Coinductive Definitions and Real Numbers

BSc Final Year Project Report

Michael Herrmann
Supervisor: Dr. Dirk Pattinson
Second Marker: Prof. Abbas Edalat

June 2009

Abstract

Real number computation in modern computers is mostly done via floating point arithmetic which can sometimes produce wildly erroneous results. An alternative approach is to use *exact real arithmetic* whose results are guaranteed correct to any user-specified precision. It involves potentially infinite data structures and has therefore in recent years been studied using the mathematical field of *universal coalgebra*. However, while the coalgebraic definition principle *corecursion* turned out to be very useful, the coalgebraic proof principle *coinduction* is not always sufficient in the context of exact real arithmetic. A new approach recently proposed by Berger in [3] therefore combines the more general *set-theoretic* coinduction with coalgebraic corecursion.

This project extends Berger's approach from numbers in the unit interval to the whole real line and thus further explores the combination of coalgebraic corecursion and set-theoretic coinduction in the context of exact real arithmetic. We propose a coinductive strategy for studying arithmetic operations on the signed binary exponent-mantissa representation and use it to define and reason about operations for computing the average and linear affine transformations over $\mathbb{Q}$ of real numbers. The strategy works well for the two chosen operations and our Haskell implementation shows that it lends itsels well to a realization in a lazy functional programming language.

Acknowledgements

I would particularly like to thank my supervisor, Dr. Dirk Pattinson, for taking me on as a project student at an unusually late stage and for his continuous support. He provided me with all the help I needed and moreover always found time to answer questions that were raised by, but went far beyond, the topics of this project. I would also like to thank my second marker, Professor Abbas Edalat, for his feedback on an early version of the report.

I am extremely grateful to my family for enabling me to study abroad even though this demands a great deal of them, in many respects. Likewise, I would like to thank my girlfriend Antonia for standing by me in the last three labour-intensive years. This project is dedicated to her.

Contents

Chapter 1

Introduction

1.1 Why Exact Real Number Computation?

The primary means of real number calculation in modern computers is floating point arithmetic. It forms part of the instruction set of most CPUs, can therefore be done very efficiently and is a basic building block of many computer languages. The fact that it is used for astronomic simulations which require many trillions ($= 10^{12}$) of arithmetic operations shows that, although being an approximation, it can be used for applications that require a high degree of accuracy. This results in a high level of confidence in floating point arithmetic.

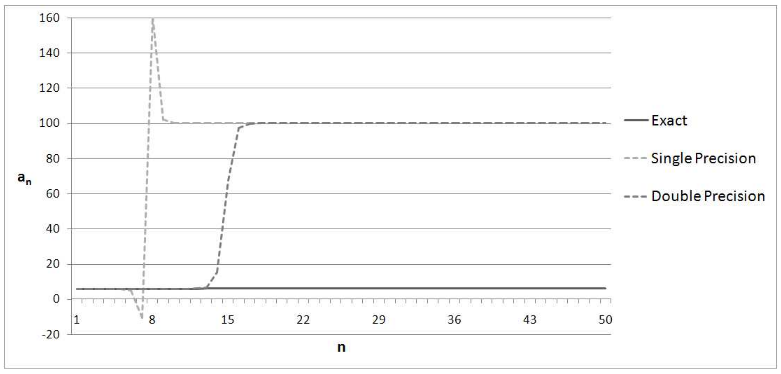
Unfortunately, however, there are cases where floating point arithmetic fails. Consider for example the following sequence discovered by Jean-Michel Muller (found in [6, 14]):

$$a_0 = \frac{11}{2},\ a_1 = \frac{61}{11},\ a_{n+1} = 111 - \frac{1130 - 3000/a_{n-1}}{a_n} \tag{1.1}$$

It can easily be shown via induction that

$$a_n = \frac{6^{n+1} + 5^{n+1}}{6^n + 5^n} \quad \left(= \frac{6}{1 + \left(\frac{5}{6}\right)^n} + \frac{5}{\left(\frac{6}{5}\right)^n + 1} \right)$$

from which we deduce that (a_n) converges to 6. However, when using the C programs given in Appendix A.1 to calculate the first few terms of Equation (1.1) with (IEEE 754) floating point arithmetic, we obtain the following results:



Already after 6 iterations (that is, after a mere 12 divisions and 10 subtractions), single precision floating point arithmetic yields wrong results that make it seem like the sequence converges to 100. Double precision performs slightly better, however only in that it takes a little longer until it exhibits the same behaviour. Interestingly, this trend continues when the size of the number representation is increased: using higher precisions only seems to defer the apparent convergence of the values to 100 [14].

Another surprising property of this sequence is that sometimes the error introduced by the floating point approximation *increases* when the precision is increased. We use the Unix utility **bc** similarly to [6] to compare two approximations to a_{10} (for the code see Appendix A.1). Using number representations with precisions of 8 and 9 decimal places, respectively, we obtain:

Precision	Computed Value	Abs. Dist. from a_{10}
8	110.95613220	105.09518068
9	-312.454929592	318.315881114

We can see that the approximation error triples (!) when increasing the precision from 8 to 9 decimal places. This is a strong counterexample to the commonly held belief that using a larger number representation is sufficient to ensure more accurate results.

Several approaches have been proposed to overcome the problems of floating point arithmetic, including interval arithmetic [9], floating point arithmetic with error analysis [11], stochastic rounding [1] and symbolic calculation [4,22]. Each of these techniques has advantages and disadvantages, however none of them can be used to obtain exact results in the general case (cf. Section 2.1).

Exact real (or arbitrary precision) arithmetic is an approach to real number computation that lets the user specify the accuracy to which results are to be computed. The desired precision is accounted for in each step of the computation, even when this means that some intermediate results have to be calculated to a very high accuracy. Arbitrary precision arithmetic is usually considerably slower than more conventional approaches, however its generality makes it an important theoretical tool. This is why it is one of the main subjects of this project.

An interesting property of exact real arithmetic is that redundancy plays an important role for its representations and that for instance the standard decimal expansion cannot be used. This is because producing the first digit of the sum of two numbers given in the decimal expansion sometimes requires an infinite amount of input (see Section 2.2.1). Using a (sufficiently) redundant representation allows algorithms to make a guess how the input might continue and correct this guess later in case it turns out to be wrong. An important representation that uses redundancy to this end is the signed binary expansion which forms the basis for the arithmetic operations of this project.

1.2 Coinductive Proof

Many algorithms for exact real arithmetic have been proposed but their correctness is rarely proved formally (argued for instance in [3]). This is quite surprising – after all, the goal of providing results that are *known* to be correct can only be achieved through proof. Moreover, those proofs that are given in literature use a large variety of different techniques that are usually only applicable to the respective approach or algorithm.

One of the most important aspects of exact real arithmetic is the inherent infiniteness of the real numbers. Consider for example the mathematical constant π which can be represented using the well-known decimal expansion as

$$\pi = 3.14159265\ldots$$

Ignoring the decimal point for the moment, this corresponds to the infinite sequence of digits $(3, 1, 4, 1, 5, 9, 2, 6, 5, \ldots)$. Any implementation of exact real arithmetic can only store a finite number of these digits at a time, however the underlying mathematical concept of π must be represented in such a way (typically an algorithm) that arbitrarily close approximations to it can be computed. This tension between finite numerical representations and the underlying infinite mathematical objects is a key characteristic of arbitrary precision arithmetic.

In recent years, the inherent infiniteness of the objects involved has been exploited in work on exact real arithmetic using a field called *universal coalgebra*. Universal coalgebra provides a formal mathematical framework for studying infinite data types and comes with associated definition and proof principles called *corecursion* and *coinduction* that can be used similarly to their algebraic namesakes. In the context of arbitrary precision arithmetic, universal coalgebra makes it possible to reason about infinite approximations to the underlying mathematical objects (in the example above, the whole sequence $(3, 1, 4, 1, \ldots)$) and thus to avoid the distinction between finite representations and infinite concepts.

Universal coalgebra models infinite structures by viewing them as states of *systems* which have a set of possible states, properties that can be observed in each state and actions that result in state transitions. In this way, for instance an infinite stream $(a_1, a_2, \ldots)$ of elements can be modelled as state 1 of the system whose possible states are the natural numbers $\mathbb{N}$, whose (one) property that can be observed has value a_n in state n and whose (one) action that takes it to the next state is the successor function $n \mapsto n + 1$:

$$\begin{tikzcd} 1 \arrow[r, "a_1"] & 2 \arrow[r, "a_2"] & 3 \arrow[r] & \cdots \end{tikzcd}$$

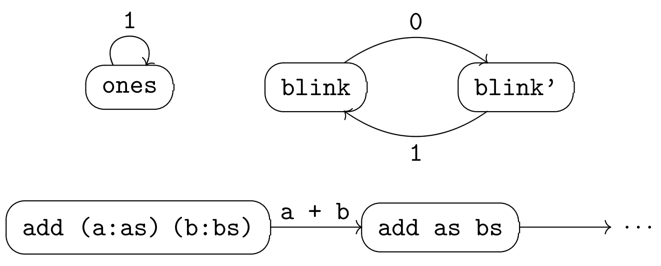
Each system described in such a way is formally called a *coalgebra* and specifying a system amounts to giving a *coinductive definition*.

In the context of a coinductive definition, corecursion is one way of specifying the observations and effects of actions for a particular state. Consider for example the following corecursive definitions in the functional programming language Haskell:

```
ones, blink, blink' :: [ Int ]
ones   = 1 : ones
blink  = 0 : blink'
blink' = 1 : blink

add :: [ Int ] -> [ Int ] -> [ Int ]
add (a : as) (b : bs) = (a + b) : add as bs
```

This specifies for instance that in state `ones`, the observation one can make is the digit `1` and that the next state is again `ones`. Similarly, for the state `add (a : as) (b : bs)`, one can observe the value of `(a + b)` while the next state is `add as bs`:



Coinduction in universal coalgebra exploits the fact that, in certain systems, the equality of two states (and thus of the two infinite structures they represent) can be shown by proving that they are *bisimilar.* Intuitively speaking, two states s_1 and s_2 are bisimilar if they are observation-equivalent, that is, if any sequence of actions and making of observations starting from s_1 leads to the same outcomes of observations as when starting from s_2. If for example s_1 and s_2 represent streams, then this means that one has to show that they have the same head and that the states corresponding to their tails are again bisimilar.

Unfortunately, due to the – for computability reasons inevitable – redundancy of number representations involved, being able to show the equality of two numerals alone is not sufficient in the context of exact real arithmetic. Following the approach recently proposed by Berger in [3], we therefore use the more general (and historically older) *set-theoretic* coinduction.

Suppose we want to show that `(add blink blink') = ones` in the example above. This is equivalent to the statement that all elements of the two streams are equal, which in turn is the same as saying that, for all $n \in \mathbb{N}$, the first $2n$ elements of `(add blink blink')` are equal to the first $2n$ elements of `ones`. If we write $\texttt{Int}^\omega$ for the set of all `Int`-streams, f^n for the n-fold application of a function f and define the operator $\mathrm{O} : \wp(\texttt{Int}^\omega \times \texttt{Int}^\omega) \to \wp(\texttt{Int}^\omega \times \texttt{Int}^\omega)$ by

$$\mathrm{O}(R) = \{(a : a' : \alpha, a : a' : \beta \mid a, a' \in \texttt{Int}, (\alpha, \beta) \in R\},$$

then this means that we have to show that

$$(\texttt{add blink blink'}, \texttt{ones}) \in \bigcap_{n \in \mathbb{N}} \mathrm{O}^n(\texttt{Int}^\omega \times \texttt{Int}^\omega).$$

However, by an application of the Knaster-Tarski fixpoint theorem (cf. Section 2.4.2), $\bigcap_{n \in \mathbb{N}} \mathrm{O}^n(\texttt{Int}^\omega \times \texttt{Int}^\omega)$ is the greatest (in terms of set-inclusion, $\subseteq$) fixpoint of O and it is sufficient to show

$$R = \{(\texttt{add blink blink'}, \texttt{ones})\} \subseteq \mathrm{O}(R).$$

This is the *set-theoretic coinduction principle.*

Using the above reformulation, the proof is now as follows: We have

```
add blink blink' = add (0 : blink') (1 : blink)
                 = 1 : add blink' blink
                 = 1 : add (1 : blink) (0 : blink')
                 = 1 : 1 : add blink blink'
```

and clearly

```
ones = 1 : 1 : ones.
```

This shows that $R \subseteq \mathrm{O}(R)$ and hence, by set-theoretic coinduction, that

```
add blink blink' = ones.
```

Despite the fact that set-theoretic coinduction does not reside in the field of universal coalgebra, the steps it involves can often be interpreted in terms of observations, actions and states. For this reason, set-theoretic coinduction may greatly benefit from coalgebraic coinductive definitions of the objects involved.

The aim of this project is to explore the combination of coalgebraic coinductive definitions and set-theoretic coinduction in the context of exact real arithmetic. To this end, we coinductively define arithmetic operations that compute the sum and linear affine transformations over $\mathbb{Q}$ of real numbers given in the signed binary exponent-mantissa representation and prove their correctness using set-theoretic coinduction.

1.3 Contributions

The main contributions of this project can be summarized as follows:

- We propose a general strategy for studying exact arithmetic operations on the signed binary exponent-mantissa representation that combines coalgebraic-coinductive definitions and set-theoric coinduction. This strategy is similar to that used in [3], however it explains the ensuing definitions from a coalgebraic perspective and can be used for operations on the real line rather than just those on the unit interval.

- We explore the proposed strategy (and thus the combination of coalgebraic coinductive definitions and set-theoretic coinduction) by using it to obtain operations that compute the sum and liner affine transformations over $\mathbb{Q}$ and prove their correctness.

- We give a Haskell implementation of the operations thus obtained and use this implementation for a brief comparison with related algorithms from literature.

Chapter 2

Background

This chapter provides more detail on some of the relevant technical background only touched upon in the previous chapter.

2.1 Alternatives to Floating Point Arithmetic

The Muller-sequence described in the previous chapter shows that there are cases in which floating point arithmetic can not be used. This section briefly describes some of its alternatives, including a slightly more exhaustive account of exact real arithmetic than that given in the introduction. A more detailed overview of these alternatives can be found in [15].

Floating point arithmetic with error analysis Floating point arithmetic with error analysis is similar to floating point arithmetic except that it keeps track of possible rounding errors that might have been introduced during the computation. It produces two floating point numbers: the first is the result obtained using normal floating point arithmetic while the second gives the range about this point that the exact result is guaranteed to be in if rounding errors are taken into account. In the example given in the introduction, the bound on the error would be very large. Knuth gives a theoretical model for floating point error analysis in [11].

Interval arithmetic Interval Arithmetic can be seen as a generalization of floating point arithmetic with error analysis. Instead of using two floating point numbers to specify the center and half length of the interval the exact result is guaranteed to be in, it uses two numbers of any finitely representable subset of the reals (eg. the rational numbers) to specify the lower and upper bounds of the interval, respectively. Similarly to floating point arithmetic with error analysis, each calculation is performed on both bounds, with strict downwards and upwards rounding respectively if the result on a bound cannot be represented exactly. Interval arithmetic is very useful, however it does not make the calculations any more exact. An accessible introduction to interval arithmetic and how it can be implemented using IEEE floating point arithmetic is given in [9].

Stochastic rounding Unlike ordinary floating point arithmetic, which rounds either always upwards or always downwards in case the result of a computation falls exactly between two floating point numbers, stochastic rounding chooses by tossing a (metaphorical) fair coin. The desired computation is repeated several times and the true result is then estimated using probability theory. While stochastic rounding cannot guarantee the accuracy of the result in a fashion similar to floating point arithmetic with error analysis, it will in general give more exact results. Moreover, it allows to obtain probabilistic information about the reliability of its calculations. An implementation of stochastic rounding is described in [20]. When used to compute the first 25 terms of the Muller-sequence, this implementation does not give more accurate results than ordinary floating point arithmetic. However, it does correctly detect the numerical instabilities and warns the user that the results are not guaranteed [1].

Symbolic calculation Symbolic approaches represent real numbers as expressions consisting of function symbols, variables and constants. Calculations are performed by simplifying such expressions rather than by manipulating numbers. It is important to note that the number to be computed is thus represented exactly at each stage of the calculation.

The problem with symbolic calculation is that the simplifaction of arbitrary mathematical expressions is very difficult and that there are many cases where it cannot be done at all. In these cases, the expression has to be evaluated numerically in order to be usable, which is why symbolic approaches are rarely used on their own. This can be seen in the two mathematics packages `Maple` [4] and `Mathematica` [22] that are known for the strength of their symbolic engines but offer support for numeric calculations as well.

Exact real arithmetic As explained in the introduction, exact real arithmetic guarantees the correctness of its results up to some user-specified precision. Since this may require calculating some intermediate results to a very high accuracy, implementations of exact real arithmetic have to be able to handle arbitrarily large number representations and are therefore often considerably slower than more conventional approaches. In return however, arbitrary precision arithmetic solves the problems of inaccuracy and uncertainty associated with interval arithmetic and stochastic rounding and can be used in many cases in which a symbolic approach would not be appropriate. Various approaches to exact real arithmetic can for instance be found in [3, 6, 7, 14, 15].

We see that, of all these approaches, only exact real arithmetic can be used to obtain arbitrarily precise results for the general case. As mentioned in the introduction, this is why exact real arithmetic holds an important place among approaches to real number computation.

2.2 Representations in Exact Real Arithmetic

It is clear that in any approach to exact real arithmetic the choice of representation largely determines how operations can be defined and reasoned about. As we will see now, this impact goes even further in that there are representations for which some operations cannot (reasonably) be defined at all.

2.2.1 The Failure of the Standard Decimal Expansion

Consider adding the two numbers $\frac{1}{3} = 0.333\ldots$ and $\frac{2}{3} = 0.666\ldots$ using the standard decimal expansion (this is a well-known example and can for instance be found in [6, 15]). After having read the first digit after the decimal point from both expansions, we know that they lie in the intervals $[0.3, 0.4]$ and $[0.6, 0.7]$, respectively. This means that their sum is contained in the interval $[0.9, 1.1]$ and so we do not know at this stage whether the first digit of the result should be a 0 or a 1. This problem continues: all that reading any finite number of digits from the inputs can tell us is that they lie in the invervals $[0.3\ldots33, 0.3\ldots34]$ and $[0.6\ldots66, 0.6\ldots67]$, and thus that their sum is contained in $[0.9\ldots99, 1.0\ldots01]$, whose bounds are strictly less and greater than 1, respectively. We would therefore never be able to output even the first digit of the result.

The way to get around this problem is to introduce some form of redundancy into the representation (see eg. [2]). This allows us in cases as above to make a guess how the input might continue and then undo this guess later in case it turns out to be wrong. The next section describes an important class of representations that use redundancy to this end.

2.2.2 Signed Digit Representations

One of the most widely-studied represenations (see eg. [2,6,11]) for real numbers is the signed digit representation in a given integer base B. Instead of, as in the standard decimal expansion, only allowing positive digits $\{0, 1, \ldots, B-1\}$ to occur in the representation, one also includes negative numbers so that the set of possible digits becomes $\{-B+1, -B+2, \ldots, -1, 0, 1, \ldots, B-1\}$. For elements b_i of this set, the sequence $(b_1, b_2, \ldots)$ then represents the real number

$$\sum_{i=1}^{\infty} b_i B^{-i} \tag{2.1}$$

It should be quite clear from this formula how negative digits can be used to make corrections to previous output.

Consider again the example of adding $\frac{1}{3}$ and $\frac{2}{3}$. These two numbers can be represented in the *signed* decimal expansion by the streams

$$\frac{1}{3} = 0.333\cdots \sim (0, 3, 3, 3, \ldots)$$
$$\frac{2}{3} = 0.666\cdots \sim (0, 6, 6, 6, \ldots)$$

After having read $(0, 3)$ from the first input, we know it is greater than or equal to 0.2 (since the sequence could continue $-9, -9, -9, \ldots$) and less than or equal to 0.4. Similarly, we know that the second input lies in the interval $[0.5, 0.7]$ so that the sum of the two is in the range $[0.7, 1.1]$. Even though this interval is larger than the one we obtained when using the ordinary decimal expansion, it is now safe to output 1 because the stream starting $(1, \ldots)$ can represent any number between 0 and 2.

A note on the word *stream* Streams are simply infinite sequences. We will use the term *sequence* to refer to both finite and infinite lists of elements, however it will always be made clear which of the two cases we are talking about.

The Signed Binary Expansion

The signed binary expansion is the simplest and most widely-used signed digit representation. It represents the case where $B = 2$ and thus uses digits from the set $\{-1, 0, 1\}$ to identify real numbers in the interval $[-1, 1]$. The signed binary expansion is the representation on which the arithmetic operations of this project are defined.

Interpreting numerals After having read the first digit d_1 of a signed binary numeral, the remaining terms in Equation (2.1) can give a contribution of magnitude at most $\frac{1}{2}$. As visualized by Figure 2.1 (seen similar in [15]), this allows us to conclude that the value of the numeral lies in the interval $[\frac{d_1}{2} - \frac{1}{2}, \frac{d_1}{2} + \frac{1}{2}]$.

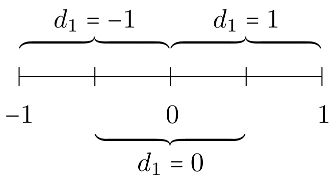


Figure 2.1: Interval of a signed binary numeral with first digit d_1

Suppose that we next read the second digit d_2. By again referring to Equation (2.1) and a reasoning similar to the above, we can conclude that the numeral lies in the interval $\left[\frac{d_1}{2} + \frac{d_2}{4} - \frac{1}{4}, \frac{d_1}{2} + \frac{d_2}{4} + \frac{1}{4}\right]$. This is exemplarily shown for the case $d_1 = 1$ in Figure 2.2.

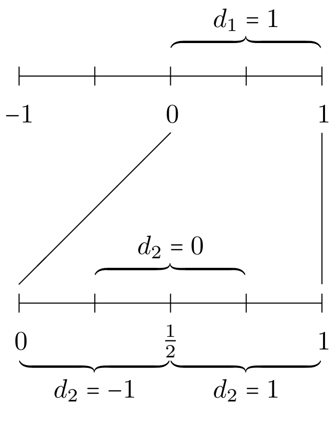


Figure 2.2: Interval of a signed binary numeral of the form $(1, d_2, \dots)$

Note the symmetry! Again, we had an interval of half-width 2^{-n}, where n was the number of digits read thus far, and reading the next digit (in this case d_2) told us whether the value of the numeral lies in the lower, middle or upper half of this interval. Since this can be shown to hold for any sequence of digits, reading a signed binary numeral can be seen as an iterative selection of sub-intervals with exponentially decreasing length.

Reference Intervals (or: The Decimal Point)

A disadvantage of the signed digit representation in base B is that it can only be used to represent real numbers in the interval $[-B+1, B-1]$. In the decimal expansion, this problem is overcome using the decimal (or, for bases other than 10 the *radix*) point, whose role we have silently ignored up until now.

What the decimal point does is that it specifies the magnitude of the number that is to be represented. For instance, in the decimal expansion of $\pi = 3.141\dots$, the position of the decimal point tells us that the normal formula

$$\sum_{i=1}^{\infty} b_i 10^{-i}$$

has to be multiplied by an additional factor of 10^1 to obtain the required result.

Instead of extending the set of allowed digits $\{-B+1, \dots, B-1\}$ to explicitly include the decimal point, it is common to use an exponent-mantissa representation, which simply specifies the exponent of the scaling constant. A real number r is thus represented by its exponent $a \in \mathbb{N}$ (or even $a \in \mathbb{Z}$) and its mantissa $(b_1, b_2, \dots)$ where $b_i \in \{-B+1, \dots, B-1\}$ and

$$r = B^a \sum_{i=1}^{\infty} b_i B^{-i}$$

This allows us to use signed digit representations for specifying any real number, even if it is outside the normally representable interval $[-B+1, B-1]$.

2.2.3 Other Representations

This section briefly describes some alternative representations for the real numbers. An important feature of a representation for the real numbers is whether it is *incremental*. A representation is said to be incremental if a numeral that represents a real number to some accuracy can be reused to calculate a more accurate approximation to that number. The signed digit representation described above is incremental since all we have to do in order to get a better approximation for a real number is to calculate more digits of its expansion. We will indicate which of the representations that are mentioned here are incremental and which are not.

Linear Fractional Transformations

A *one-dimensional linear fractional transformation (1-LFT)* or *Möbius transformation* is a function $L : \mathbb{C} \to \mathbb{C}$ of the form

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

where a, b, c and d are arbitrary but fixed real or complex numbers. Similarly, a function $T : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ of the form

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

where a, b, c, d, e, f, g and h are fixed real or complex numbers is called a *two-dimensional linear fractional transformation (2-LFT)*.

Linear fractional transformations can be written as matrices: The 1-LFT above can be represented by the matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

The application of this matrix to an argument can then naturally defined via its corresponding LFT:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}(x) = \frac{ax + c}{bx + d}$$

It can be shown that the composition of two 1-LFTs $L_1(x)$ and $L_2(x)$ that are represented by matrices M_1 and M_2, respectively, is again a 1-LFT and can be computed using matrix multiplication:

$$(L_1 \circ L_2)(x) = (M_1 \cdot M_2)(x)$$

Finally, it is possible to represent any real number r as the limit of an infinite composition of LFTs with integer parameters applied to the interval $[-1, 1]$:

$$r = \lim_{n \to \infty} ((M_1 \cdot M_2 \cdot \ldots \cdot M_n)([-1, 1]))$$

If these LFTs are constructed in the right way, one can guarantee that each element of the sequence is a better approximation to r than the previous ones. This implies that the linear fractional transformation representation is incremental.

The main advantage of LFTs is that numerical operations can be defined at a high level of abstraction and that many well-known Taylor series and continued fraction expansions can be directly translated into an infinite product of LFTs. For an accessible introduction to LFTs see for instance [6].

Continued Fractions

The (generalised) continued fraction representation of a real number r is parameterized by two integer sequences $(a_n)_{n \geq 0}$ and $(b_n)_{n \geq 0}$ such that

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

Many important mathematical constants have surprisingly simple continued fractions representations. For instance, the mathematical constant π by can be represented by the sequences

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

The continued fraction representation is incremental. Vuillemin proposed the use of continued fractions with constant $a_n = 1$ for all $n \in \mathbb{N}$ in [21] and defined algorithms for the standard arithmetic operations as well as some transcendental functions. His work is partially based on results originally proposed by Gosper in [8].

Dyadic Rational Streams

A *dyadic rational* is a rational number whose denominator is a power of two, i.e. a number of the form

$$\frac{a}{2^b}$$

where a is an integer and b is a natural number. Similarly to the signed binary expansion, a stream $(d_1, d_2, \dots)$ of dyadic rationals in the interval $[-1, 1]$, can be used to represent a real number $r \in [-1, 1]$ via the formula

$$r = \sum_{i=1}^{\infty} d_i 2^{-i}.$$

Observe that this representation has the same rate of convergence as the signed binary expansion. However, each dyadic digit d_i can incorporate significantly more information than a signed binary digit. This can greatly simplify certain algorithms but unfortunately often leads to a phenomenon called *digit swell* in which the size of the digits increases so rapidly that the performance is destroyed. Plume gives an implementation of the standard arithmetic operations for the signed binary and the dyadic stream representations and discusses such issues in [15].

Nested Intervals

The nested interval representation describes a real number r by an infinite sequence of nested closed intervals

$$[a_1, b_1] \supseteq [a_2, b_2] \supseteq [a_3, b_3] \supseteq \dots .$$

The progressing elements of the sequence represent better and better approximations to r. In order for this to be a valid representation, it must have the property that $|a_n - b_n| \to 0$ as $n \to \infty$ and that both a_n and b_n converge to r.

The endpoints of the intervals are elements of a finitely representable subset of the reals, typically the rational numbers. Computation using nested intervals is performed by calculating further elements of the sequence and thus better approximations to r. This implies that this representation is incremental.

An implementation of nested intervals for the programming language PCF is given in [7]. This reference uses rational numbers to represent the endpoints of the intervals and uses topological and domain theoretic arguments to develop the theory behind the approach.

2.3 Computability Issues

Without getting bogged down in technical details of the various formalisations of computability, there are a few key issues we would like to mention in order to give the reader a feeling for the setting exact real arithmetic resides in. All results that are presented here are described in a very informal and general manner, even though the statements made by the given references are much more precise and confined. However, the ideas that underlie them are of such generality and intuitive truth that we hope that the reader will bear with us in taking this step.

2.3.1 Computable Numbers

Turing showed in his key paper [18,19] that not every real number that is definable is computable and that only a countable subset of the reals is computable. Fortunately, the numbers and operations we encounter and use every day all turn out to be in this set so that this limitation rarely affects us. Nevertheless, computability is an important concept that greatly influences the way our operations are defined, even if we do not always make this explicit.

An interesting result in this context is that there cannot be a general algorithm to determine whether two computable real numbers are equal [16]. The intuitive explanation for this is that an infinite amount of data would have to be read, which is of course impossible. Although our operations do not directly need to compare two real numbers, they are constrained in a similar way: Even though it would make the algorithms simpler and would not require redundancy in the representation, operations for exact real arithmetic cannot work from right to left, ie. in the direction in which the significance of the digits increases, since this would require reading an infinite amount of data.

2.4 Coalgebra and Coinduction

Finally, in this section, we make precise how we are going to use coinduction for our definitions and proofs.

We will use the following notation: If A is a set, then we write $\mathsf{id}_A : A \to A$ for the identity function on A and A^ω for the set of all streams of elements of A. Streams themselves are denoted by greek letters or expressions of the form (a_n) and their elements are written as a_1, a_2 etc. The result of prepending an element a to a stream α will be written as $a : \alpha$. If $f : A_1 \to B_1$ and $g : A_2 \to B_2$ are functions, then $f \times g : (A_1 \times A_2) \to (B_1 \times B_2)$ denotes the function defined by $(f \times g)(a_1, a_2) = (f(a_1), g(a_2))$. For two functions $f : A \to B$ and $g : A \to C$ with the same domain, $\langle f, g\rangle : A \to B \times C$ is defined by $\langle f, g\rangle(a) = (f(a), g(a))$. Finally, for any function $f : A \to A$, $f^n : A \to A$ denotes its nth iteration, that is

$$f^0(x) = x \qquad \text{and} \qquad f^{n+1}(x) = f\left(f^n(x)\right).$$

Many of the results presented here can be found in a similar or more general form in [10] and [17]. For an application of coinduction to exact real arithmetic see [3].

As briefly outlined in the introduction, universal coalgebra can be used to study infinite data types, which includes streams but also more complex structures such as infinite trees. Since the coalgebraically interesting part of our representation lies in the stream of digits however, we do not need the full theory behind universal coalgebra but can rather restrict ourselves to the following simple case:

Definition 2.1. *Let A be a set. An A-*stream coalgebra *is a pair (X, γ) where*

1. *X is a set*
2. *$\gamma : X \to A \times X$ is a function*

Any stream $(a_n)_{n\in\mathbb{N}}$ of elements in a set A can be modelled as an A-stream coalgebra by taking $X = \mathbb{N}$ and defining γ by

$$\gamma(n) = (a_n, n+1).$$

The following important definition captures one way in which A-stream coalgebras can be related:

Definition 2.2. *Let (X,γ) and (Y,δ) be two A-stream coalgebras. A function $f : X \to Y$ is called an* A-stream homomorphism *from (X,γ) to (Y,δ) iff $(\mathsf{id}_A \times f) \circ \gamma = \delta \circ f$, that is, the following diagram commutes:*

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\gamma"'] & Y \arrow[d, "\delta"] \\
A \times X \arrow[r, "\mathsf{id}_A \times f"'] & A \times Y
\end{tikzcd}$$

This immediately gives rise to

Definition 2.3. *An A-stream homomorphism f is called an* A-stream isomorphism *iff its inverse exists and is also an A-stream homomorphism.*

The most trivial example of an A-stream isomorphism of a coalgebra (X,γ) is $\mathsf{id}_X : X \to X$, the identity map on X: it is clearly bijective and we have

$$(\mathsf{id}_A \times \mathsf{id}_X) \circ \gamma = \gamma = \gamma \circ \mathsf{id}_X.$$

This shows that id_X is an A-stream isomorphism.

The composition of two A-stream homomorphisms is again a homomorphism:

Proposition 2.4. *Let (X,γ), (Y,δ) and (Z,ϵ) be A-stream coalgebras and $f : X \to Y$ and $g : Y \to Z$ be homomorphisms. Then $g \circ f : X \to Z$ is an A-stream homomorphism from (X,γ) to (Z,ϵ).*

Proof. We have

$$(\mathsf{id}_A \times (g \circ f)) \circ \gamma = (\mathsf{id}_A \times g) \circ (\mathsf{id}_A \times f) \circ \gamma = (\mathsf{id}_A \times g) \circ \delta \circ f = \epsilon \circ (g \circ f)$$

This shows that $g \circ f$ is an A-stream homomorphism. □

A coalgebraic concept that turns out to be extremely useful is that of *finality*:

Definition 2.5. *An A-stream coalgebra (X,γ) is called* final *iff for any A-stream coalgebra (Y,δ) there exists a unique homomorphism $f : Y \to X$.*

Proposition 2.6. *Final A-stream coalgebras are unique, up to isomorphism: If (X,γ) and (Y,δ) are final A-stream coalgebras then there is a unique isomorphism $f : X \to Y$ of A-stream coalgebras.*

Proof. If (X,γ) and (Y,δ) are final A-stream coalgebras, then there are unique homomorphisms $f : X \to Y$ and $g : Y \to X$. By Proposition 2.4, their composition $g \circ f : X \to X$ is again a homomorphism. Since $\mathsf{id}_X : X \to X$ is a homomorphism, too, we have $g \circ f = \mathsf{id}_X$ by the uniqueness part of finality. A similar argument yields that $f \circ g = \mathsf{id}_Y$. This shows that $f^{-1} = g$ exists and, since it is a homomorphism, that f is an A-stream isomorphism. □

Finality allows us to justify the claim that our definition of A-stream coalgebras captures the concept of a stream of elements of a set A:

Proposition 2.7. *Let A be a set. If the functions* $\mathsf{hd} : A^\omega \to A$ *and* $\mathsf{tl} : A^\omega \to A^\omega$ *are defined by*

$$\mathsf{hd}((a_n)) = a_1 \qquad \textit{and} \qquad \mathsf{tl}((a_n)) = (a_2, a_3, \dots),$$

then the A-stream coalgebra $(A^\omega, \langle \mathsf{hd}, \mathsf{tl}\rangle)$ is final.

Proof. Let $(U, \langle \mathsf{value}, \mathsf{next}\rangle)$ be an arbitrary A-stream coalgebra. Define the function $f : U \to A^\omega$ for $u \in U$ and $n \in \mathbb{N}$ by

$$(f(u))_n = \mathsf{value}\,(\mathsf{next}^n(u))\,.$$

Then f is a homomorphism:

$$(\mathsf{id}_A \times f) \circ \langle \mathsf{value}, \mathsf{next}\rangle = \langle \mathsf{value}, f \circ \mathsf{next}\rangle = \langle \mathsf{value}, \mathsf{tl} \circ f\rangle = \langle \mathsf{hd}, \mathsf{tl}\rangle \circ f$$

Uniqueness can now easily be shown by noting that $\langle \mathsf{hd}, \mathsf{tl}\rangle$ is a bijection. □

2.4.1 Coalgebraic Coinduction

The existence part of finality can be exploited to coinductively define functions. Consider the function $\mathsf{merge} : A^\omega \times A^\omega \to A^\omega$ which merges two streams:

$$\mathsf{merge}((a_n), (b_n)) = (a_1, b_1, a_2, b_2, \dots)$$

Instead of specifying merge directly, we can take it to be the unique homomorphism that arises by the finality of $(A^\omega, \langle \mathsf{hd}, \mathsf{tl}\rangle)$ in the following diagram:

$$\begin{tikzcd}
A^\omega \times A^\omega \arrow[r, "\mathsf{merge}"] \arrow[d, "{\begin{matrix}(\alpha,\beta)\\ \downarrow \\ (\mathsf{hd}(\alpha),(\beta,\mathsf{tl}(\alpha)))\end{matrix}}"'] & A^\omega \arrow[d, "{\langle \mathsf{hd},\mathsf{tl}\rangle}"] \\
A \times (A^\omega \times A^\omega) \arrow[r, "\mathsf{id}_A \times \mathsf{merge}"'] & A \times A^\omega
\end{tikzcd}$$

This use of the existence part of finality to define a function is referred to as the *coinductive definition principle.*

Observe that, by the commutativity of the above diagram, we have

$$\mathsf{hd}(\mathsf{merge}(\alpha, \beta)) = \mathsf{hd}(\alpha) \qquad \text{and} \qquad \mathsf{tl}(\mathsf{merge}(\alpha, \beta)) = \mathsf{merge}(\beta, \mathsf{tl}(\alpha)),$$

that is,

$$\mathsf{merge}(\alpha, \beta) = \mathsf{hd}(\alpha) : \mathsf{merge}(\beta, \mathsf{tl}(\alpha)).$$

This is a *corecursive* definition of merge: Instead of, as in a recursive definition, descending on the argument, we ascend on the result by filling in the observations (in this case the head) one can make about it.

As another example, consider the function $\mathsf{odd} : A^\omega \to A^\omega$ which can be defined coinductively via the function $\langle \mathsf{hd}, \mathsf{tl}^2\rangle$ in the following diagram:

$$\begin{tikzcd}
A^\omega \arrow[r, "\mathsf{odd}"] \arrow[d, "{\langle \mathsf{hd},\mathsf{tl}^2\rangle}"'] & A^\omega \arrow[d, "{\langle \mathsf{hd},\mathsf{tl}\rangle}"] \\
A \times A^\omega \arrow[r, "\mathsf{id}_A \times \mathsf{odd}"'] & A \times A^\omega
\end{tikzcd}$$

Again, the commutativity of the diagram implies

$$\mathsf{hd}(\mathsf{odd}(\alpha)) = \mathsf{hd}(\alpha) \qquad \text{and} \qquad \mathsf{tl}(\mathsf{odd}(\alpha)) = \mathsf{odd}(\mathsf{tl}(\mathsf{tl}(\alpha))).$$

In the following, we will also use odd's counterpart $\mathsf{even} : A^\omega \to A^\omega$ which we define by

$$\mathsf{even}(\alpha) = \mathsf{odd}(\mathsf{tl}(\alpha)).$$

Suppose we want to prove the equality $\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)) = \alpha$ for all $\alpha \in A^\omega$. Since we know that $\mathsf{id}_{A^\omega} : A^\omega \to A^\omega$ is a homomorphism, and since the finality of $(A^\omega, \langle \mathsf{hd}, \mathsf{tl} \rangle)$ tells us that this homomorphism is unique, it is enough to show that $\mathsf{merge} \circ \langle \mathsf{odd}, \mathsf{even} \rangle : A^\omega \to A^\omega$ is a homomorphism to deduce that $\mathsf{merge} \circ \langle \mathsf{odd}, \mathsf{even} \rangle = \mathsf{id}_{A^\omega}$. This is an example of a coalgebraic proof *by coinduction.*

In order to prove that $\mathsf{merge} \circ \langle \mathsf{odd}, \mathsf{even} \rangle$ is a homomorphism, we have to show

$$\langle \mathsf{hd}, \mathsf{tl} \rangle \circ (\mathsf{merge} \circ \langle \mathsf{odd}, \mathsf{even} \rangle) = (\mathsf{id}_A \times (\mathsf{merge} \circ \langle \mathsf{odd}, \mathsf{even} \rangle)) \circ \langle \mathsf{hd}, \mathsf{tl} \rangle.$$

This follows from the two chains of equalities

$$\begin{aligned} \mathsf{hd}(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha))) &= \mathsf{hd}(\mathsf{odd}(\alpha)) \\ &= \mathsf{hd}(\alpha) \end{aligned}$$

and

$$\begin{aligned} \mathsf{tl}(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha))) &= \mathsf{merge}(\mathsf{even}(\alpha), \mathsf{tl}(\mathsf{odd}(\alpha))) \\ &= \mathsf{merge}(\mathsf{even}(\alpha), \mathsf{odd}(\mathsf{tl}(\mathsf{tl}(\alpha)))) \\ &= \mathsf{merge}(\mathsf{odd}(\mathsf{tl}(\alpha)), \mathsf{even}(\mathsf{tl}(\alpha))) \\ &= (\mathsf{merge} \circ \langle \mathsf{odd}, \mathsf{even} \rangle)(\mathsf{tl}(\alpha)). \end{aligned}$$

Hence, by coinduction, $\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)) = \alpha$ for all $\alpha \in A^\omega$.

As a concluding remark, recall that only the finality of $(A^\omega, \langle \mathsf{hd}, \mathsf{tl} \rangle)$ allowed us to use the coinductive definition and proof principles in the above examples. This is one of the main reasons why finality plays such a key role in the field of coalgebra.

2.4.2 Set-theoretic Coinduction

Because of the – for computability reasons inevitable – redundancy of exact real number representations, the coinductive proof principle outlined in the previous section is often too restricted to express equality of real numbers given as streams (argued for instance in [3]). The more general (and historically older) set-theoretic form of coinduction exploits the fact that every monotone set operator has a greatest fixpoint. Since this kind of coinduction will be used to prove the main results of the next chapter, we here introduce its underlying principles and show how it can be used to prove the merge identity from the previous section. For a thorough comparison of coalgebraic and set-theoretic coinduction see for instance [12].

Background

Recall the following standard definitions and results:

Definition 2.8. *A* partially ordered set *is a set P together with a binary relation $\leq$ on P that satisfies, for all $a, b, c, \in P$*

- *$a \leq a$ (reflexivity)*
- *$a \leq b$ and $b \leq a \Rightarrow a = b$ (antisymmetry)*
- *$a \leq b$ and $b \leq c \Rightarrow a \leq c$ (transitivity)*

Definition 2.9. *A* complete lattice *is a partially ordered set in which all subsets have both a supremum and an infimum.*

Theorem 2.10 (Knaster-Tarski)**.** *If $(L, \leq)$ is a complete lattice and $m : L \to L$ is a monotone function, then the set of all fixpoints of m in L is also a complete lattice.*

Corollary 2.11. *If $(L, \leq)$ is a complete lattice then any monotone function $m : L \to L$ has a least fixpoint $LFP(m)$ and a greatest fixpoint $GFP(m)$ given by*

- *$GFP(m) = \sup\ \{x \in L \mid x \leq m(x)\}$*
- *$LFP(m) = \inf\ \{x \in L \mid x \geq m(x)\}$*

Moreover, for any $l \in L$,

- *if $m(l) \leq l$ then $LFP(m) \leq l$ and*
- *if $l \leq m(l)$ then $l \leq GFP(m)$.*

These results can be used as follows: Let X, Y be sets, $R, M \subseteq X \times Y$ be binary relations on X and Y and suppose we want to show that $(x, y) \in M$ for all $(x, y) \in R$. If we manage to find a monotone operator $\mathrm{O} : \wp(X \times Y) \to \wp(X \times Y)$ whose greatest fixpoint in the complete lattice $(\wp(X \times Y), \subseteq)$ is M, then it suffices to show $R \subseteq \mathrm{O}(R)$ to deduce that $R \subseteq M$ by the last part of the above corollary. This is the *set-theoretic coinduction principle.*

An Example Proof

Recall the merge identity from the previous section: For all $\alpha \in A^\omega$,

$$\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)) = \alpha. \tag{2.2}$$

In order to prove this result by set-theoretic coinduction, we work in the complete lattice $(\wp(A^\omega \times A^\omega), \subseteq)$ and define the operator

$$\mathrm{O} : \wp(A^\omega \times A^\omega) \to \wp(A^\omega \times A^\omega)$$
$$\mathrm{O}(R) = \{(a : a' : \alpha, a : a' : \beta) \mid a, a' \in A, (\alpha, \beta) \in R\}.$$

O is clearly monotone and we claim that its greatest fixpoint $GFP(\mathrm{O})$ is given by

$$M = \{(\alpha, \alpha) \mid \alpha \in A^\omega\}.$$

To see this, let $(\alpha, \beta) \in \mathit{GFP}(\mathrm{O})$. We want to show that $(\alpha, \beta) \in M$ which is equivalent to proving that $\alpha = \beta$. Since $\mathit{GFP}(\mathrm{O})$ is a fixpoint of O, we have $\mathit{GFP}(\mathrm{O}) = \mathrm{O}(\mathit{GFP}(\mathrm{O}))$ and thus $(\alpha, \beta) \in \mathrm{O}(\mathit{GFP}(\mathrm{O}))$. This means by the definition of O that the first two digits of α and β are equal and that $\big(\mathsf{tl}^2(\alpha), \mathsf{tl}^2(\beta)\big) \in \mathit{GFP}(\mathrm{O})$. Repeating the same argument for $\big(\mathsf{tl}^2(\alpha), \mathsf{tl}^2(\beta)\big)$, then for $\big(\mathsf{tl}^4(\alpha), \mathsf{tl}^4(\beta)\big)$ etc. shows that all digits of α and β are equal and thus that $\alpha = \beta$. Hence $\mathit{GFP}(\mathrm{O}) \subseteq M$. Since clearly $M \subseteq \mathrm{O}(M)$ and thus by the last part of the above corollary $M \subseteq \mathit{GFP}(\mathrm{O})$, this shows that $\mathit{GFP}(\mathrm{O}) = M$.

The next step is to define the relation $R \subseteq \wp(A^\omega \times A^\omega)$ by

$$R = \{(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)), \alpha) \mid \alpha \in A^\omega\}.$$

We want to show that $R \subseteq M$. By the set-theoretic coinduction principle however, we only have to show $R \subseteq \mathrm{O}(R)$: Let $(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)), \alpha) \in R$ and suppose $\alpha = a : a' : \alpha'$ for some $a, a' \in A, \alpha' \in A^\omega$. Then

$$\begin{aligned}
\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)) &= \mathsf{merge}(\mathsf{odd}(a : a' : \alpha'), \mathsf{even}(a : a' : \alpha')) \\
&= \mathsf{merge}(a : \mathsf{odd}(\alpha'), \mathsf{odd}(a' : \alpha')) \\
&= \mathsf{merge}(a : \mathsf{odd}(\alpha'), a' : \mathsf{odd}(\mathsf{tl}(\alpha'))) \\
&= a : \mathsf{merge}(a' : \mathsf{odd}(\mathsf{tl}(\alpha')), \mathsf{odd}(\alpha')) \\
&= a : a' : \mathsf{merge}(\mathsf{odd}(\alpha'), \mathsf{odd}(\mathsf{tl}(\alpha'))) \\
&= a : a' : \mathsf{merge}(\mathsf{odd}(\alpha'), \mathsf{even}(\alpha')).
\end{aligned}$$

This shows that $(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)), \alpha) \in \mathrm{O}(R)$ and thus that $R \subseteq \mathrm{O}(R)$. Therefore, by (set-theoretic) coinduction, Equation (2.2) holds for all $\alpha \in A^\omega$.

As already mentioned above, this is the kind of coinductive proof that will be given for the arithmetic operations in our project.

Chapter 3

Arithmetic Operations

In this chapter, we use the coinductive principles outlined in Section 2.4 to study operations that compute the sum and linear affine transformations over $\mathbb{Q}$ of signed binary exponent-mantissa numerals. We will keep using the same notation (cf. page 13), however we will additionally write

$$\mathcal{D} = \{-1, 0, 1\}$$

for the set of signed binary digits.

3.1 Preliminaries

Definition 3.1. *The function $\sigma_{|1|} : \mathcal{D}^\omega \to [-1, 1]$ that identifies the real number represented by a signed binary numeral is defined by*

$$\sigma_{|1|}(\alpha) = \sum_{i=1}^{\infty} 2^{-i} \alpha_i .$$

Its extension $\sigma : \mathbb{N} \times \mathcal{D}^\omega \to \mathbb{R}$ to the exponent-mantissa representation is defined by

$$\sigma(e, \alpha) = 2^e \sigma_{|1|}(\alpha) = 2^e \sum_{i=1}^{\infty} 2^{-i} \alpha_i .$$

The following results will be used throughout the remainder of this chapter:

Lemma 3.2 (Properties of $\sigma_{|1|}$). *Let $\alpha \in \mathcal{D}^\omega$. Then*

1. $\left|\sigma_{|1|}(\alpha)\right| \leq 1$
2. $\sigma_{|1|}(\alpha) = \sigma(0, \alpha)$
3. $\sigma_{|1|}(\mathsf{tl}(\alpha)) = 2\sigma_{|1|}(\alpha) - \mathsf{hd}(\alpha)$

Proof.

1. Using the standard expansion of the geometric series,

$$\left|\sigma_{|1|}(\alpha)\right| = \left|\sum_{i=1}^{\infty} 2^{-i} \alpha_i\right| \leq \sum_{i=1}^{\infty} 2^{-i} \left|\alpha_i\right| \leq \sum_{i=1}^{\infty} 2^{-i} = \left(\frac{1}{1 - \frac{1}{2}} - 1\right) = 1.$$

2. Clear from the definition of σ.

3. Direct manipulation:

$$\begin{aligned}\sigma_{|1|}(\mathsf{tl}(\alpha)) &= \sum_{i=1}^{\infty} 2^{-i}(\mathsf{tl}(\alpha))_i \\ &= \sum_{i=1}^{\infty} 2^{-i}\alpha_{i+1} \\ &= \sum_{i=2}^{\infty} 2^{-i+1}\alpha_i \\ &= 2\sum_{i=2}^{\infty} 2^{-i}\alpha_i \\ &= 2\sum_{i=1}^{\infty} 2^{-i}\alpha_i - \alpha_1 \\ &= 2\sigma_{|1|}(\alpha) - \mathsf{hd}(\alpha).\end{aligned}$$

□

Corollary 3.3 (Properties of σ)**.** *Let $(e,m) \in \mathbb{N} \times \mathcal{D}^\omega$. Then*

1. $|\sigma(e,\alpha)| \le 2^e$

2. $\sigma(e,\mathsf{tl}(\alpha)) = 2\sigma(e,\alpha) - 2^e\mathsf{hd}(\alpha)$

Proof.

1. By the definition of σ and by Lemma 3.2(1),

$$|\sigma(e,\alpha)| = 2^e \left|\sigma_{|1|}(\alpha)\right| \le 2^e.$$

2. By the definition of σ and by Lemma 3.2(3),

$$\sigma(e,\mathsf{tl}(\alpha)) = 2^e\sigma_{|1|}(\mathsf{tl}(\alpha)) = 2^e(2\sigma_{|1|}(\alpha) - \mathsf{hd}(\alpha)) = 2\sigma(e,\alpha) - 2^e\mathsf{hd}(\alpha).$$

□

Definition 3.4. *The relation $\sim\, \in \wp((\mathbb{N} \times \mathcal{D}^\omega) \times \mathbb{R})$ that specifies when a real number is represented by a signed binary exponent-mantissa numeral is defined by*

$$\sim\, = \{((e,\alpha),r) \mid \sigma(e,\alpha) = r\}.$$

Its restriction $\sim_{|1|} \in \wp(\mathcal{D}^\omega \times [-1,1])$ to the unit interval is defined by

$$\sim_{|1|} = \{(\alpha,r) \mid \sigma_{|1|}(\alpha) = r\}.$$

We write $(e,\alpha) \sim r$ if $((e,\alpha),r) \in\, \sim$ and $\alpha \sim_{|1|} r$ if $(\alpha,r) \in\, \sim_{|1|}$.

The following operator will form the basis of our coinductive proofs:

Definition 3.5. *The operator $\mathrm{O}_{|1|} : \wp(\mathcal{D}^\omega \times [-1,1]) \to \wp(\mathcal{D}^\omega \times [-1,1])$ is defined by*

$$\mathrm{O}_{|1|}(R) = \{(\alpha,r) \mid (\mathsf{tl}(\alpha), 2r - \mathsf{hd}(\alpha)) \in R\}.$$

Note. The codomain of $\mathrm{O}_{|1|}$ really is $\wp(\mathcal{D}^\omega \times [-1,1])$: Let $R \in \wp(\mathcal{D}^\omega \times [-1,1])$ and $(\alpha, r) \in \mathrm{O}_{|1|}(R)$. Then $(\mathsf{tl}(\alpha), 2r - \mathsf{hd}(\alpha))$ is in R, so that $|2r - \mathsf{hd}(\alpha)| \leq 1$. But $2|r| - |\mathsf{hd}(\alpha)| \leq |2r - \mathsf{hd}(\alpha)|$ and hence $2|r| - |\mathsf{hd}(\alpha)| \leq 1$. This implies $2|r| \leq 1 + |\mathsf{hd}(\alpha)| \leq 2$ and thus $|r| \leq 1$.

Proposition 3.6. $\mathrm{O}_{|1|}$ *has a greatest fixpoint and this fixpoint is* $\sim_{|1|}$.

Proof. $\mathrm{O}_{|1|}$ is clearly monotone and so by Corollary 2.11 has a greatest fixpoint. Call this fixpoint M. We want to prove that $M = \sim_{|1|}$.

Let $(\alpha, r) \in M$. Note that

$$\begin{aligned}(\alpha, r) \in \sim_{|1|} &\Leftrightarrow \sigma_{|1|}(\alpha) = r \\ &\Leftrightarrow \sigma_{|1|}(\alpha) - r = 0 \\ &\Leftrightarrow \left|\sigma_{|1|}(\alpha) - r\right| = 0 \\ &\Leftrightarrow \left|\sigma_{|1|}(\alpha) - r\right| \leq 2^{1-n} \qquad \forall n \in \mathbb{N}. \end{aligned} \tag{3.1}$$

In order to show that $(\alpha, r) \in \sim_{|1|}$, it therefore suffices to prove that Equation (3.1) holds. This can be done via induction:

$\boldsymbol{n = 0}$**:** We have (using Lemma 3.2(1))

$$\begin{aligned}\left|\sigma_{|1|}(\alpha) - r\right| &\leq \left|\sigma_{|1|}(\alpha)\right| + |r| \\ &\leq 1 + |r|.\end{aligned}$$

Now $(\alpha, r) \in M$ implies $|r| \leq 1$ and so

$$\left|\sigma_{|1|}(\alpha) - r\right| \leq 2,$$

as required.

$\boldsymbol{n \to n+1}$**:** Let $n \in N$ be such that $\left|\sigma_{|1|}(\alpha') - r'\right| \leq 2^{1-n}$ for all $((\alpha', r')$ in M. Since M is a fixpoint of $\mathrm{O}_{|1|}$, $M = \mathrm{O}_{|1|}(M)$ and thus $(\alpha, r) \in \mathrm{O}_{|1|}(M)$. This implies that $(\mathsf{tl}(\alpha), 2r - \mathsf{hd}(\alpha))$ is in M so that, by the inductive hypothesis and Lemma 3.2(3):

$$\begin{aligned}2^{1-n} &\geq \left|\sigma_{|1|}(\mathsf{tl}(\alpha)) - 2r + \mathsf{hd}(\alpha)\right| \\ &= \left|2\sigma_{|1|}(\alpha) - \mathsf{hd}(\alpha) - 2r + \mathsf{hd}(\alpha)\right| \\ &= 2\left|\sigma_{|1|}(\alpha) - r\right|.\end{aligned}$$

Hence, as required

$$\left|\sigma_{|1|}(\alpha) - r\right| \leq 2^{1-(n+1)}.$$

Thus by induction, $(\alpha, r) \in \sim_{|1|}$. Since $(\alpha, r) \in M$ was arbitrary, this shows that $M \subseteq \sim_{|1|}$.

In order to show $\sim_{|1|} \subseteq M$, it suffices by Corollary 2.11 to prove

$$\sim_{|1|} \subseteq \mathrm{O}_{|1|}(\sim_{|1|}).$$

Recall

$$\begin{aligned}\mathrm{O}_{|1|}(\sim_{|1|}) &= \{(\alpha, r) \mid (\mathsf{tl}(\alpha), 2r - \mathsf{hd}(\alpha)) \in \sim_{|1|}\} \\ &= \{(\alpha, r) \mid \sigma_{|1|}(\mathsf{tl}(\alpha)) = 2r - \mathsf{hd}(\alpha)\}.\end{aligned}$$

Let $(\alpha, r) \in \sim_{|1|}$. Then by Lemma 3.2(3) and the definition of $\sim_{|1|}$,

$$\sigma_{|1|}(\mathsf{tl}(\alpha)) = 2\sigma_{|1|}(\alpha) - \mathsf{hd}(\alpha) = 2r - \mathsf{hd}(\alpha).$$

Hence $(\alpha, r) \in \mathrm{O}_{|1|}(\sim_{|1|})$ and so $\sim_{|1|} \subseteq \mathrm{O}_{|1|}(\sim_{|1|})$, as required. $\square$

3.2 The Coinductive Strategy

Coinduction gives us the following strategy to define and reason about arithmetic operations on the signed binary exponent-mantissa representation: Let X be a set and suppose we have a function $F : X \times \mathbb{R} \to \mathbb{R}$ for which we want to find an implementation on the exponent-mantissa representation, that is, an operation $\widetilde{F} : X \times (\mathbb{N} \times \mathcal{D}^\omega) \to \mathbb{N} \times \mathcal{D}^\omega$ which satisfies, for all $x \in X$ and $(e, \alpha) \in \mathbb{N} \times \mathcal{D}^\omega$,

$$\widetilde{F}(x, (e, \alpha)) \sim F(x, \sigma(e, \alpha)).$$

We first find a subset Y of X and a function $f : Y \times [-1, 1] \to [-1, 1]$ that, in some intuitive sense, is representative of F on the unit interval. Then, we look for a coinductive definition of an implementation of f on the level of streams which is, similarly to above, an operation $\widetilde{f} : Y \times \mathcal{D}^\omega \to \mathcal{D}^\omega$ that satisfies, for all $y \in Y$ and $\alpha \in \mathcal{D}^\omega$,

$$\widetilde{f}(y, \alpha) \sim_{|1|} f(y, \sigma_{|1|}(\alpha)).$$

In the remainder of this section, we call $\widetilde{f}$ *correct* if and only if this equation holds.

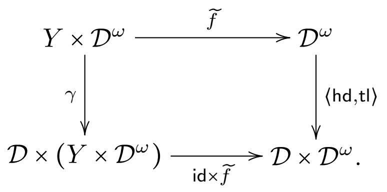
Once we have (coinductively) defined $\widetilde{f}$, we use the set-theoretic coinduction principle outlined in Section 2.4.2 to prove its correctness in the above sense as follows: We define the relation $R \subseteq (\mathcal{D}^\omega \times [-1, 1])$ by

$$R = \left\{ \left(\widetilde{f}(y, \alpha), f\left(y, \sigma_{|1|}(\alpha)\right) \right) \mid y \in Y, \alpha \in \mathcal{D}^\omega \right\},$$

which makes proving the correctness of $\widetilde{f}$ equivalent to showing $R \subseteq \sim_{|1|}$. However, since we have seen in the previous section that $\sim_{|1|}$ is the greatest fixpoint of the monotone operator $\mathrm{O}_{|1|}$, it in fact suffices by the set-theoretic coinduction principle to prove $R \subseteq \mathrm{O}_{|1|}(R)$. Let $\left(\widetilde{f}(y, \alpha), f\left(y, \sigma_{|1|}(\alpha)\right) \right) \in R$ and recall that

$$\begin{aligned} \mathrm{O}_{|1|}(R) &= \{(\alpha, r) \mid (\mathsf{tl}(\alpha), 2r - \mathsf{hd}(\alpha)) \in R\} \\ &= \{(\alpha, r) \mid \exists y' \in Y, \alpha' \in \mathcal{D}^\omega \text{ s.t. } \mathsf{tl}(\alpha) = \widetilde{f}(y', \alpha'), \\ &\qquad\qquad\qquad\qquad\qquad\quad 2r - \mathsf{hd}(\alpha) = f(y', \sigma_{|1|}(\alpha'))\}. \end{aligned}$$

Because $\widetilde{f}$ was defined coinductively, there is a function γ for which the following diagram commutes:



$$\begin{tikzcd} Y \times \mathcal{D}^\omega \arrow[r, "\widetilde{f}"] \arrow[d, "\gamma"'] & \mathcal{D}^\omega \arrow[d, "\langle \mathsf{hd}, \mathsf{tl} \rangle"] \\ \mathcal{D} \times (Y \times \mathcal{D}^\omega) \arrow[r, "\mathsf{id} \times \widetilde{f}"'] & \mathcal{D} \times \mathcal{D}^\omega. \end{tikzcd}$$

Choose y', α' and d such that $\gamma(y, \alpha) = (d, (y', \alpha'))$. The commutativity of the diagram implies $\mathsf{tl}\left(\widetilde{f}(y, \alpha)\right) = \widetilde{f}(y', \alpha')$ and $\mathsf{hd}\left(\widetilde{f}(y, \alpha)\right) = d$, so that all we have to show in order to prove that $\left(\widetilde{f}(y, \alpha), f\left(y, \sigma_{|1|}(\alpha)\right) \right) \in \mathrm{O}_{|1|}(R)$ and thus that $\widetilde{f}$ is correct is

$$2f\left(y, \sigma_{|1|}(\alpha)\right) - d = f\left(y', \sigma_{|1|}(\alpha')\right).$$

Once we have done this, it should be easy to define $\widetilde{F}$ in terms of $\widetilde{f}$ and prove that $\widetilde{F}$ really is an implementation of F using the correctness of $\widetilde{f}$.

The next two sections show how this strategy can be used in practice.

3.3 Addition

Our aim in this section is to define the operation

$$\oplus : (\mathbb{N} \times \mathcal{D}^\omega) \times (\mathbb{N} \times \mathcal{D}^\omega) \to \mathbb{N} \times \mathcal{D}^\omega$$

that computes the sum of two signed binary exponent-mantissa numerals. Following the strategy outlined in Section 3.2, we do this by first (coinductively) defining a corresponding operation on the level of streams. Since signed binary numerals are not closed under addition, the natural choice for this operation is the average function that maps $x, y \in [-1,1]$ to $\frac{x+y}{2}$ $(\in [-1,1])$. We will represent this function on the level of digit streams by defining an operation $\mathsf{avg} : \mathcal{D}^\omega \times \mathcal{D}^\omega \to \mathcal{D}^\omega$ that satisfies

$$\sigma_{|1|}(\mathsf{avg}(\alpha,\beta)) = \frac{\sigma_{|1|}(\alpha) + \sigma_{|1|}(\beta)}{2} \tag{3.2}$$

for all signed binary streams α and β.

3.3.1 Coinductive Definition of avg

Recall that a coinductive definition of avg consists of first specifying a function

$$\gamma : \mathcal{D}^\omega \times \mathcal{D}^\omega \to \mathcal{D} \times (\mathcal{D}^\omega \times \mathcal{D}^\omega)$$

and then taking avg to be the unique induced homomorphism in the diagram

$$\begin{tikzcd}
\mathcal{D}^\omega \times \mathcal{D}^\omega \arrow[r, "\mathsf{avg}"] \arrow[d, "\gamma"'] & \mathcal{D}^\omega \arrow[d, "\langle \mathsf{hd},\mathsf{tl} \rangle"] \\
\mathcal{D} \times (\mathcal{D}^\omega \times \mathcal{D}^\omega) \arrow[r, "\mathsf{id}\times\mathsf{avg}"'] & \mathcal{D} \times \mathcal{D}^\omega.
\end{tikzcd} \tag{3.3}$$

If we let $\alpha = (a_1, a_2, \dots)$ and $\beta = (b_1, b_2, \dots)$ be two signed binary streams and write $\gamma(\alpha,\beta) = (s, (\alpha', \beta'))$, the commutativity of the diagram will imply

$$\mathsf{hd}(\mathsf{avg}(\alpha,\beta)) = s \qquad \text{and} \qquad \mathsf{tl}(\mathsf{avg}(\alpha,\beta)) = \mathsf{avg}(\alpha',\beta'),$$

that is,

$$\mathsf{avg}(\alpha,\beta) = s : \mathsf{avg}(\alpha',\beta'). \tag{3.4}$$

Since our goal is to coinductively define avg in such a way that Equation (3.2) holds, this means that s, α' and β' have to satisfy

$$\sigma_{|1|}(s : \mathsf{avg}(\alpha',\beta')) = \frac{\sigma_{|1|}(\alpha) + \sigma_{|1|}(\beta)}{2}. \tag{3.5}$$

Finding the First Digit s

Suppose we read the first two digits from both α and β. By the definition of $\sigma_{|1|}$,

$$\sigma_{|1|}(\alpha) = \frac{a_1}{2} + \frac{a_2}{4} + \underbrace{\sum_{i=3}^{\infty} 2^{-i} a_i}_{\in[-\frac{1}{4},\frac{1}{4}]} \qquad \text{and} \qquad \sigma_{|1|}(\beta) = \frac{b_1}{2} + \frac{b_2}{4} + \underbrace{\sum_{i=3}^{\infty} 2^{-i} b_i}_{\in[-\frac{1}{4},\frac{1}{4}]}.$$

What we know after having read a_1, a_2, b_1 and b_2 is therefore precisely that

$$\sigma_{|1|}(\alpha) \in \left[\frac{a_1}{2} + \frac{a_2}{4} - \frac{1}{4}, \frac{a_1}{2} + \frac{a_2}{4} + \frac{1}{4}\right] \quad \text{and} \quad \sigma_{|1|}(\beta) \in \left[\frac{b_1}{2} + \frac{b_2}{4} - \frac{1}{4}, \frac{b_1}{2} + \frac{b_2}{4} + \frac{1}{4}\right]$$

which implies

$$\frac{\sigma_{|1|}(\alpha) + \sigma_{|1|}(\beta)}{2} \in \left[p - \frac{1}{4}, p + \frac{1}{4}\right]$$

where

$$p = \frac{a_1 + b_1}{4} + \frac{a_2 + b_2}{8}.$$

If p is less than $-\frac{1}{4}$, then the lower and upper bounds of $\left[p - \frac{1}{4}, p + \frac{1}{4}\right]$ are strictly less than $-\frac{1}{2}$ and 0, respectively, so we can (and must!) output -1 as the first digit. Similarly, if $|p| \le \frac{1}{4}$, then $\left[p - \frac{1}{4}, p + \frac{1}{4}\right] \subseteq [-\frac{1}{2}, \frac{1}{2}]$ so we output 0 and if $p > \frac{1}{4}$, then we output 1. This can be formalized by writing

$$s = \mathrm{sg}_{\frac{1}{4}}(p)$$

where p is as above and the *generalised sign function* $\mathrm{sg}_\epsilon : \mathbb{Q} \to \mathcal{D}$ (taken from [3]) is defined for $\epsilon \in \mathbb{Q}_{\ge 0}$ by

$$\mathrm{sg}_\epsilon(q) = \begin{cases} 1 & \text{if } q > \epsilon \\ 0 & \text{if } |q| \le \epsilon \\ -1 & \text{if } q < -\epsilon. \end{cases}$$

Interlude: Required Lookahead

The above reasoning shows that reading two digits from each of the two input streams is always enough to determine the first digit of output. A natural question to ask is whether the same could be achieved with fewer digits. We will see now that the answer is no: In certain cases, already reading one digit less makes it impossible to produce a digit of output.

Suppose we read the first two digits from α but only the first digit from β. By a reasoning similar to the above, one can show that this implies

$$\frac{\sigma_{|1|}(\alpha) + \sigma_{|1|}(\beta)}{2} \in \left[\frac{a_1 + b_1}{4} + \frac{a_2}{8} - \frac{3}{8}, \frac{a_1 + b_1}{4} + \frac{a_2}{8} + \frac{3}{8}\right].$$

If for instance $a_1 = b_1 = 1$ and $a_2 = 0$, this is the interval $\left[\frac{1}{8}, \frac{7}{8}\right]$. Since $\left[\frac{1}{8}, \frac{7}{8}\right]$ is contained in $[0, 1]$, we can safely output 1 without examining any further input. If, however, $a_1 = 1$ and $b_1 = a_2 = 0$, the interval becomes $\left[-\frac{1}{8}, \frac{5}{8}\right]$. Because $-\frac{1}{8}$ can only be represented by a stream starting with -1 or 0 and $\frac{5}{8}$ can only be represented by a stream starting with 1, we do not know at this stage what the first digit of output should be. We require more input.

Determining (α', β')

Recall Equation (3.5) which stated the condition for the correctness of avg as

$$\sigma_{|1|}(s : \mathsf{avg}(\alpha', \beta')) = \frac{\sigma_{|1|}(\alpha) + \sigma_{|1|}(\beta)}{2}. \tag{3.5}$$

Both sides of this equation can be rewritten. For the left-hand side, we use the ansatz

$$\alpha' = a'_1 : \mathsf{tl}^2(\alpha) \qquad\qquad \beta' = b'_1 : \mathsf{tl}^2(\beta),$$

where a'_1 and b'_1 are signed binary digits. **Assuming avg will be correct(!)**, this yields

$$\begin{aligned}
\sigma_{|1|}(s : \mathsf{avg}(\alpha', \beta')) &= \frac{s}{2} + \frac{1}{2}\sigma_{|1|}(\mathsf{avg}(\alpha', \beta')) \\
&= \frac{s}{2} + \frac{1}{2}\frac{\sigma_{|1|}(\alpha') + \sigma_{|1|}(\beta')}{2} \qquad (!) \\
&= \frac{s}{2} + \frac{1}{2}\frac{\frac{a'_1}{2} + \frac{1}{2}\sigma_{|1|}(\mathsf{tl}^2(\alpha)) + \frac{b'_1}{2} + \frac{1}{2}\sigma_{|1|}(\mathsf{tl}^2(\beta))}{2} \\
&= \frac{s}{2} + \frac{a'_1 + b'_1}{8} + \frac{1}{8}(\sigma_{|1|}(\mathsf{tl}^2(\alpha)) + \sigma_{|1|}(\mathsf{tl}^2(\beta))).
\end{aligned}$$

For the right-hand side,

$$\begin{aligned}
\frac{\sigma_{|1|}(\alpha) + \sigma_{|1|}(\beta)}{2} &= \frac{\frac{a_1}{2} + \frac{a_2}{4} + \sum_{i=3}^{\infty} 2^{-i} a_i + \frac{b_1}{2} + \frac{b_2}{4} + \sum_{i=3}^{\infty} 2^{-i} b_i}{2} \\
&= \underbrace{\frac{a_1 + b_1}{4} + \frac{a_2 + b_2}{8}}_{p} + \frac{1}{2}\left(\sum_{i=3}^{\infty} 2^{-i} a_i + \sum_{i=3}^{\infty} 2^{-i} b_i\right) \\
&= p + \frac{1}{8}\left(\sum_{i=3}^{\infty} 2^{-(i-2)} a_i + \frac{1}{8}\sum_{i=3}^{\infty} 2^{-(i-2)} b_i\right) \\
&= p + \frac{1}{8}\left(\sigma_{|1|}\left(\mathsf{tl}^2(\alpha)\right) + \sigma_{|1|}\left(\mathsf{tl}^2(\beta)\right)\right).
\end{aligned} \tag{3.6}$$

Using these steps, Equation (3.5) becomes

$$\frac{s}{2} + \frac{a'_1 + b'_1}{8} = p,$$

i.e.

$$\frac{a'_1 + b'_1}{8} = p - \frac{s}{2}.$$

Now a'_1 and b'_1 are signed binary digits so that a'_1 must be 1 when $p - \frac{s}{2} > \frac{1}{8}$ and -1 when $p - \frac{s}{2} < \frac{1}{8}$. Also, when $\left|p - \frac{s}{2}\right| \leq \frac{1}{8}$, we can take a'_1 to be 0 by the symmetry of the equation. We therefore choose

$$a'_1 = \mathrm{sg}_{\frac{1}{8}}\left(p - \frac{s}{2}\right),$$

which forces us take

$$b'_1 = 8p - 4s - a'_1.$$

An easy case analysis on p shows that b'_1 is in fact a signed binary digit.

Final Definition of γ and avg

To summarize, the function $\gamma : \mathcal{D}^\omega \times \mathcal{D}^\omega \to \mathcal{D} \times (\mathcal{D}^\omega \times \mathcal{D}^\omega)$ is defined for any signed binary numerals $\alpha = (a_1, a_2, \dots)$ and $\beta = (b_1, b_2, \dots)$ by

$$\gamma(\alpha, \beta) = (s, (\alpha', \beta'))$$

where

$$s = \mathrm{sg}_{\frac{1}{4}}(p) \qquad \alpha' = a_1' : \mathsf{tl}^2(\alpha) \qquad \beta' = b_1' : \mathsf{tl}^2(\beta)$$

$$p = \frac{a_1 + b_1}{4} + \frac{a_2 + b_2}{8} \qquad a_1' = \mathrm{sg}_{\frac{1}{8}}\left(p - \frac{s}{2}\right) \qquad b_1' = 8p - 4s - a_1'$$

and the generalised sign function sg_ϵ is as on page 24. By the finality of the $\mathcal{D}$-stream coalgebra $(\mathcal{D}^\omega, \langle \mathsf{hd}, \mathsf{tl}\rangle)$, this induces the (unique) homomorphism avg in Diagram (3.3):

$$\begin{tikzcd}
\mathcal{D}^\omega \times \mathcal{D}^\omega \arrow[r, "\mathsf{avg}"] \arrow[d, "\gamma"'] & \mathcal{D}^\omega \arrow[d, "\langle \mathsf{hd},\mathsf{tl}\rangle"] \\
\mathcal{D} \times (\mathcal{D}^\omega \times \mathcal{D}^\omega) \arrow[r, "\mathsf{id}\times\mathsf{avg}"'] & \mathcal{D} \times \mathcal{D}^\omega
\end{tikzcd} \tag{3.3}$$

In particular, if we let s, α' and β' be as above, then

$$\mathsf{avg}(\alpha, \beta) = s : \mathsf{avg}(\alpha', \beta'). \tag{3.4}$$

3.3.2 Correctness of avg

In order to prove the correctness of avg, we first need the following lemma:

Lemma 3.7 (Correctness of γ)**.** *Let α and β be signed binary streams and suppose s, α' and β' are as in the definition of γ. Then*

$$\frac{\sigma_{|1|}(\alpha) + \sigma_{|1|}(\beta)}{2} = \frac{s}{2} + \frac{1}{2}\frac{\sigma_{|1|}(\alpha') + \sigma_{|1|}(\beta')}{2}.$$

Proof. For the left-hand side, we refer back to Equation (3.6):

$$\frac{\sigma_{|1|}(\alpha) + \sigma_{|1|}(\beta)}{2} = p + \frac{1}{8}\left(\sigma_{|1|}\left(\mathsf{tl}^2(\alpha)\right) + \sigma_{|1|}\left(\mathsf{tl}^2(\beta)\right)\right),$$

where p is as in the definition of γ.
For the right-hand side, recall that a_1' and b_1' were constructed so that

$$\frac{s}{2} + \frac{a_1' + b_1'}{8} = p.$$

Together with the definitions of $\sigma_{|1|}$, α' and β', this implies

$$\begin{aligned}
\frac{s}{2} + \frac{1}{2}\frac{\sigma_{|1|}(\alpha') + \sigma_{|1|}(\beta')}{2} &= \frac{s}{2} + \frac{1}{2}\frac{\frac{a_1'}{2} + \frac{1}{2}\sigma_{|1|}\left(\mathsf{tl}^2(\alpha)\right) + \frac{b_1'}{2} + \frac{1}{2}\sigma_{|1|}\left(\mathsf{tl}^2(\beta)\right)}{2} \\
&= \frac{s}{2} + \frac{a_1' + b_1'}{8} + \frac{1}{8}\left(\sigma_{|1|}\left(\mathsf{tl}^2(\alpha)\right) + \sigma_{|1|}\left(\mathsf{tl}^2(\beta)\right)\right) \\
&= p + \frac{1}{8}\left(\sigma_{|1|}\left(\mathsf{tl}^2(\alpha)\right) + \sigma_{|1|}\left(\mathsf{tl}^2(\beta)\right)\right),
\end{aligned}$$

as required. □

Theorem 3.8 (Correctness of avg). *Let α and β be signed binary streams. Then*

$$\mathsf{avg}(\alpha,\beta) \sim_{|1|} \frac{\sigma_{|1|}(\alpha)+\sigma_{|1|}(\beta)}{2}.$$

Proof. Following the strategy outlined in Section 3.2, we define $R \subseteq (\mathcal{D}^\omega \times [-1,1])$ by

$$R = \left\{ \left(\mathsf{avg}(\alpha,\beta), \frac{\sigma_{|1|}(\alpha)+\sigma_{|1|}(\beta)}{2} \right) \;\middle|\; \alpha,\beta \in \mathcal{D}^\omega \right\}.$$

The statement of the lemma is equivalent to $R \subseteq \sim_{|1|}$. By the set-theoretic coinduction principle, this can be proved by showing $R \subseteq \mathrm{O}_{|1|}(R)$.
Let $\left(\mathsf{avg}(\alpha,\beta), \frac{\sigma_{|1|}(\alpha)+\sigma_{|1|}(\beta)}{2}\right) \in R$ and recall

$$\begin{aligned}\mathrm{O}_{|1|}(R) &= \{(\alpha,r) \mid (\mathsf{tl}(\alpha), 2r-\mathsf{hd}(\alpha)) \in R\} \\ &= \Big\{(\alpha,r) \mid \exists \alpha',\beta' \in \mathcal{D}^\omega \text{ s.t. } \mathsf{tl}(\alpha) = \mathsf{avg}(\alpha',\beta') \\ &\qquad\qquad \text{and } 2r-\mathsf{hd}(\alpha) = \tfrac{\sigma_{|1|}(\alpha')+\sigma_{|1|}(\beta')}{2}\Big\}.\end{aligned}$$

Take α', β' as in the definition of γ. By the commutativity of Diagram (3.3), we have

$$\mathsf{tl}(\mathsf{avg}(\alpha,\beta)) = \mathsf{avg}(\alpha',\beta') \qquad \text{and} \qquad \mathsf{hd}(\mathsf{avg}(\alpha,\beta)) = s.$$

Moreover, by the correctness of γ (Lemma 3.7),

$$\frac{\sigma_{|1|}(\alpha)+\sigma_{|1|}(\beta)}{2} = \frac{s}{2} + \frac{1}{2}\frac{\sigma_{|1|}(\alpha')+\sigma_{|1|}(\beta')}{2},$$

which implies

$$2\frac{\sigma_{|1|}(\alpha)+\sigma_{|1|}(\beta)}{2} - \mathsf{hd}(\mathsf{avg}(\alpha,\beta)) = \frac{\sigma_{|1|}(\alpha')+\sigma_{|1|}(\beta')}{2}.$$

This shows that $\left(\mathsf{avg}(\alpha,\beta), \frac{\sigma_{|1|}(\alpha)+\sigma_{|1|}(\beta)}{2}\right) \in \mathrm{O}_{|1|}(R)$ and thus that $R \subseteq \mathrm{O}_{|1|}(R)$. □

3.3.3 Definition and Correctness of ⊕

The final ingredient we need before being able to define the addition operation ⊕ is the small lemma below. It will give us a way of finding a common exponent for two signed binary exponent-mantissa numerals.

Lemma 3.9. *If α is a signed binary stream, then for all $n \in \mathbb{N}$*

$$\sigma_{|1|}\left((0:)^n\alpha\right) = 2^{-n}\sigma_{|1|}(\alpha)$$

where $(0:)^n\alpha$ is the result of prepending n zeros to α.

Proof. By induction on n. The result clearly holds for the case $n=0$. Also, if it holds for some $n \in \mathbb{N}$, then

$$\sigma_{|1|}\left((0:)^{n+1}\alpha\right) = \sigma_{|1|}\left(0:((0:)^n\alpha)\right) = \frac{0}{2} + \frac{1}{2}\sigma_{|1|}\left((0:)^n\alpha\right) = 2^{-(n+1)}\sigma_{|1|}(\alpha),$$

as required. □

Definition 3.10. *The addition operation* $\oplus : (\mathbb{N} \times \mathcal{D}^\omega) \times (\mathbb{N} \times \mathcal{D}^\omega) \to \mathbb{N} \times \mathcal{D}^\omega$ *is defined for any signed binary exponent-mantissa numerals* (e, α) *and* (f, β) *by*

$$(e,\alpha) \oplus (f,\beta) = \left(\max(e,f)+1, \mathsf{avg}\left((0:)^{\max(e,f)-e}\alpha, (0:)^{\max(e,f)-f}\beta\right)\right).$$

Theorem 3.11 (Correctness of $\oplus$)**.** *For any* $(e,\alpha), (f,\beta) \in \mathbb{N} \times \mathcal{D}^\omega$,

$$(e,\alpha) \oplus (f,\beta) \sim (\sigma(e,\alpha) + \sigma(f,\beta)).$$

Proof. By the correctness of avg (Theorem 3.8) and Lemma 3.9,

$$\begin{aligned} &\sigma_{|1|}\left(\mathsf{avg}\left((0:)^{\max(e,f)-e}\alpha, (0:)^{\max(e,f)-f}\beta\right)\right) \\ &\qquad = 2^{e-\max(e,f)-1}\sigma_{|1|}(\alpha) + 2^{f-\max(e,f)-1}\sigma_{|1|}(\beta). \end{aligned}$$

Hence, by the definitions of σ and $\oplus$,

$$\begin{aligned} \sigma((e,\alpha) \oplus (f,\beta)) &= 2^e\sigma_{|1|}(\alpha) + 2^f\sigma_{|1|}(\beta) \\ &= \sigma(e,\alpha) + \sigma(f,\beta). \end{aligned}$$

□

3.4 Linear Affine Transformations over $\mathbb{Q}$

This section introduces the operation $\mathsf{Lin}_\mathbb{Q} : \mathbb{Q} \times (\mathbb{N} \times \mathcal{D}^\omega) \times \mathbb{Q} \to \mathbb{N} \times \mathcal{D}^\omega$ that represents the mapping

$$\begin{array}{ccc} \mathbb{Q} \times \mathbb{R} \times \mathbb{Q} & \longrightarrow & \mathbb{R} \\ (u,x,v) & \longmapsto & ux + v. \end{array}$$

As in the last section, we first (coinductively) define the corresponding operation $\mathsf{lin}_\mathbb{Q}$ on the level of streams in such a way that

$$\sigma_{|1|}(\mathsf{lin}_\mathbb{Q}(u,\alpha,v)) = u\sigma_{|1|}(\alpha) + v, \tag{3.7}$$

where $\alpha = (a_1, a_2, \dots)$ is any signed binary stream and $u, v \in \mathbb{Q}$ are such that $|u| + |v| \le 1$. This last condition ensures that the result of the operation can in fact be represented by a signed binary numeral.

3.4.1 Coinductive Definition of $\mathsf{lin}_\mathbb{Q}$

Before we give a coinductive definition of $\mathsf{lin}_\mathbb{Q}$, we first find a lower bound on the lookahead required to output a digit in the general case. In some cases such as when $u = 0$ and $v = 1$, we can output one (or even all) digits of the result without examining the input. In other cases however, even one digit does not suffice: If $u = \frac{3}{4}$, $v = \frac{1}{4}$ and $a_1 = 0$, then all we know after having read a_1 is that

$$u \underbrace{\sigma_{|1|}(\alpha)}_{\in[-\frac{1}{2},\frac{1}{2}]} + v \in \left[-\frac{1}{8}, \frac{3}{8}\right].$$

Since this interval is not contained in any of $[-1,0]$, $[-\frac{1}{2},\frac{1}{2}]$ and $[0,1]$, we cannot determine what the first digit of output should be. This shows that a lookahead of at least two digits is required to produce a digit of output in the general case.

In order to (coinductively) define $\mathsf{lin}_{\mathbb{Q}}$, we first specify its domain C that captures the constraint mentioned in the introduction to this section:

$$C = \{(u, \alpha, v) \in \mathbb{Q} \times \mathcal{D}^{\omega} \times \mathbb{Q} : |u| + |v| \leq 1\}\,.$$

The coinductive part of the definition now consists of specifying a function $\delta : C \to \mathcal{D} \times C$ and taking $\mathsf{lin}_{\mathbb{Q}}$ to be the (unique) ensuing homomorphism in the diagram

$$\begin{tikzcd} C \arrow[r, "\mathsf{lin}_{\mathbb{Q}}"] \arrow[d, "\delta"'] & \mathcal{D}^{\omega} \arrow[d, "\langle \mathsf{hd}, \mathsf{tl} \rangle"] \\ \mathcal{D} \times C \arrow[r, "\mathsf{id} \times \mathsf{lin}_{\mathbb{Q}}"'] & \mathcal{D} \times \mathcal{D}^{\omega}. \end{tikzcd} \tag{3.8}$$

Let $(u, \alpha, v) \in C$ and write $\delta(u, \alpha, v) = (l, (u', \alpha', v'))$. If we follow the steps above, then the commutativity of the diagram will imply

$$\mathsf{lin}_{\mathbb{Q}}(u, \alpha, v) = l : \mathsf{lin}_{\mathbb{Q}}(u', \alpha', v'). \tag{3.9}$$

Since we want $\mathsf{lin}_{\mathbb{Q}}$ to satisfy the correctness condition given by Equation (3.7), this means that we have to choose l, u', α' and v' in such a way that

$$\sigma_{|1|}(l : \mathsf{lin}_{\mathbb{Q}}(u', \alpha', v')) = u\sigma_{|1|}(\alpha) + v. \tag{3.10}$$

To find l, we observe that

$$\begin{aligned} u\sigma_{|1|}(\alpha) + v &= u\left(\frac{a_1}{2} + \frac{a_2}{4} + \frac{1}{4}\sigma_{|1|}\left(\mathsf{tl}^2(\alpha)\right)\right) + v \\ &= u\left(\frac{a_1}{2} + \frac{a_2}{4}\right) + v + \underbrace{\frac{u}{4}\sigma_{|1|}\left(\mathsf{tl}^2(\alpha)\right)}_{\in\left[-\frac{|u|}{4}, \frac{|u|}{4}\right]}, \end{aligned} \tag{3.11}$$

so that

$$u\sigma_{|1|}(\alpha) + v \in \left[q - \frac{|u|}{4}, q + \frac{|u|}{4}\right] \subseteq \left[q - \frac{1}{4}, q + \frac{1}{4}\right]$$

where

$$q = u\left(\frac{a_1}{2} + \frac{a_2}{4}\right) + v.$$

By a reasoning similar to that behind the choice of s in the definition of γ/avg, this implies that we can choose

$$l = \mathrm{sg}_{\frac{1}{4}}(q).$$

For u', α' and v', we note that if $\mathsf{lin}_{\mathbb{Q}}$ is correct, then

$$\sigma_{|1|}(l : \mathsf{lin}_{\mathbb{Q}}(u', \alpha', v')) = \frac{l}{2} + \frac{1}{2}\left(u'\sigma_{|1|}(\alpha') + v'\right).$$

But we also have by Equation (3.11) that

$$u\sigma_{|1|}(\alpha) + v = \frac{l}{2} + \frac{1}{2}\left(\frac{u}{2}\sigma_{|1|}\left(\mathsf{tl}^2(\alpha)\right) + 2q - l\right).$$

Since we want Equation (3.10) to hold, we therefore compare terms and choose

$$u' = \frac{u}{2} \qquad\qquad \alpha' = \mathsf{tl}^2(\alpha) \qquad\qquad v' = 2q - l.$$

We have to check that $|u'| + |v'| \le 1$. This can be done by a case analysis on δ, for which we will need

$$\begin{aligned} |q| &= \left| u\left(\frac{a_1}{2} + \frac{a_2}{2}\right) + v \right| \\ &\le \frac{3}{4}|u| + |v| \\ &= |u| + |v| - \frac{|u|}{4} \\ &\le 1 - \frac{|u|}{4}. \end{aligned} \tag{3.12}$$

$l = 0$: By the definition of l, we have $|q| \le \frac{1}{4}$. Also, $|u| + |v| \le 1$ implies $|u| \le 1$ and so

$$|u'| + |v'| = \frac{|u|}{2} + 2|q| \le \frac{1}{2} + \frac{1}{2} = 1.$$

$l = 1$: We have $q > \frac{1}{4}$ and thus $v' = 2q - l > -\frac{1}{2}$. Also, by Equation (3.12), $v' \le 1 - \frac{|u|}{2}$. These inequalities tell us that $|v'| \le \max\left(\left|-\frac{1}{2}\right|, 1 - \frac{|u|}{2}\right)$. But, $|u| \le 1$ implies $1 - \frac{|u|}{2} \ge \frac{1}{2}$, so this is in fact $|v'| \le 1 - \frac{|u|}{2} = 1 - |u'|$. The result follows.

$l = -1$: Similarly to the case $l = 1$, we have $\frac{|u|}{2} - 1 \le v' < \frac{1}{2}$ and $\frac{|u|}{2} - 1 \le -\frac{1}{2}$. Hence $|v'| \le 1 - \frac{|u|}{2} = 1 - |u'|$ and so $|u'| + |v'| \le 1$.

Final Definition of δ and $\mathsf{lin}_{\mathbb{Q}}$

Let $C = \{(u, \alpha, v) \in \mathbb{Q} \times \mathcal{D}^\omega \times \mathbb{Q} : |u| + |v| \le 1\}$. The function $\delta : C \to \mathcal{D} \times C$ is defined for any $(u, \alpha, v) \in C$ by

$$\delta(u, \alpha, v) = (l, (u', \alpha', v'))$$

where, writing $\alpha = (a_1, a_2, \dots)$,

$$\begin{aligned} l &= \mathrm{sg}_{\frac{1}{4}}(q) \qquad & u' &= \frac{u}{2} \qquad & \alpha' &= \mathsf{tl}^2(\alpha) \\ q &= u\left(\frac{a_1}{2} + \frac{a_2}{4}\right) + v \qquad & v' &= 2q - l. \end{aligned}$$

This allows us to take $\mathsf{lin}_{\mathbb{Q}}$ to be the unique induced homomorphism in Diagram (3.8):

$$\begin{tikzcd} C \arrow[r, "\mathsf{lin}_{\mathbb{Q}}"] \arrow[d, "\delta"'] & \mathcal{D}^\omega \arrow[d, "\langle \mathsf{hd}, \mathsf{tl} \rangle"] \\ \mathcal{D} \times C \arrow[r, "\mathsf{id} \times \mathsf{lin}_{\mathbb{Q}}"'] & \mathcal{D} \times \mathcal{D}^\omega \end{tikzcd} \qquad [3.8]$$

The commutativity of the diagram implies that, for $(u, \alpha, v) \in C$,

$$\mathsf{lin}_{\mathbb{Q}}(u, \alpha, v) = l : \mathsf{lin}_{\mathbb{Q}}(u', \alpha', v') \qquad [3.9]$$

where l, u', α' and v' are as above.

3.4.2 Correctness of $\mathsf{lin}_{\mathbb{Q}}$

Theorem 3.12 (Correctness of $\mathsf{lin}_{\mathbb{Q}}$). *For any* $(u, \alpha, v) \in C$,

$$\mathsf{lin}_{\mathbb{Q}}(u, \alpha, v) \sim_{|1|} u\sigma_{|1|}(\alpha) + v.$$

Proof. Define $R \subseteq (\mathcal{D}^{\omega} \times [-1, 1])$ by

$$R = \left\{ \left(\mathsf{lin}_{\mathbb{Q}}(u, \alpha, v), u\sigma_{|1|}(\alpha) + v\right) \middle| \, (u, \alpha, v) \in C \right\}.$$

We show that $R \subseteq \sim_{|1|}$ by showing that $R \subseteq \mathrm{O}_{|1|}(R)$.
Recall

$$\begin{aligned} \mathrm{O}_{|1|}(R) &= \{(\alpha, r) \mid (\mathsf{tl}(\alpha), 2r - \mathsf{hd}(\alpha)) \in R\} \\ &= \{(\alpha, r) \mid \exists (u', \alpha', v') \in C \text{ s.t. } \mathsf{tl}(\alpha) = \mathsf{lin}_{\mathbb{Q}}(u', \alpha', v'), \\ &\qquad\qquad\qquad\qquad\qquad 2r - \mathsf{hd}(\alpha) = u'\sigma_{|1|}(\alpha') + v'\}. \end{aligned}$$

Let $\left(\mathsf{lin}_{\mathbb{Q}}(u, \alpha, v), u\sigma_{|1|}(\alpha) + v\right) \in R$ and take $l \in \mathcal{D}$, $(u', \alpha', v') \in C$ as in the definition of $\mathsf{lin}_{\mathbb{Q}}$. By the commutativity of Diagram (3.8), we have

$$\mathsf{tl}(\mathsf{lin}_{\mathbb{Q}}(u, \alpha, v)) = \mathsf{lin}_{\mathbb{Q}}(u', \alpha', v').$$

Moreover,

$$\begin{aligned} u'\sigma_{|1|}(\alpha') + v' &= \frac{u}{2}\sigma_{|1|}\left(\mathsf{tl}^2(\alpha)\right) + 2q - l \\ &= \frac{u}{2}\left(2\sigma_{|1|}(\mathsf{tl}(\alpha)) - a_2\right) + 2q - l \\ &= \frac{u}{2}\left(4\sigma_{|1|}(\alpha) - 2a_1 - a_2\right) + 2q - l \\ &= 2u\sigma_{|1|}(\alpha) - 2u\left(\frac{a_1}{2} + \frac{a_2}{4}\right) + 2q - l \\ &= 2u\sigma_{|1|}(\alpha) - 2(q - v) + 2q - l \\ &= 2(u\sigma_{|1|}(\alpha) + v) - l. \end{aligned}$$

Since the commutativity of Diagram (3.8) also implies $\mathsf{hd}(\mathsf{lin}_{\mathbb{Q}}(u, \alpha, v)) = l$, this shows that $\left(\mathsf{lin}_{\mathbb{Q}}(u, \alpha, v), u\sigma_{|1|}(\alpha) + v\right) \in \mathrm{O}_{|1|}(R)$ and thus that $R \subseteq \mathrm{O}_{|1|}(R)$. $\square$

3.4.3 Definition and Correctness of $\mathsf{Lin}_{\mathbb{Q}}$

Let $u, v \in \mathbb{Q}$ and $(e, \alpha) \in \mathbb{N} \times \mathcal{D}^{\omega}$. In order to define $\mathsf{Lin}_{\mathbb{Q}}$ that computes $u\sigma(e, \alpha) + v$ using $\mathsf{lin}_{\mathbb{Q}}$ from the previous section, we have to find scaled-down versions $u', v' \in \mathbb{Q}$ of u and v, respectively such that $|u'| + |v'| \leq 1$. The scale will be determined using the (computable!) function $\lceil \log_2 \rceil$:

Definition 3.13. *The function* $\lceil \log_2 \rceil : \mathbb{Q}_{\geq 0} \to \mathbb{N}$ *is defined by*

$$\lceil \log_2 \rceil(s) = \begin{cases} 0 & \textit{if } s \leq 1 \\ 1 + \lceil \log_2 \rceil\left(\frac{s}{2}\right) & \textit{otherwise.} \end{cases}$$

Lemma 3.14. *For any* $s \in \mathbb{Q}_{\geq 0}$,

$$\log_2(s) \leq \lceil \log_2 \rceil(s)$$

where $\log_2 : \mathbb{R}_{>0} \to \mathbb{R}$ *is the (standard) logarithm to base* 2.

Proof. Observe that $\mathbb{Q}_{\geq 0} = \bigcup_{n \in \mathbb{N}} \lceil \log_2 \rceil^{-1}(n)$ so that it suffices to show that the result holds for all $n \in \mathbb{N}$ and $s \in \lceil \log_2 \rceil^{-1}(n)$. This can be done by induction.

If $n = 0$ and $s \in \lceil \log_2 \rceil^{-1}(n)$ then $s \leq 1$ and thus $\log_2(s) \leq 0 = \lceil \log_2 \rceil(s)$. Conversely, suppose the result holds for all $t \in \lceil \log_2 \rceil^{-1}(n)$ where $n \in \mathbb{N}$ is fixed. Let $s \in \lceil \log_2 \rceil^{-1}(n+1)$. Then $n+1 = \lceil \log_2 \rceil(s) = 1 + \lceil \log_2 \rceil\left(\frac{s}{2}\right)$ so that $\frac{s}{2} \in \lceil \log_2 \rceil^{-1}(n)$. This implies by our assumption that $\log_2(s) - 1 = \log_2\left(\frac{s}{2}\right) \leq \lceil \log_2 \rceil\left(\frac{s}{2}\right) = n$ and thus that $\log_2(s) \leq \lceil \log_2(s) \rceil$. □

Corollary 3.15. *Let $u, v \in \mathbb{Q}$ and $u' = 2^{-n}u$, $v' = 2^{-n}v$ where $n = \lceil \log_2 \rceil(|u|+|v|)$. Then $|u'| + |v'| \leq 1$.*

Proof. By Lemma 3.14, $n = \lceil \log_2 \rceil(|u| + |v|) \geq \log_2(|u| + |v|)$. This implies that $2^{-n} \leq \frac{1}{|u|+|v|}$ so that $|u'| + |v'| = 2^{-n}(|u| + |v|) \leq \frac{|u|+|v|}{|u|+|v|} = 1$, as required. □

Corollary 3.15 implies that we can write

$$\begin{aligned} u\sigma(e,\alpha) + v &= 2^n u' 2^e \sigma_{|1|}(\alpha) + 2^n v' \\ &= 2^{e+n}\left(u'\sigma_{|1|}(\alpha) + 2^{-e}v'\right) \end{aligned} \tag{3.13}$$

where n, u', v' are as above and (!)

$$|u'| + 2^{-e}|v'| \leq |u'| + |v'| \leq 1. \tag{3.14}$$

This immediately gives rise to

Definition 3.16 ($\mathsf{Lin}_{\mathbb{Q}}$)**.** *The function $\mathsf{Lin}_{\mathbb{Q}} : \mathbb{Q} \times (\mathbb{N} \times \mathcal{D}^\omega) \times \mathbb{Q} \to \mathbb{N} \times \mathcal{D}^\omega$ is defined for $u, v \in \mathbb{Q}$ and $(e, \alpha) \in \mathbb{N} \times \mathcal{D}^\omega$ by*

$$\mathsf{Lin}_{\mathbb{Q}}(u, (e,\alpha), v) = (e+n, \mathsf{lin}_{\mathbb{Q}}(u', \alpha, 2^{-e}v'))$$

where

$$n = \lceil \log_2 \rceil(|u| + |v|) \qquad u' = 2^{-n}u \qquad v' = 2^{-n}v.$$

Theorem 3.17 (Correctness of $\mathsf{Lin}_{\mathbb{Q}}$)**.** *For any $u, v \in \mathbb{Q}$, $(e, \alpha) \in \mathcal{D}^\omega$*

$$\sigma(\mathsf{Lin}_{\mathbb{Q}}(u, (e,\alpha), v)) = u\sigma_{|1|}(\alpha) + v.$$

Proof. Let n, u' and v' be as in the definition of $\mathsf{Lin}_{\mathbb{Q}}$. Then by Equation (3.14), $|u'| + 2^{-e}|v'| \leq 1$, so that, by the correctness of $\mathsf{lin}_{\mathbb{Q}}$ (Theorem 3.12) and Equation (3.13),

$$\begin{aligned} \sigma(\mathsf{Lin}_{\mathbb{Q}}(u, (e,\alpha), v)) &= 2^{e+n}\sigma_{|1|}\left(\mathsf{lin}_{\mathbb{Q}}\left(u', \alpha, 2^{-e}v'\right)\right) \\ &= 2^{e+n}\left(u'\sigma_{|1|}(\alpha) + 2^{-e}v'\right) \\ &= u\sigma(e,\alpha) + v. \end{aligned}$$

□

Chapter 4

Haskell Implementation

4.1 Overview

This section briefly describes the most important parts of the Haskell implementation of the arithmetic operations introduced in the previous chapter. The full (though still not very long) code is given in Appendix A.2.

The most important types in the Haskell implementation are `SD`, `SDS` and `Real` which represent signed binary digits, signed binary streams and numerals in the exponent-mantissa representation, respectively. The functions `fromSD` and `sdToRational` convert objects of type `SD` to the Haskell types `Integer` and `Rational`, respectively, so that calculations on signed binary digits can be performed in the natural way.

Apart from some type conversions, the implementations of `avg` and `linQ` correspond exactly to the definitions of avg and $\mathsf{lin}_{\mathbb{Q}}$:

```
approximate :: SDS -> Int -> Rational
-- Calculates the value represented by the given
   stream to an accuracy of 2^(-n).
approximate (a1 : a') 1 = (fromSD a1) % 2
approximate (a1 : a') n
  = (sdToRational a1 + approximate a' (n - 1)) / 2

approximateReal :: Real -> Int -> Rational
approximateReal (e, m) n = 2^e * approximate m n

approximateAsDecimal :: SDS -> Int -> IO ()
```

```
    s = sg (1 % 4) p
    p = (fromSD a1 + fromSD b1) % 4 + (fromSD a2 +
        fromSD b2) % 8
    a1' = sg (1 % 8) (p - (fromSD s) % 2)
    b1' = sg 0 (8 * p - 4 * sdToRational s -
        sdToRational a1')

prependZeros :: Integer -> SDS -> SDS
```

```
-- Prepends the given number of zeros to the given
   signed binary stream.
-- Defined in Section 3.3.3.
prependZeros 0 a = a
prependZeros n a = prependZeros (n - 1) (Z : a)
```

In order to implement ⊕, we need to be able to prepend a stream with an arbitrary number of zeros (this was $(0\,:)^n$ in the definition of ⊕). This is done by the function `prependZeros` using which, again, the translation from mathematical definition to implementation is immediate:

```
showRationalAsDecimal q decPlaces
  = show p ++ "." ++ showFractionalPart (10 * r) (
     denominator q) decPlaces
  where
    (p, r) = divMod (numerator q) (denominator q)
    showFractionalPart n d 0 = []
```

Finally, `linQ_real` is the Haskell implementation of the function $\mathsf{Lin}_{\mathbb{Q}}$. It uses the implementation `ceilLog2` of the function $\lceil \log_2 \rceil$:

```
add :: Real -> Real -> Real
-- Calculates the sum of two real numbers to an
   arbitrary precision.
-- Introduced and proved correct in Section 3.3.3.
add (e, a) (f, b)
  = (max e f + 1, avg (prependZeros ((max e f) - e) a)
      (prependZeros ((max e f) - f) b))

ceilLog2 :: Rational -> Integer
-- Exactly computes the ceiling function of the
   logarithm to base 2 of the given number.
-- Introduced and proved correct in Section 3.4.3.
```

As pointed out, all translations from the mathematical definitions to the respective Haskell implementations are straightforward and do not require more than a few syntactic changes and extra type conversions. The credit for this is of course mostly due to Haskell, however it also shows that the coinductive approach lends itself well to an implementation in a functional programming language that supports corecursive definitions.

Manual tests were performed and verified that the developed algorithms are indeed correct.

4.2 Related Operations in Literature

Operations on signed binary streams similar to those studied in this work can for instance be found in [3, 5](avg) and [13]($\mathsf{lin}_{\mathbb{Q}}$). We would like to find out whether these are output-equivalent to our operations, that is, whether they produce the same streams when given the same input.

The reference [13] defines the function `applyfSDS` which corresponds to our $\mathsf{lin}_{\mathbb{Q}}$ and gives a Haskell implementation. The average operation `av` from [3] can easily be translated as follows:

```
av :: SDS -> SDS -> SDS
-- Calculates the average (a + b)/2
-- Implementation of the definition given in [3]
av (a1 : a') (b1 : b') = av_aux a' b' ((fromSD a1) + (
    fromSD b1))

av_aux :: SDS -> SDS -> Integer -> SDS
av_aux (a1 : a') (b1 : b') i
  = d : av_aux a' b' (j - 4 * (fromSD d))
  where
    j = 2 * i + (fromSD a1) + (fromSD b1)
    d = sg 2 (j % 1)
```

For `linQ` and `applyfSDS`, we try the input `u = 2%3`, `v = -1%3` and `a = toSDS 1%5`, where `toSDS` converts a Haskell `Rational` to a signed digit stream (cf. Appendix A.2). For the first 5 signed binary digits, this yields

Digit:	1	2	3	4	5
`linQ`:	-1	1	0	1	0
`applyfSDS`:	-1	1	0	1	-1

So `linQ` and `applyfSDS` are *not* output-equivalent (even though, of course, the infinite streams do represent the same real number).

For `avg` and `av`, we compare the respective outputs for the following values (again using `toSDS`):

a	b
`0`	`0`
`1`	`1`
`1`	`-1`
`1%2`	`1%2`
`3%4`	`-1%3`
`-13%27`	`97%139`

In all cases, the first 100 digits of output are the same. We also note that `avg` and `av` have the same effective lookahead: For the first digit of output, both read two digits from each of the two input streams. For every following digit, `av` reads one digit from both streams while `avg` reads two, of which however the first was generated in the last iteration of `avg`. For these reasons, we conjecture that the `Integer` parameter `i` of `av_aux` and the two new digits `a1'` and `b1'` that are prepended to the tails of the input streams in `avg` are in a one-to-one correspondence.

Chapter 5

Conclusion, Related and Future Work

We have proposed a general strategy similar to that given in [3] for studying exact arithmetic operations on the signed binary exponent-mantissa representation using coinduction. The main steps of this strategy consist of giving a coalgebraic coinductive definition of a function that represents the required operation on the level of streams and then proving the correctness of this function using set-theoretic coinduction. In this way, the strategy brings together two different (though related) forms of coinduction that are usually not combined.

Using the proposed strategy, we have obtained operations that compute the sum and linear affine transformations over $\mathbb{Q}$ of real numbers given in the above-mentioned representation and proved their correctness. In each case, the strategy gave us a criterion that guided our search for the coinductive part of the definition. More importantly, the use of coalgebraic coinductive definition principles provided us with a pattern of choices to make in the set-theoretic coinduction proofs which essentially reduced these proofs to one key step.

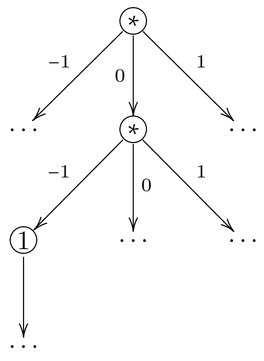
Our Haskell implementation shows that the operations we have developed can be used in practice. Moreover, the fact that the algorithms it consists of are almost literal translations of the corecursive definitions of the corresponding operations indicates that the coinductive approach is well-suited for an implementation in a lazy functional programming language. This is particularly important because it alleviates the likelihood of introducing an error when going from mathematical definitions to implementation.

Evaluating the results of a project as theoretical as this one is difficult. The main contribution of this work is to show that the combination of coalgebraic coinductive definition principles and set-theoretic coinduction can be used in the context of exact real arithmetic and, for the aforementioned reasons, we feel that this combination is very fruitfuil. One drawback of our approach might be that it does not take performance considerations into account so that the ensuing algorithms may not be very efficient.

Operations on signed binary streams similar to those studied in this project can for instance be found in [3,5,13]. Our experiments with appropriate implementations have shown that the average operation `av` from [3] may be closely related to the function `avg` defined in this work while the function `applyfSDS` from [13] is certainly different from our corresponding operation $\mathsf{lin}_{\mathbb{Q}}$.

An interesting approach related to that followed in this project is *program extraction* as for instance studied in [13]. It involves giving inductive and (set-theoretic) coinductive proofs of general computation problems and then extracting algorithms and programs from these proofs in a systematic way. This has the advantage that the obtained programs are correct by construction and that formalisations become simpler because the approximating data are left implicit in the proofs.

As regards future work, there are several interesting possibilities. We believe that the coinductive approach should lend itself similarly well to other representations and a next step could be to apply it to the more general linear fractional transformations. Another possibility would be to study coinductive representations of *operations* rather than just real numbers. For instance, an operation on a signed binary stream can be represented as an infinite tree whose edges represent the digits of the input and whose nodes represent the output digits or a symbol $*$ that indicates that more input is required:



On this representation, for instance an integral operator could then be defined.

Bibliography

[1] Study of iterations of a recurrence sequence of order two. Example application of the CADNA library; Available from `http://www-pequan.lip6.fr/cadna/Examples_Dir/ex4.php`.

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

[3] U. Berger and T. Hou. Coinduction for exact real number computation. *Theory of Computing Systems*, 43:394–409, December 2008.

[4] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language Reference Manual.* Springer Berlin / Heidelberg, first edition, November 1991.

[5] A. Ciaffaglione and Di P. Gianantonio. A certified, corecursive implementation of exact real numbers. *Theoretical Computer Science*, 351(1):39–51, February 2006.

[6] A. Edalat and R. Heckmann. *Computing with Real Numbers*, volume 2395 of *Lecture Notes in Computer Science*, pages 953–984. Springer Berlin / Heidelberg, January 2002.

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

[8] R. W. Gosper. Item 101b: Continued fraction arithmetic. Memo 239 ("HAKMEM"), MIT Artificial Intelligence Laboratory, February 1972.

[9] T. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM (JACM)*, 48(5):1038–1068, September 2001.

[10] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[11] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, third edition, 1997.

[12] M. Lenisa. From set-theoretic coinduction to coalgebraic coinduction: some results, some problems. In B. Jacobs and J. Rutten, editors, *Proc. Coalgebraic Methods in Computer Science(CMCS'99),* volume 19 of *Electr. Notes Theor. Comput. Sci.*, 1999.

[13] S. Lloyd. Implementation and verification of exact real number algorithms. Bachelor's Thesis, Department of Computer Science, University of Wales Swansea, 2009.

[14] V. Ménissier-Morain. Arbitrary precision real arithmetic: Design and algorithms. Submitted to J. Symbolic Computation. Available from `http://www-calfor.lip6.fr/~vmm/documents/submission_JSC.ps.gz`, September 1996.

[15] D. Plume. A calculator for exact real number computation. 4th Year Project Report, Departments of Computer Science and Artificial Intelligence, University of Edinburgh, 1998.

[16] H. G. Rice. Recursive real numbers. *Proceedings of the American Mathematical Society*, 5(5):784–791, October 1954.

[17] J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theoretical Computer Science*, 249(1):3–80, October 2000.

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

[19] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. A correction. *Proceedings of the London Mathematical Society*, 2(43):544–546, 1937.

[20] J. Vignes. Discrete stochastic arithmetic for validating results of numerical software. *Numerical Algorithms*, 37(1-4):377–390, December 2004.

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

[22] S. Wolfram. *Mathematica: a System for Doing Mathematics by Computer.* Addison Wesley, second edition, April 1991.

Appendix A

Code Listings

A.1 Calculating the Muller-Sequence

This section gives the code that was used to compute the values for the floating point arithmetic example in section 1.1.

The two standard C programs `Single.c` and `Double.c` use single and double precision, respectively, to calculate elements 1 to N of the sequence given on page 1, where N can be specified on the command line. As a sanity check, both report a measure of the precision with which the system they are run on executes floating point arithmetic. This precision should be `1.19209e-07` for `Single.c` and `2.22045e-16` for `Double.c`.

Listing A.1: Single.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int MAX;
    if(argc == 1 || (MAX = atoi(argv[1])) == 0) {
        printf("Single.c\n");
        printf("========\n");
        printf("Uses single precision floating point
           arithmetic to compute elements 1...N of the
            sequence\n");
        printf("\ta_0 = 11/2, a_1 = 61/11, a_(n+1) =
           111 - (1130 - 3000/a_(n-1))/a_n\n");
        printf("\n");
        printf("Usage: \n");
        printf("======\n");
        printf("\tSingle N\nwhere N is the largest
           element of the sequence that should be
           computed\n");
        return 0;
    }

    float a_n_minus_1 = 5.5f;
    float a_n = 61.0f/11.0f;
    int n;
    for (n = 1; n <= MAX; n++) {
        printf("%f\n", a_n);
        float tmp = a_n;
        a_n = 111 - (1130 - 3000 / a_n_minus_1) / a_n;
        a_n_minus_1 = tmp;
    }

    float precision = 1.0f;
    float y;
    while( (y = precision + 1.0f) != 1.0f )
        precision /= 2.0f;
    precision *= 2.0f;
    printf("Precision is %g \n", precision);
    return 0;
}
```

Listing A.2: Double.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int MAX;
    if(argc == 1 || (MAX = atoi(argv[1])) == 0) {
        printf("Double.c\n");
        printf("========\n");
        printf("Uses double precision floating point
           arithmetic to compute elements 1...N of the
            sequence\n");
        printf("\ta_0 = 11/2, a_1 = 61/11, a_(n+1) =
           111 - (1130 - 3000/a_(n-1))/a_n\n");
        printf("\n");
        printf("Usage: \n");
        printf("======\n");
        printf("\tDouble N\nwhere N is the largest
           element of the sequence that should be
           computed\n");
        return 0;
    }


    double a_n_minus_1 = 5.5;
    double a_n = 61.0/11.0;
    int n;
    for (n = 1; n <= MAX; n++) {
        printf("%1f\n", a_n);
        double tmp = a_n;
        a_n = 111 - (1130 - 3000 / a_n_minus_1) / a_n;
        a_n_minus_1 = tmp;
    }

    double precision = 1.0;
    double y;
    while( (y = precision + 1.0) != 1.0 )
        precision /= 2.0;
    precision *= 2.0;
    printf("Precision is %g \n", precision);
    return 0;
}
```

`Arbitrary.bc` uses a fixed but arbitrary precision to calculate the nth element of the sequence given on page 1. As mentioned in section 1.1, it is a script for the UNIX program **bc**.

Listing A.3: Arbitrary.bc

```
# Call with UNIX utility bc
# Uses arbitrary precision floating point arithmetic
   to compute the Nth element of the sequence
#    a_0 = 11/2, a_1 = 61/11, a_(n+1) = 111 - (1130 -
   3000/a_(n-1))/a_n

# Precision
print "\nPlease enter the precision (in number of
   decimal places after the decimal point) the
   calculations should be executed with: ";
scale = read();

print "\nPlease enter which element of the sequence
   you would like to have computed: ";
max = read();

a_n_minus_1 = 5.5;
a_n = 61.0/11.0;

for(n = 2; n <= max; n++) {
    tmp = a_n;
    a_n = 111 - (1130 - 3000/a_n_minus_1)/a_n;
    a_n_minus_1 = tmp;
}

print "\na_", max, " calculated using the given
   precision: ", a_n, "\n";
print "distance from the exact value of a_n: ", a_n -
   (6 ^ (max + 1) + 5 ^ (max + 1)) / (6 ^ max + 5 ^
   max), "\n";
quit;
```

A.2 Haskell Implementation

This section gives the full code of the Haskell implementation of the arithmetic operations developed in Chapter 3. An overview of the most important parts of this code is given in Chapter 4.

Listing A.4: Coinductive Reals.hs

```haskell
import Ratio

data SD = N | Z | P deriving Eq
type SDS = [ SD ]
type Real = (Integer, SDS)

instance Show SD where
  show = show . fromSD

fromSD :: SD -> Integer
fromSD N = -1
fromSD Z = 0
fromSD P = 1

sdToRational :: SD -> Rational
sdToRational = (%1) . fromSD

sg :: Rational -> Rational -> SD
-- Generalised sign function
-- Introduced in Section 3.3.1.
sg d x
  | x > d       = P
  | abs(x) <= d = Z
  | x < -d      = N

toSDS :: Rational -> SDS
-- Returns the signed digit stream that exactly
   represents the given rational.
toSDS u
  = d : toSDS (2 * u - sdToRational d)
  where
    d = sg (1 % 2) u

toReal :: Rational -> Real
toReal q
  = (n, toSDS (q/(2^n)))
  where
    n = ceilLog2 q

approximate :: SDS -> Int -> Rational
-- Calculates the value represented by the given
   stream to an accuracy of 2^(-n).
approximate (a1 : a') 1 = (fromSD a1) % 2
approximate (a1 : a') n
  = (sdToRational a1 + approximate a' (n - 1)) / 2

approximateReal :: Real -> Int -> Rational
approximateReal (e, m) n = 2^e * approximate m n

approximateAsDecimal :: SDS -> Int -> IO ()
-- Approximates (and prints) the value of the given
   SDS to an accuracy of n decimal places
approximateAsDecimal ds n = putStr (
   showRationalAsDecimal (approximate ds (floor ((
   fromIntegral n) * log(10) / log(2)))) n)

showRationalAsDecimal :: Rational -> Int -> String
showRationalAsDecimal q n
  | q < 0 = '-' : showRationalAsDecimal (-q) n
showRationalAsDecimal q 0 = show ((numerator q) `div`
   (denominator q))
showRationalAsDecimal q decPlaces
  = show p ++ "." ++ showFractionalPart (10 * r) (
     denominator q) decPlaces
  where
    (p, r) = divMod (numerator q) (denominator q)
    showFractionalPart n d 0 = []
    showFractionalPart n d decPlaces = (show (n `div`
       d)) ++ showFractionalPart (10*(n `mod` d)) d (
       decPlaces - 1)

avg :: SDS -> SDS -> SDS
-- Calculates the average (a + b)/2.
-- Introduced in Section 3.3.1, proved correct in
   Section 3.3.2.
avg (a1 : a2 : a') (b1 : b2 : b')
  = s : avg (a1' : a') (b1' : b')
  where
    s = sg (1 % 4) p
    p = (fromSD a1 + fromSD b1) % 4 + (fromSD a2 +
       fromSD b2) % 8
    a1' = sg (1 % 8) (p - (fromSD s) % 2)
    b1' = sg 0 (8 * p - 4 * sdToRational s -
       sdToRational a1')

prependZeros :: Integer -> SDS -> SDS
-- Prepends the given number of zeros to the given
   signed binary stream.
-- Defined in Section 3.3.3.
prependZeros 0 a = a
prependZeros n a = prependZeros (n - 1) (Z : a)

add :: Real -> Real -> Real
```

```
-- Calculates the sum of two real numbers to an
   arbitrary precision.
-- Introduced and proved correct in Section 3.3.3.
add (e, a) (f, b)
  = (max e f + 1, avg (prependZeros ((max e f) - e) a)
      (prependZeros ((max e f) - f) b))

ceilLog2 :: Rational -> Integer
-- Exactly computes the ceiling function of the
   logarithm to base 2 of the given number.
-- Introduced and proved correct in Section 3.4.3.
ceilLog2 s
  | s <= 1    = 0
  | otherwise = 1 + ceilLog2 (s / 2)

linQ :: Rational -> SDS -> Rational -> SDS
-- Computes the linear affine transformation u*a + v
   where |a| <= 1 and |u| + |v| <= 1.
-- Introduced in Section 3.4.1, proved correct in
   Section 3.4.2.
linQ u (a1 : a2 : a') v
  = d : linQ u' a' v'
  where
    u' = u / 2
    v' = 2 * r - sdToRational d
    r = u * (fromSD a1 % 2 + fromSD a2 % 4) + v
    d = sg (1 % 4) r

linQ_Real :: Rational -> Real -> Rational -> Real
-- Computes the linear affine transformation u*a + v
   to an arbitrary precision.
-- Introduced and proved correct in Section 3.4.3.
linQ_Real u (e, a) v
  = (e + n, linQ u' a (v'/(2^e)))
  where
    n = ceilLog2 (abs u + abs v)
    u' = u/(2^n)
    v' = v/(2^n)

av :: SDS -> SDS -> SDS
-- Calculates the average (a + b)/2
-- Implementation of a definition given in [3]
av (a1 : a') (b1 : b') = av_aux a' b' ((fromSD a1) + (
   fromSD b1))

av_aux :: SDS -> SDS -> Integer -> SDS
av_aux (a1 : a') (b1 : b') i
  = d : av_aux a' b' (j - 4 * (fromSD d))
  where
    j = 2 * i + (fromSD a1) + (fromSD b1)
    d = sg 2 (j % 1)
```