic digit (ie. x is odd or $(x, y) = (0, 0)$).

We examine the powers of the denominators b and d and come up with three different cases.

Case 1: $(b > d)$

$$\frac{x}{2^y} = \frac{a \cdot 2^d + c \cdot 2^b}{2^{b+c+1}} = \frac{a + c \cdot 2^{b-d}}{2^{b+1}}$$

As $(b > d)$, we know $(a, b) \neq (0, 0)$, so a is odd, $(c \cdot 2^{b-d})$ is even, and hence $(a + c \cdot 2^{b-d})$ is odd. Hence:

$$b > d \Rightarrow (a, b) \oplus (c, d) = (a + c \cdot 2^{b-d}, b + 1)$$

Case 2: $(d > b)$ As average is commutative, we can treat this in exactly the same way as the previous case, but with the arguments reversed. Hence:

$$d > b \Rightarrow (a, b) \oplus (c, d) = (a \cdot 2^{d-b} + c, d + 1)$$

Case 3: $(b = d)$:

$$\frac{x}{2^y} = \frac{a \cdot 2^d + c \cdot 2^b}{2^{b+c+1}} = \frac{a+c}{2^{b+1}}$$

However as a and c are odd or zero, $(a+c)$ may be even, or $(a+c, b+1)$ may have a zero first element and non-zero second element. Therefore it is necessary to apply the function described in section A.1 to the pair $(a+c, b+1)$.

Using an almost identical approach, it is possible to define similar functions for finding the average of one digit with the negation of another, addition or subtraction of two digits (with the caveat that if the result of the addition or subtraction is not in the range $[-1, 1]$, the pair returned will not be a valid dyadic digit), and other combinations such as addition or subtraction of two dyadic digits (a, b) where a pair of digits (x, y) is returned with the property that:

$$(x, y) = \begin{cases} (-1, [\![a]\!] + [\![b]\!] - 1) & \text{if } [\![a]\!] + [\![b]\!] < 1 \\ ([\![a]\!] + [\![b]\!], 0) & \text{if } \left|[\![a]\!] + [\![b]\!]\right| \leq 1 \\ (1, [\![a]\!] + [\![b]\!] + 1) & \text{if } [\![a]\!] + [\![b]\!] > 1 \end{cases}$$

A.3 Multiplication

Multiplication of dyadic digits is simpler than the average operation. Given two dyadics (a, b) and (c, d), we know that a and c are both odd or zero. Hence, if either a or c are zero, the result will be the dyadic digit $(0, 0)$. Otherwise, $a \cdot c$ is odd, so the result is:

$$(a, b) \times (c, d) = (a \cdot c, b + d)$$

because

$$[\![(a \times c, b + d)]\!] = \frac{a \cdot c}{2^{b+d}} = \frac{a}{2^b} \times \frac{c}{2^d} = [\![(a, b)]\!] \times [\![(c, d)]\!]$$

A.4 Shift operators

Multiplication or division by a power of two (2^n say) of a dyadic digit (a, b) reduces to simply adding or subtracting n to or from b. Provided (a, b) is not zero (ie. $(0, 0)$, for which the zero digit is returned). Range checking may also be performed on the shift left (multiplication by positive powers of 2) operation to ensure that the result remains in the range $[-1, 1]$.

A.5 Relational Tests

Equality of dyadic digits is simply the equality of its elements.

$$(a, b) = (c, d) \Rightarrow (a = c) \text{ and } (b = d)$$

In order to test other relationships, we first express the numerator terms of the dyadic fraction in terms of the same denominator, and then test the numerators. For example, to test whether $(a, b) \geq (c, d)$, let

$$(a', c') = \begin{cases} (a, c) & \text{if } b = d \\ (a, c \cdot 2^{b-d}) & \text{if } b > d \\ (a \cdot 2^{d-b}, c) & \text{if } b < d \end{cases}$$

Now return the result $(a' \geq c')$. Other relational operators may be substituted for the greater than or equal ($\geq$) to perform other tests.

Appendix B. Algorithms

B.1 Division

```
-- sbDiv : Division of two (corrected) signed binary streams. This
--         function is used by the sbfDiv function. Note that negative
--         denominator streams result in the negation of both
--         denominator and numerator so the denominator is positive.
sbDiv :: SBinStream -> SBinStream -> DyStream

sbDiv x ( 1:y) = sbDiv' x (1:y)
sbDiv x (-1:y) = sbDiv' (sbNegate x) (1:sbNegate y)
sbDiv _ ( 0:_)      = undefined
sbDiv _ ( 1: -1: _) = undefined
sbDiv _ (-1:  1: _) = undefined


-- sbDiv_emit : Generate specified dyadic digit and make the appropriate
--              recursive call. This function makes the code more
--              readable. Used by sbDiv'.
sbDiv_emit :: Int -> SBinStream -> SBinStream -> DyStream

sbDiv_emit ( 4) x y = (dyd_One          : sbDiv' x y)
sbDiv_emit ( 2) x y = (dyd_Half         : sbDiv' x y)
sbDiv_emit ( 1) x y = (dyd_Quarter      : sbDiv' x y)
sbDiv_emit ( 0) x y = (dyd_Zero         : sbDiv' x y)
sbDiv_emit (-1) x y = (dyd_minusQuarter : sbDiv' x y)
sbDiv_emit (-2) x y = (dyd_minusHalf    : sbDiv' x y)
sbDiv_emit (-4) x y = (dyd_minusOne     : sbDiv' x y)


-- sbDiv' : The body of the division algorithm, used by sbDiv.
sbDiv' :: SBinStream -> SBinStream -> DyStream

sbDiv'   ( 1: -1:x') y = (sbDiv_emit 0 ( 1:x') y)
sbDiv'   (-1:  1:x') y = (sbDiv_emit 0 (-1:x') y)
sbDiv'   ( 0:x') y     = (sbDiv_emit 0 x' y)

sbDiv' x@( 1:x') y     = case a of {
    (-1) -> if (a'==1) then  (sbDiv_emit 2 (-1:r'') y) else
                             (sbDiv_emit 1 (p 0 (sbSub x (0:y))) y);
     0  -> sbDiv_emit 2 r' y;
```

```
     1  -> if (a'== -1) then  (sbDiv_emit 2 (1:r'') y) else
                              (sbDiv_emit 4 (p 0 (sbSub r y)) y)}

        where r@(a:r'@(a':r'')) = sbSub x y

sbDiv' x@(-1:x') y = case a of {
    1  -> if (a'== -1) then  (sbDiv_emit (-2) (1:r'') y) else
                             (sbDiv_emit (-1) (p 0 (sbAdd x (0:y))) y);
    0  -> sbDiv_emit (-2) r' y;
   (-1) -> if (a'== 1) then  (sbDiv_emit (-2) (-1:r'') y) else
                             (sbDiv_emit (-4) (p 0 (sbAdd r y)) y)}

        where r@(a:r'@(a':r'')) = sbAdd x y




-- fixinput' : Auxiliary function used by fixinput.
--             Fixes a number so that it always start with a
--             non-zero digit.
fixinput' :: SBinStream -> Integer -> SBinFloat
fixinput' ( 0:  x)    n = fixinput'     x  (n+1)
fixinput' ( 1: -1: x) n = fixinput' ( 1:x) (n+1)
fixinput' (-1:  1: x) n = fixinput' (-1:x) (n+1)
fixinput' x           n = (n,x)

fixinput :: SBinStream -> SBinFloat
fixinput x = fixinput' x 0


-- sbfDiv : Top level division function.
sbfDiv :: SBinFloat -> SBinFloat -> SBinFloat
sbfDiv (ex, mx) (ey,my) = dyfToSbf (dyf (ex-ey+ey_fix+2, sbDiv mx my'))
        where (ey_fix, my') = fixinput my
```

B.2 Limits

```
type SbInterval = (SBinFloat, SBinFloat)
type SbIntervalx = (SBinStream, SBinStream)

type IReal = [SbInterval]
type IRealx = [SbIntervalx]
```

```
-- sbBB : Signed binary stream 'banana bracket' operation
--          (Simplified call. Could use general call instead).
sbBB :: SBinStream -> SBinStream -> SBinStream -> SBinStream
sbBB x@(a:x') y z@(c:z')
-- | r_lo > s_hi = undefined     -- Debug only
 | (r_lo' >= -4) && (s_hi' <= 4)
                 = ( 0: sbBB (p 0 x) (p 0 y) (p 0 z))
 | r_lo' >= 0    = ( 1: sbBB (p 1 x) (p 1 y) (p 1 z))
 | s_hi' <= 0    = (-1: sbBB (p (-1) x) (p (-1) y) (p (-1) z))
 | r_hi' < s_lo' = y
 | otherwise   = sbBB' 2 r' s' [a] [c] x' y z'
     where {r'@(r_lo', r_hi') = (-4 + a*4, 4 + a*4);
            s'@(s_lo', s_hi') = (-4 + c*4, 4 + c*4)}


-- sbBB' : Signed binary stream 'banana bracket' operation
--           (General call).
--         r and s represent the range of the lower and upper
--           bounds x and z from digits evaluated so far.
--           eg. [-8,0] -> [-1,0],  [2,6] -> [1/4,3/4].
--         sig represents the amount by which the next digit may
--           vary r and s.
--         outx and outz are digits from x and z seen so far.
--         x and z are lower and upper bounds (x <= z)
--         y is the number being 'banana bracketed', (x <= y <= z).
--
--         The choice of [-8,8] to represent the range [-1,1] is
--           based on the fact that after examining 3 digits we
--           can either emit a digit, or determine that x <> z.
--         r' and s' are the new ranges of the lower/upper bounds
--           after the digits a and c are considered.
sbBB' :: Int -> (Int, Int) -> (Int, Int) -> [Int] -> [Int] ->
         SBinStream -> SBinStream -> SBinStream -> SBinStream
sbBB' sig (r_lo,r_hi) (s_lo,s_hi) outx outz x@(a:x') y z@(c:z')
-- | sig == 0    = undefined     -- Debug only
-- | r_lo > s_hi = undefined     -- Debug only
 | (r_lo' >= -4) && (s_hi' <= 4)
              = ( 0: sbBB (p 0 (outx++x)) (p 0 y) (p 0 (outz++z)))
 | r_lo' >= 0 = ( 1: sbBB (p 1 (outx++x)) (p 1 y) (p 1 (outz++z)))
 | s_hi' <= 0 = (-1: sbBB (p (-1) (outx++x)) (p (-1) y) (p (-1) (outz++z)))
 | r_hi' < s_lo' = y
 | otherwise   = sbBB' (sig `div` 2) r' s' (outx++[a]) (outz++[c]) x' y z'
```

```
     where {r'@(r_lo', r_hi') = (r_lo + (a+1)*sig, r_hi + (a-1)*sig);
            s'@(s_lo', s_hi') = (s_lo + (c+1)*sig, s_hi + (c-1)*sig)}



-- sbfBB : 'Banana Bracket' operation on signed binary floats.
--          Note: exponent specified by parameter e, and absent from
--                output.
sbfBB :: Integer -> SBinFloat -> SBinFloat -> SBinFloat -> SBinStream
sbfBB e (ex, mx) (ey,my) (ez, mz) =
        sbBB (sbShift mx (e-ex)) (sbShift my (e-ey)) (sbShift mz (e-ez))



-- sbfLimit' : Convert a sequence of nested intervals into an SBinFloat
--             with specified exponent
sbfLimit' :: Integer -> IReal -> SBinFloat
sbfLimit' e ((x, z): y) = (e, sbfBB e x (sbfLimit' e y) z)


-- sbfLimit : Convert a sequence of nested intervals into an SBinFloat
--            by first finding max exponent required, and
--            then using sbfLimit'.
--            We use sbfNormMax with 500 (an arbitrary number) so that
--            if we see zero it won't fall over. In fact it is only for
--            efficiency that we bother normalising at all, so this doesn't
--            really matter.
sbfLimit :: IReal -> SBinFloat
sbfLimit ((x, z): y) = sbfLimit' (max ex ez) ((x,z):y)
        where {(ex, _) = sbfNormMax x 500;
               (ez, _) = sbfNormMax z 500}
```

Appendix C. Expression Language

Figure C.1 illustrates the grammar of the expression language. It was automatically generated by Peter Thiemann's "Ebnf2ps" package.

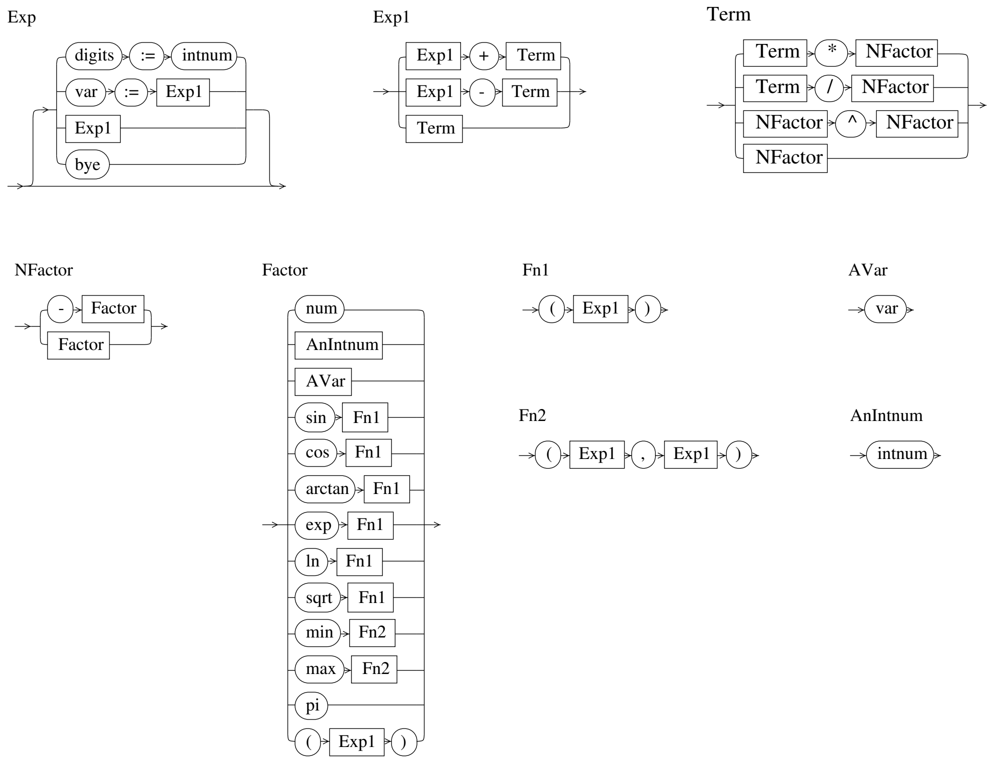

Figure C.1: The calculator's expression language (excluding integration)

```
Calculator version 1.0:

==> (1/3)
(1/3)
2^(1) * 0.  2 -4 7 -4 7 -4 7 -4 7 -4
= 0.3333333333
```

```
==> (1/3)*3
(1/3)*3
2^(4) * 0.  0 6 2 5 0 0 0 0 0 0 0
= 1

==> a := sin(pi/3)
a := sin(pi/3)
Okay.

==> b := exp(2)
b := exp(2)
Okay.

==> sqrt(b+(7*a-3))
sqrt(b+(7*a-3))
2^(3) * 0.  4 0 4 1 0 5 -4 0 3 -1
= 3.2328368232

==> Digits:=35
Digits:=35
Digits = 35

==> pi
pi
2^(5) * 0.  1 0 -2 2 -2 -5 -2 -3 0 4 2 5 -3 -2 1 0 4 -1
 -3 0 2 0 -4 -2 -4 0 6 -3 3 -2 -5 -2 4 5 -4 5
= 3.1415926535897932384626433832795028 8

==> c := 3
c := 3
Okay.

==> d := c+1
d := c+1
Okay.

==> Digits := 15
Digits := 15
Digits = 15

==> d
d
2^(4) * 0.  2 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0
= 4
```

```
==> c := 2
c := 2
Okay.

==> d
d
2^(4) * 0.  2 -1 -2 -5 0 0 0 0 0 0 0 0 0 0 0
= 3

==> e
e
No such var

==> 34-
34-
Parse Error

==> notafunction(23)
notafunction(23)
Parse Error

==> exit
exit
Exiting ...
```