University of Edinburgh
Department of Computer Science

Exact Arithmetic Using the Golden Ratio

4th Year Project Report

David McGaw <davm@dcs.ed.ac.uk>
Supervisor: Martín Escardó <mhe@dcs.ed.ac.uk>

1999

Abstract: The usual approach to real arithmetic on computers consists of using floating point approximations. Unfortunately, floating point arithmetic can sometimes produce wildly erroneous results. One alternative approach is to use exact real arithmetic. Exact real arithmetic allows exact real number computation to be performed without the roundoff errors characteristic of other methods. Conventional representations such as decimal and binary notation are inadequate for this purpose. We consider an alternative representation of reals, using the golden ratio. Firstly we look at the golden ratio and its relation to the Fibonacci series, finding some interesting identities. Then we implement algorithms for basic arithmetic operations, trigonometric and logarithmic functions, conversion and integration. These include new algorithms for addition, multiplication, multiplication by 2, division by 2 and manipulating finite and infinite streams.

Acknowledgements

I would especially like to than my supervisor Martín Escardó for his patience, time, advice and support whilst doing this project.

I would like to thank John Longley for taking the time to explain his algorithm for integration to me.

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

Contents

1. Introduction

The aim of this project is to construct a calculator that implements exact arithmetic using Golden Ratio notation. The calculator that we have implemented has basic arithmetic functions, for example addition, subtraction, multiplication and division; elementary functions, for example trigonometric and logarithmic functions; basic operations on functions for example definite integration.

We begin by motivating the use of exact arithmetic and, in particular, Golden Ratio (GR) notation.

1.1 Problems with Floating-Point Representation

In floating point representation, finitely many digits are kept. This introduces rounding errors, which when combined with other values can make the error of the answer grow. In some cases the rounding errors in a calculation can be so large that the result is meaningless.

For example take the following system of equations, this example is taken from [9]:

$$\begin{array}{rrrcl} -1.41x_1 & 2x_2 & & = & 1 \\ x_1 & -1.41x_2 & x_3 & = & 1 \\ & 2x_2 & -1.41x_3 & = & 1 \end{array}$$

Solving using Gaussian elimination (without row operations) and rounding at each stage to 3 significant figures gives: $x_1 = 0.709, x_2 = 1, x_3 = 0.7$. But the correct answer (to 3 significant figures) is: $x_1 = x_2 = x_3 = 1.69$. We can see that these rounding errors have resulted in the answer that we get is completely meaningless.

Increasing the number of digits of accuracy will not solve this problem, it will only delay it, because by doing more operations we could get the same magnitude of error. Similarly allowing row operations (swapping rows to make coefficients as large as possible) would be useful but would not remove errors, since doing any operations on a number containing an error can make the error grow uncontrollably.

There are some ways to estimate the error, and therefore how accurate the answer is. These are as follows:

- Floating Point Arithmetic with error analysis.
- Interval Arithmetic.

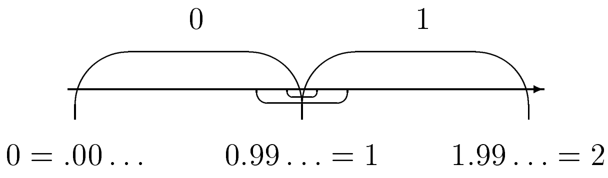

Figure 1.1: Ranges in decimal notation

- Stochastic Rounding.
- Symbolic Manipulation.

A discussion of these techniques can be found in [14].

Of all these methods only symbolic manipulation will actually represent a value exactly, the rest will only approximate the value. But at some stage using symbolic manipulation we would have to convert to decimal using one of the other methods, we would then get the same problems associated with the other methods.

1.2 A Solution to Rounding Errors

We can argue that if errors are introduced because we truncate the remaining digits of a number, we could get rid of this error by not truncating the number, representing a number as an infinite stream of digits.

We would ideally like to represent a number as an infinite stream of decimal digits, but we cannot. Simple operations such as addition may not be possible, for example if we try to add 4/9 with 5/9 as follows we will not be able to produce one digit of output:

$$\begin{array}{rll} & 0.444\ldots & = 4/9 \\ + & 0.555\ldots & = 5/9 \\ \hline & ? & \end{array}$$

To add these numbers together we must start at the most significant bit of the input (there is no least significant bit). We will try to find the first digit of the output.

Looking at the first digit of the inputs (0 for both inputs) we can tell that the sum will start with either a 0 or a 1, but we do not know which. Therefore we need to look at the second digit of each input (a 4 and a 5). We still cannot decide the first digit of the result. The remaining digits of input will always be a 4 and a 5, so no matter how many digits we look at we will not be able to decide the first digit of the output. A graphical representation of this is in Figure 1.1.

The two large arcs represent the range of the possible values of numbers beginning with the digit 0 and 1. We can see from the diagram that the ranges of the two arcs intersect at one point $0.99\ldots = 1.00\ldots$. The small arcs represent the ranges of possible values of the sum of $4/9 + 5/9$ after we have looked at each successive digit, notice that the range of the sum is always in both the range for digits beginning with 0 and 1. We can see the range of the answer is getting much smaller after looking at each successive digit. But no matter how many digits we look at, in this case the value of the result will always in both the interval beginning with 0 and 1. Therefore we will never be able to determine the first digit of the output by looking at a finite number of digits of the input, another example of this can be found in [6].

In the above example, to decide the first digit of output we would have to look at an infinite number of input digits. We can say that addition using infinite streams of decimals is not of finite character, where "finite character" is defined as — A finite number of output digits depends only on a finite number of input digits. It is important to note that the set of functions that are computable is strictly contained in the set of functions of finite character. If a function is computable then it must be of finite character. However it is possible that a function is of finite character but is not computable.

Other approaches to exact computation, which are of finite character are as follows:

- Signed Digits & Variations.
- Limits of Effective Cauchy Sequence.
- B-approximable.
- Dedekind Cuts.
- Continued Fractions.
- Mobius Transformations.

Papers related to these approaches that we have studied can be found in the references at the end of this report. In [14], David Plume does a survey of papers related to these notations. It is useful to know that all these representations are equivalent, meaning that if we have a stream in one representation we can write an algorithm that will convert it to any other representation. Golden Ratio notation discussed below is also equivalent to the above approaches.

The problem with representing a number as an infinite stream of decimal digits is that the intersection of two intervals is at most one point. Therefore, if a number is at one of these points, we have to decide in which interval it is in. Since we can never decide we will never get an answer.

The way to solve this problem is to let the intersection of two intervals contain more than one point. This means that at no point do we need to decide if the value of a stream is greater than some value. By looking at enough digits we can

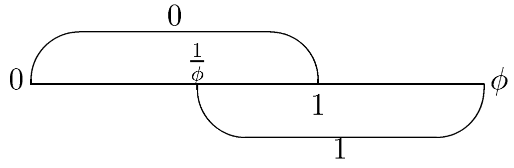

Figure 1.2: Ranges in Golden Ratio notation

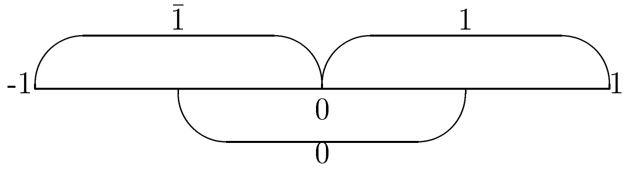

Figure 1.3: Ranges in Signed Binary notation

determine the value of the answer within some ϵ that we determine. Then we make this ϵ small enough so that it can not contain both end points.

It is impossible to explicitly store an infinite number of digits in finite space. The main purpose of using an infinite streams is that we can look at 'enough' digits of the answer to guarantee the result within some error bound, where the number of digits we look at depends on the error bound.

Therefore we never consider the whole stream but only a finite portion. The reason why this is desirable is that in floating point representation we can control the error by varying the number of digits of accuracy but this will only bound the value of a number before we have carried out an operation. The error of the result of an operation is not necessarily bounded to the same error as the input because the input's are exactly the input, without an error. Then the output is exact. We introduce the error when truncating the infinite stream. Therefore in this case an error is only introduced in the stage that we convert to decimal, up to this point the answer is completely correct. When we convert to decimal we can then control the error.

The graphical representation of Golden Ratio notation is in Figure 1.2, where ϕ is the golden ratio (see Chapter 2). This is going to be discussed in Section 1.3 Similarly in Signed Binary (SB) notation we have the same properties as with the Golden Ratio notation, the graphical representation of SB notation can be seen in Figure 1.3. David Plume uses this SB notation to implement a calculator, using exact arithmetic.

1.3 Why Golden Ratio notation?

Golden Ratio notation uses infinite streams of digits consisting of the digits $\{0, 1\}$, the base we use is the golden ratio. This is described fully in Chapter 2.

We could have decided to use SB notation instead of Golden Ratio notation. For example David Plume [14] has implemented a calculator with transcendental functions using SB notation in Haskell.

The main reasons that we chose to represent numbers in GR notation is that every digit in a GR notation stream can only have two possible values (0 or 1), but a digit in SB notation can have three values ($\bar{1}, 0$ or 1).

If we wanted to implement these algorithms in hardware, we would have a problem which is how to implement the three digits from SB notation in a system that only uses binary digits. But GR notation uses binary digits, so it would be easier to implement in hardware.

All the algorithms we will write or implement are written using case analysis, for every digit in GR notation. We thus only have to consider two possible values, but in SB notation we have to consider three values for each digit. Therefore we consider fewer cases after looking at a number of digits in GR notation than in SB notation.

Finally in SB notation we have two identities $1\bar{1} \equiv 01$ and also $\bar{1}1 \equiv 0\bar{1}$. Notice that the second identity is the negation of the first. Ideally we would like to only have to deal with one identity, so we would like the identity to be the negation of itself. This suggests a base with 2 digits, one digit being the negation of the other, for example binary digits. The simplest identities would be $1 \equiv 0$, $10 \equiv 01$ or $100 \equiv 011$. The first two of these we can see are meaningless, but the third identity is not. Finding the positive solution of this equation gives us ϕ, the golden ratio, and the GR notation.

We could have also defined representations with the identities $1000 \equiv 0111$ and so on, however if the number of digits we have to consider in algorithms increase linearly, the number of cases we would have to consider would increase exponentially.

1.4 Work Performed

This project can be split into two parts: the first part is on the theory behind the Golden Ratio notation, including why we use it, and identities involving the golden ratio, that contains an interesting result concerning summing parts of the Fibonacci series, we also prove another interesting result showing that every value of the form $\frac{1}{2^k}$ has a periodic representation in GR notation. The second part of this report concerns the algorithms that we use to construct all the functions used by the implemented calculator. These can be further subdivided into 4 sections: manipulation of streams, conversion between representations, intersection of nested sequences of intervals and definite integration.

Manipulation of streams covers both algorithms that try to compare two streams and algorithms rewriting a stream in a different, but equivalent, form. It would be useful if we could compare streams in golden ratio notation to find if one stream has a greater numerical value than the other. There is a simple algorithm to decide this for finite decimal numerals, but it has two problems: this may not give the correct answer for GR streams, and also we are not able to determine if the value of two streams are equal, because infinatley many digits have to be compared. Therefore in this case the algorithm would not terminate. We can solve the first problem by rewriting the two streams using identities in such a way that the simple algorithm will always give the correct answer if it terminates. This is called lexicographical normalisation (Section 4.2).

We can then use lexicographical normalisation to implement the cases function, which is the closest that we can get to an **if ... then ... else ...** function, though we do need an extra condition (Section 4.3).

The algorithms that we will implement to do basic operations will be of two different forms: The algorithms designed and proved by Pietro Di Gianantonio take in streams then combine them in the desired way, and then outputs them. The algorithms that we have written work by rewriting one stream in a given form to another stream with the same value but in a different form. For example if there was a function that took an infinite stream of digits containing either digits 0 or 1, and then rewrote this stream so that its value was the same but now all digits were either 0 or 2, then this can clearly be used as an algorithm to divide a stream by 2 (Section 5.4).

In order to print the results, we consider conversion between Golden Ratio notation and decimal. When we convert from GR to decimal we have to take a finite portion of the GR stream. Also we convert from GR to SB and discuss converting from SB to GR. It is useful to convert between SB notation because all algorithms invented and implemented by David Plume in [14] can be used in conjunction with our algorithms (Chapter 6).

Intersection of nested sequences of intervals can be used to find the intersection of a series of intervals that are converging at one point. We can use the Taylor series expansion of a function and then, by finding upper and lower bounds up to a certain step, we can construct a converging sequence of intervals. This is how we calculate the values of $e^x, \sin(x), \cos(x), \ln(x)$ and x^y (Chapter 7).

Finally we deal with definite integration. We implement definite integration by using Riemann strips. There are two main differences between this type of implementation and other implementations. These are: This algorithm will only calculate the value of the integral within some predetermined error value, therefore the answer we get will be within this error value but will not be exact. Secondly this algorithm uses a function called Mod that although appears to do nothing, it actually counts the number of digits that are examined to get a finite portion of the output. This is crucial in calculating the strips so that each strip will be within a specified error bound (Chapter 8).

Summary of Contributions

There are several original algorithms, some of which have been implemented in this calculator. They implement, multiplication by 2, division by 2, addition, multiplication, conversion to and from decimal and signed-binary notation. This is in addition to the algorithms for addition, subtraction, multiplication and division designed and proved by Di Gianantonio [6].

In the course of making this calculator, original technical work was done. For example, lexicographical normalisation in Golden Ratio notation is new, although this is based on the algorithm in Signed Binary notation by Escardó [5]. Since the identity used is different, the resulting algorithm is completely new. Lexicographical normalisation is not used directly. However it is used in the cases function and the intersection of nested intervals functions.

The flip function described in Section 4.1 is also original material. The algorithm and proof are new. This function only works with streams of finite length. Although it may appear that there are no applications in a calculator using infinite streams, in the algorithms for definite integration and also for conversion from GR notation to Decimal and back to GR notation, finite streams are used and also in the above algorithms the flip function is used.

Whilst making this calculator, new algorithms and identities were encountered that are not implemented directly or indirectly by the calculator. This is part of the material covered in Chapter 2 about the Golden Ratio notation. In particular the proof of, and the algorithm for calculating the periodic representation of $\frac{1}{2^k}$ in GR notation. Other new material is the identity concerning summing terms of the Fibonacci sequence. Although these identities are not implemented in the calculator, we included them because they are of interest.

1.5 Organisation

In chapter 2 we consider the Golden Ratio notation and associated identities. In chapter 3 we discuss how we implemented this calculator. Chapters 4, 5, 6, 7 and 8 we design or redesign, prove and implement all the functions associated with this calculator.

2. Golden Ratio Notation

This chapter is more mathematically based. We will look at the relationship between Fibonacci numbers and the golden ratio.

The golden ratio ϕ is defined as the positive solution of $\phi^2 = \phi + 1$. This definition of ϕ is important because it gives us an essential identity. In decimal notation the only identity that we have is $.999\ldots = 1.00\ldots$. This is because, as discussed in the introduction, intervals only intersect at one point. In SB notation, because we have an overlap of intervals, we have an infinite number of identities, however they all follow from the identity $\phi^2 = \phi + 1$. This is discussed below.

The number denoted by a decimal in a stream is defined by:

$$[\![d]\!]_d = \sum_{i=1}^{\infty} d_i.10^{-i} = d_1.10^{-1} + d_2.10^{-2} + \ldots$$

From the diagram of the ranges in Figure 1.2 Golden Ratio notation, it is obvious that we can only have streams with values in the range $[0, \phi]$.

These streams are made up of an infinite number of 0's and 1's. This notation is called simplified notation. Simplified notation is defined as follows:

$$[\![\alpha]\!]_s = \sum_{i=1}^{\infty} \alpha_i.\phi^{-i} = \alpha_1.\phi^{-1} + \alpha_2.\phi^{-2} + \ldots$$

where $\alpha = 0 \cdot \alpha_1\alpha_2\alpha_3\ldots$, α_i is the i'th digit of α, also $\alpha_{|i} = \alpha_i\alpha_{i+1}\ldots$ and $\alpha_i \in \mathbf{2}$, **2** represents the set {0,1} and **3** represents the set {0,1,2}.

To be able to represent all real numbers we associate an exponent to the simplified notation to get the full notation. This is defined as follows:

$$[\![z : \alpha]\!]_f = (-1 + [\![\alpha]\!]_s).\phi^{2z}$$

Here z is an integer, that is associated with α.

By using our definition of simplified notation and also using the definition of the golden ratio we can find the following equivalent identities:

$$\begin{gathered} \phi^2 = \phi + 1 \\ \phi^2 = \phi^1 + \phi^0 \\ \phi^{-1} = \phi^{-2} + \phi^{-3} \\ [\![100(0)^\omega]\!]_s = [\![011(0)^\omega]\!]_s \end{gathered}$$

Here α^ω means that α is repeated infinatly.

We can multiply this identity by any power of ϕ, to give the identity:

$$\alpha 100\beta \equiv \alpha 011\beta$$

$\alpha \equiv \beta$ means $[\![\alpha]\!]_s = [\![\beta]\!]_s$, where $\alpha = \mathbf{2}^*$ and $\beta = \mathbf{2}^\omega$. ω means repeated infinitely, $*$ means repeated a finite amount of times. We will exploit this identity in all algorithms written using GR notation.

2.1 Infinite sums

When we prove algorithms in the next sections, we will use the following two identities, that we prove here:

$$\sum_{k=0}^{\infty} \frac{1}{\phi^k} = \phi^2 \tag{2.1}$$

$$\sum_{k=0}^{\infty} \frac{1}{\phi^{2k}} = \phi \tag{2.2}$$

We can prove this using the Geometric series defined as follows:

$$\frac{1}{1-t} = \sum_{k=0}^{\infty} t^k, |t| < 1 \tag{2.3}$$

Proof of 2.1 Using 2.3 with $t = \frac{1}{\phi}$, note $|\frac{1}{\phi}| < 1$, gives

$$\sum_{k=0}^{\infty} \frac{1}{\phi^k} = \phi^2$$

since $1 - \frac{1}{\phi} = \frac{1}{\phi^2}$.
Proof of 2.2 Using 2.3 with $t = \frac{1}{\phi^2}$, note $|\frac{1}{\phi^2}| < 1$, gives

$$\sum_{k=0}^{\infty} \frac{1}{\phi^{2k}} = \phi$$

since $1 - \frac{1}{\phi^2} = \frac{1}{\phi}$.

2.2 Basic identities

We defined the golden ratio in the previous section, as the positive solution to the equation:

$$\phi^2 = \phi + 1$$

However we could also define ϕ to be:

$$\phi = \lim_{i\to\infty}\frac{\mathrm{F}_i}{\mathrm{F}_{i+1}}$$

where F_i is the ith element of the Fibonacci sequence, defined as follows:

$$\begin{aligned}&\mathrm{F}_0 = 0\\ &\mathrm{F}_1 = 1\\ &\mathrm{F}_i = \mathrm{F}_{i-1} + \mathrm{F}_{i-2}\end{aligned}$$

Using this definition of the Fibonacci numbers we can rearrange $\mathrm{F}_i = \mathrm{F}_{i-1} + \mathrm{F}_{i-2}$ as $\mathrm{F}_{i-2} = \mathrm{F}_i - \mathrm{F}_{i-1}$, to give the Fibonacci numbers F_i when $i < 0$.

Therefore the initial sequence of Fibonacci numbers is as follows:

...	−8	−7	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8	...
...	−21	13	−8	5	−3	2	−1	1	0	1	1	2	3	5	8	13	21	...

It is possible to see that

$$\text{If } i < 0 \text{ then } \mathrm{F}_i = (-1)^{i+1}\mathrm{F}_{-i}$$

This is proved in [8, Section 6.6, page 279].

2.3 ϕ^k and the Fibonacci numbers

We will show that

$$\forall k \in \mathbb{Z}\ \phi^k = \mathrm{F}_k.\phi + \mathrm{F}_{k-1}$$

Proof

We will prove this by induction on k. In this proof we will assume that $k \geq 0$, and then that $k \leq 0$.

Base Case

$$\text{If } k = 0 \text{ then } \phi^0 = 1 = 0.\phi + 1 = \mathrm{F}_0 + \mathrm{F}_{-1}.$$

Inductive Step($k \geq 0$)

Assume $\phi^k = \mathrm{F}_k.\phi + \mathrm{F}_{k-1}$ for $k = 0, \ldots, n$

$$\begin{aligned}\phi^{n+1} &= \phi.\phi^n\\ &= \phi.(\mathrm{F}_n.\phi + \mathrm{F}_{n-1}) && \text{By inductive hypothesis}\\ &= \mathrm{F}_n.\phi^2 + \mathrm{F}_{n-1}.\phi\\ &= \mathrm{F}_n.(\phi + 1) + \mathrm{F}_{n-1}.\phi\\ &= (\mathrm{F}_n + \mathrm{F}_{n-1}).\phi + \mathrm{F}_n\\ &= \mathrm{F}_{n+1}.\phi + \mathrm{F}_n && \text{By definition of } F_{n+1}\end{aligned}$$

Inductive Step($k \leq 0$)
Assume $\phi^k = \mathrm{F}_k.\phi + \mathrm{F}_{k-1}$ for $k = 0, \ldots, -n$

$$\begin{array}{lll}
\phi^{-n-1} & = \frac{\phi^{-n}}{\phi} & \\
 & = \frac{1}{\phi}.(\mathrm{F}_{-n}.\phi + \mathrm{F}_{-n-1}) & \text{By inductive hypothesis} \\
 & = \mathrm{F}_{-n} + (\phi - 1).\mathrm{F}_{n-1} & \text{Since } \frac{1}{\phi} = \phi - 1 \\
 & = \mathrm{F}_{-n} + \mathrm{F}_{-n-1}.\phi - \mathrm{F}_{-n-1} & \\
 & = \mathrm{F}_{-n-1}.\phi + (\mathrm{F}_{-n} - \mathrm{F}_{-n-1}) & \\
 & = \mathrm{F}_{-n-1}.\phi + \mathrm{F}_{-n-2} & \text{Since } \mathrm{F}_{-n-2} = \mathrm{F}_{-n} - \mathrm{F}_{-n-1}
\end{array}$$

2.4 Representing $\frac{1}{2^k}$ in GR notation

In this section we derive a representation of the values of the form $\frac{1}{2^k}$ in Golden Ratio notation. Obviously this is possible, since we can represent any value $[0, \phi]$ in simplified notation. We will see that:

$$\frac{1}{2^k} = [\![\alpha(\beta)^\omega]\!]_s$$

where $\alpha, \beta \in \mathbf{2}^*$, meaning that we can write any value of the form $\frac{1}{2^k}$ as a prefix (α) and a finite part(β) that is repeated infinitely. This is quite a surprising result, because we are dealing with values in base ϕ, and there is no reason to think that it is possible to write binary fractions in a periodic form.

We start with another surprising result, which we will then prove:

Proposition 1

$$\forall k \in \mathbb{N}, \exists l, m \in \mathbb{N}, \text{ such that } \sum_{i=1}^{2^k} \frac{1}{\phi^{3i-2}} = 2^k.(l.\phi + m)$$

Before we prove this we will consider the relation between ϕ^{-p} and ϕ^{-p-3}. Assume $\phi^{-p} = a.\phi + b$ for some a and b that are integers:

$$\begin{aligned}
\phi^{-p} &= a.\phi + b \\
\phi^{-p-1} &= b.\phi + (a - b) \\
\phi^{-p-2} &= (a - b).\phi + (2b - a) \\
\phi^{-p-3} &= (2b - a).\phi + (2a - b)
\end{aligned}$$

We use here the the identity that $\phi^k = F_k\phi + F_{k-1}$. Looking at the coefficients of ϕ^{-k} and ϕ^{-k-3} we can say the following:

If		Then	
a	b	$(2b - a)$	$(2a - b)$
odd	*odd*	*odd*	*odd*
even	*odd*	*even*	*odd*
odd	*even*	*odd*	*even*
even	*even*	*even*	*even*

Therefore if $\phi^p = a.\phi + b$ and $\phi^{p-3} = c.\phi + d$ then a and c both either odd or even, similarly b and d are both either odd or even. In particular since $\phi^1 = \phi$ then $\phi^{3p} = a.\phi + b$ always has the property that a is odd and b is even, when $p \leq 0$. We can repeat the same argument with ϕ^i and ϕ^{i+3}, to get a similar result for $i \in \mathbb{Z}$.

Proof of Proposition 1

We will use a proof by induction on $k \geq 0$ to prove this.

Base Case

If $k = 0$, then we have:

$$\begin{aligned}\textstyle\sum_{i=1}^{2^0} \frac{1}{\phi^{3i-2}} &= \frac{1}{\phi} \\ &= \phi - 1 && \text{since } \phi^2 = \phi + 1 \\ &= 2^0.(l.\phi + m) \text{ where } l = 1, m = -1.\end{aligned}$$

Inductive Step

Inductive Hypothesis — Assume

$$\exists l, m \in \mathbb{Z} \text{ such that } \sum_{i=1}^{2^k} \frac{1}{\phi^{3i-2}} = 2^k.(l.\phi + m),\ \text{ for } k = 0, \ldots, n.$$

We want to show $\sum_{i=1}^{2^{n+1}} \frac{1}{\phi^{3i-2}} = 2^k.(l.\phi + m),$ for some integers l and m.

We know by our Inductive Hypothesis $\sum_{i=1}^{2^n} \frac{1}{\phi^{3i-2}} = 2^n.(l.\phi+m),$ for some integers l and m.

Therefore:

$$\begin{aligned}\textstyle\sum_{i=1}^{2^{n+1}} \frac{1}{\phi^{3i-2}} &= \textstyle\sum_{i=1}^{2^n} \frac{1}{\phi^{3i-2}} + \sum_{i=2^n+1}^{2^{n+1}} \frac{1}{\phi^{3i-2}} \\ &= \textstyle\sum_{i=1}^{2^n} \frac{1}{\phi^{3i-2}} + \frac{1}{\phi^{3n}}.\sum_{i=1}^{2^n} \frac{1}{\phi^{3i-2}} \text{ ,Since } \sum_{i=1}^{2^n} \frac{1}{\phi^{3(2^n+i)-2}} = \frac{1}{\phi^{3.2^n}}.\sum_{i=1}^{2^n} \frac{1}{\phi^{3i-2}} \\ &= 2^n.(l.\phi + m) + \tfrac{1}{\phi^{3n}}.(2^n.(l.\phi + m)), \text{ By inductive hypothesis} \\ &= 2^n.(l.\phi + m).(1 + \tfrac{1}{\phi^{3n}}) \\ &= 2^n.(l.\phi + m).(1 + a.\phi + b), \text{ where } a \text{ is even, and } b \text{ is odd.} \\ &= 2^{n+1}.(l.\phi + m).(c.\phi + d), \text{ where } c = a/2 \text{ and } d = (b+1)/2. \\ &= 2^{n+1}.((c.l + c.m + d.l).\phi + (m.d + c.l))\end{aligned}$$

Note that c and d are integers since we know that a was even and that b was odd. □

By noticing that $\mathrm{F}_i = b$ where $\phi^i = a.\phi + b$. We can then define the Fibonacci numbers as above by ignoring the ϕ part of the theorem. We can show that:

$$\forall k \in \mathbb{N}, \exists m \in \mathbb{Z} s.t. \sum_{i=1}^{2^k} \mathrm{F}_{3i-2} = 2^k.m$$

Also we can generalise this and say that:

$$\forall j \in \mathbb{Z}, k \in \mathbb{N}, \exists m \in \mathbb{Z} s.t. \sum_{i=1}^{2^k} \mathrm{F}_{3i+j} = 2^k.m,\ \text{ for some integer } m.$$

This follows from the following identities:

$$\begin{aligned}\textstyle\sum_{i=1}^{2^k}\frac{1}{\phi^{3i+j}} &= \textstyle\frac{1}{\phi^{j+2}}.\sum_{i=1}^{2^k}\frac{1}{\phi^{3i-2}}\\ &= \textstyle\frac{1}{\phi^{j+2}}.2^k.(l.\phi+m)\\ &= 2^k.(a.\phi+b)(l.\phi+m),\ \text{ where } \phi^{-j-2}=a.\phi+b\\ &= 2^k.((a.l+a.m+b.l).\phi+(m.b+a.l))\end{aligned}$$

We can then ignore the ϕ part and get the desired result.

This result may seem obvious. However at closer examination there is no reason for why this should hold. It is not known by the author if this has been observed before.

At the beginning of this section we wanted to find an algorithm, that would give $\frac{1}{2^k}$ in the form $[\![\alpha(\beta)^\omega]\!]_s$. To give the algorithm of how we might be able to calculate α and β, we consider the following Proposition. Then we will see how this applies to the algorithm.

Proposition 2 $\forall k, \exists \alpha_1, \ldots, \alpha_{3.2^k+1} \in \mathbf{2},$ *such that*

$$\frac{1}{2^k}.(\frac{1}{\phi}+\frac{1}{\phi^4}+\ldots+\frac{1}{\phi^{3.2^k+1}}) = \sum_{i=1}^{3.2^k+1}\frac{\alpha_i}{\phi^i}$$

Proof

This can be seen as an extension to Proposition 1. From the way that we defined $\phi^p = a\phi + b$ above we can see that the difference between a and b are terms in the Fibonacci sequence. In particular $a - b = \mathrm{F}_{p-2}$.

We will consider the gaps between the a's and b's to show that if:

$\forall k, \exists \alpha_1, \ldots \in \mathbf{2},$ such that

$$\frac{1}{2^k}.(\frac{1}{\phi}+\frac{1}{\phi^4}+\ldots+\frac{1}{\phi^{3.2^k}}) = \sum_{p=3.2^k+1}^{\infty}\frac{\alpha_p}{\phi^p} \text{ then } \alpha_i = 0, \forall i \geq 3.2^k+1 \in \mathbb{N}$$

This is an equivalent definition of the proposition. We will prove this by contradiction: Assume that $\exists i$ such that $\alpha_i = 1$. In this case $\frac{\alpha_i}{\phi^i} = a\phi + b$ with $a - b = \mathrm{F}_{i-2}$. Define property Gap$(k) = a - b$ where $\phi^k = a.\phi + b$. We will show that:

$$|Gap(\frac{1}{2^k}(\frac{1}{\phi}+\frac{1}{\phi^4}+\ldots+\frac{1}{\phi^{3.2^k+1}}))| < |Gap(\frac{1}{\phi^{3.2^k}})| \leq |Gap(\sum_{p=3.2^k+2}^{\infty}\frac{\alpha_p}{\phi^p})| \quad (2.4)$$

And therefore all $\alpha_i = 0, \forall i \geq 3.2^k + 1$.

Proof

Proof of first part of 2.4

We want to show:

$$|Gap(\frac{1}{\phi}+\frac{1}{\phi^4}+\ldots+\frac{1}{\phi^{3.2^k+1}})|<|Gap(\frac{1}{\phi^{3.2^k+1}})| \tag{2.5}$$

Proof by induction on k
Base Case, $k=0$

$$|Gap(\frac{1}{\phi}+\frac{1}{\phi^4})|=|Gap(-2\phi+4)|=6<8=|Gap(\frac{1}{\phi^4})|$$

Step Case

Assume 2.5 holds for $k=1,\ldots,n$. We will show this holds for $n+1$. We use the fact that $|Gap(1+\frac{1}{\phi^{3.2^k}})|<|Gap(\frac{1}{\phi^{3.2^k}})|$ since $\forall i\in\mathbb{N}\exists a,b\in\mathbb{N}$ such that $\frac{1}{\phi^{3.i}}=a\phi-b<a\phi-(b-1)$ so $|Gap(1+\frac{1}{\phi^{3.2^k}})|=a-b<a-b+1=|Gap(\frac{1}{\phi^{3.2^k}})|$

$$\begin{aligned}
|Gap(\tfrac{1}{\phi}+\tfrac{1}{\phi^4}+\ldots+\tfrac{1}{\phi^{3.2^k+1}})| &= |Gap(\textstyle\sum_{i=0}^{2^{n+1}}\tfrac{1}{\phi^{3.i+1}})| \\
&= |Gap(\textstyle\sum_{i=0}^{2^n}\tfrac{1}{\phi^{3.i+1}}+\sum_{i=2^n+1}^{2^n+2^n}\tfrac{1}{\phi^{3.i+1}})| \\
&= |Gap(\textstyle\sum_{i=0}^{2^n}\tfrac{1}{\phi^{3.i+1}}+\tfrac{1}{3.2^n}.(\sum_{i=0}^{2^n}\tfrac{1}{\phi^{3.i+1}}))| \\
&= |Gap(\tfrac{1}{3.2^n+1}+\tfrac{1}{3.2^n}.\tfrac{1}{3.2^n+1}) && \text{By inductive hypothesis} \\
&= |Gap(\tfrac{1}{3.2^n+1}(1+\tfrac{1}{\phi^{3.2^n}}))| \\
&= |Gap(\tfrac{1}{3.2^n+1}).Gap(1+\tfrac{1}{\phi^{3.2^n}})| \\
&< |Gap(\tfrac{1}{3.2^n+1}).Gap(\tfrac{1}{\phi^{3.2^n}})| && \text{By above} \\
&= |Gap(\tfrac{1}{3.2^n+1}.\tfrac{1}{\phi^{3.2^n}})| \\
&= |Gap(\tfrac{1}{3.2^{n+1}+1})|
\end{aligned}$$

Proof of second part of 2.4

$$|Gap(\frac{1}{\phi^{3.2^k}})|\leq|Gap(\sum_{p=3.2^k+2}^{\infty}\frac{\alpha_p}{\phi^p})| \tag{2.6}$$

Here we are trying to minimise the value of $|Gap(\sum_{p=3.2^k+2}^{\infty}\frac{\alpha_p}{\phi^p})|$, we know that $Gap(\phi^p)=F_{p-2}$. Therefore, to minimise this (without making all the α_i's=0), we need to find the value of p that is closest to 0. Clearly from equation 2.1 we can make this sum $=\frac{1}{\phi^{3.2^k}}$ by making all the $\alpha_i=1$ for $\forall i\geq 3.2^k+2$. Therefore:

$$\begin{aligned}
|Gap(\tfrac{1}{2^k}.(\tfrac{1}{\phi}+\tfrac{1}{\phi^4}+\ldots+\tfrac{1}{\phi^{3.2^k+1}}))| &\leq |Gap(\tfrac{1}{\phi}+\tfrac{1}{\phi^4}+\ldots+\tfrac{1}{\phi^{3.2^k+1}})| && \text{Obvious} \\
&< |Gap(\tfrac{1}{\phi^{3.2^k+1}})| && \text{By 2.5} \\
&\leq |Gap(\textstyle\sum_{p=3.2^k+2}^{\infty}\tfrac{\alpha_p}{\phi^p})| && \text{By 2.6}
\end{aligned}$$

Hence $\forall i\geq 3.2^k+2, \alpha_i=0$ since if this was not the case then we can show that the smallest gap on the left side is greater than the largest gap of the right side. Therefore we have proved Proposition 2. $\square$

Now that we have proved Proposition 2 we need to now show that this can be used to construct the algorithm. To motivate this algorithm we will first show that $\frac{1}{2} = [\![0(100)^\omega]\!]_s$

We have that $2.[\![0(100)^\omega]\!]_s = 1$, because

$$\begin{array}{rll} 2.[\![0(100)^\omega]\!]_s & = \frac{2}{\phi^2}(\sum_{i=0}^{\infty} \frac{1}{\phi^{3.i}}) & \\ & = \frac{1}{\phi^2}(\sum_{i=0}^{\infty} \frac{2}{\phi^{3.i}}) & \\ & = \frac{1}{\phi^2}(\sum_{i=0}^{\infty} \frac{1}{\phi^{i}}) & \text{Since } 2 = 1 + \frac{1}{\phi} + \frac{1}{\phi^2} \\ & = \frac{1}{\phi}(\sum_{i=1}^{\infty} \frac{1}{\phi^{i}}) & \\ & = 1 & \text{From 2.1} \end{array}$$

Now we can write the periodic part of $\frac{1}{2}$ as $\frac{1}{\phi}$(=100 in GR notation). To find the periodic part of $\frac{1}{4}$ we cannot directly apply Proposition 2 since it is not of the correct form, but $(100)^\omega = (100100)^\omega$, the recursive part of which can be written $\frac{1}{\phi} + \frac{1}{\phi^4}$ (=100100 in GR notation). Now we can apply Proposition 2 with $k = 1$:

$$\frac{1}{2}.(\frac{1}{\phi} + \frac{1}{\phi^4}) = \frac{\alpha_1}{\phi} + \frac{\alpha_2}{\phi^2} + \frac{\alpha_3}{\phi^3} + \frac{\alpha_4}{\phi^4}$$

This means we can write:

$$\frac{1}{4} = [\![0(\alpha_1\alpha_2\alpha_3\alpha_4)^\omega]\!]_s$$

We can now use the same Proposition to find the periodic parts of $\frac{1}{8}, \frac{1}{16}, \ldots$. In fact we can use the function that divides by 2, applying this to the finite recursive, part is guaranteed to terminate by Proposition 2.

We have implemented the algorithm described above. The main problem with the implementation is that the periodic part of the number doubles in size after each step, so for large values of k it may be impractical to calculate $\frac{1}{2^k}$.

However the main point is that it is possible to do. This is an interesting result as it is not immediately obvious that all powers of $\frac{1}{2^k}$ can be written in a periodic form. Unfortunately, although this result is interesting, we have not found a practical application for this in this project.

3. Design and Implementation

The mathematically inclined reader can safely skip the computational technical details discussed in this chapter, as in the following chapters we write our algorithms and pseudocode that is close to mathematical notation.

3.1 Lazy Evaluation Programming Languages

Before we start the implementation we must decide, what programming language to use, and also how to represent streams in this language. Algorithms using infinite lists as a definition for infinite streams have been successfully implemented in Haskell by David Plume [14].

The definition of a lazy evaluation language is that an argument is only evaluated when it has to be. Haskell is similar to ML, the only major difference being that Haskell is a lazy evaluation language, and ML is not. For example take the function from defined as follows:

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

If we ask for `from 0;` in Haskell we get the infinite stream of digits `[0,1,2,3,4,5,` .... To stop we have to type `<ctrl-c>`. In ML we would get no output. It loops forever producing no output. The reason for this is that when we ran the program in Haskell, the argument `from (n+1)` was not executed until we have returned `n`. In this way we will construct an infinite stream outputting each digit as we find it. If we run the same code in ML, we will not get any output because although we know `n`, we will first execute `from (n+1)` and then add `n` onto the result. The problem is that `from (n+1)` will not terminate and therefore will never output any digits.

The advantages of using Haskell is that we could represent infinite streams as infinite lists. If we did this we could use pattern matching to split the input into specific cases. Being able to use the list structure to both deconstruct and construct a stream would make the code readable and therefore easier to debug.

However using infinite lists in a lazy evaluation language means that we are not able to have side effects. Unfortunately the integration algorithm that we will implement makes essential use of side effects.

Instead we will use ML because it allows side effects. Implementing in ML means that we will not be able to represent infinite streams as infinite lists.

We can define a list like structure that delays the evaluation of the rest of the stream by introducing an anonymous function. This will stop the evaluation of the remaining digits until we force them to be evaluated. The stream type and the force function are defined as follows:

```
datatype 'a stream = cons of 'a * (unit -> 'a stream);
```

and also

```
fun force s = s();
```

This defines an 'a stream as a constructor containing the next digit in the stream, and also a function. Forcing this function (using force) will give the next 'a stream, which contains the next digit of the stream, and also a function that returns the next 'a stream, and so on. Thus the purpose of the use of the unit type is to delay evaluation.

Writing a stream using this delayed list method is not the only way that we could implement this. However the reason we are implementing this method is that we want to keep the algorithms as listlike as possible.

This representation causes the following problem. We cannot use pattern matching for more than the first digit, since to get the next digit we would have to execute a function (we cannot execute functions whilst pattern-matching). To solve this we need to deconstruct the list in a `let ...in ...end` statement then have individual `if ...then ...else` statements to separate the algorithm into cases. We illustrate this by considering th ML code that will applies the identity to the first elements of a stream:

```
fun identity(cons(digit_1, stream)) =
    let
        val cons(digit_2, stream_2) = (force stream)
        val cons(digit_3, stream_3) = (force stream_2)
    in
        if (digit_1=1) andalso (digit_2=0) andalso (digit_3=0) then
            cons(0, fn()=> cons(1, fn() => cons(1, stream_3)))
        else if (digit_1=0) andalso (digit_2=1) andalso (digit_3=1) then
            cons(1, fn()=> cons(0, fn() => cons(0, stream_3)))
             else
                 cons(digit_1, stream)
    end;
```

In Haskell the code for the same function is:

```
fun identity(1::0::0::list) = 0::1::1::list
  | identity(0::1::1::list) = 1::0::0::list
  | identity(list) = list;
```

We can see that the Haskell code is much easier to understand, and would be easier to debug. Also it is clear that Haskell code is much shorter. However we need to implement the following algorithms by using ML, because the integration algorithm that we will be implementing will need to use a language which supports side effects. Unfortunately Haskell doesn't. The pseudo code of the algorithms that we will use will be in a Haskell form. This is because it is much easier to understand compared to if the code was written as it appears in the actual calculator. Notice also that it is closer to mathematical notation.

3.2 Algorithms Developed and Implemented

I have designed, proved and implemented the following algorithms:

- Flip function on a finite list.
- New algorithms for addition and multiplication by 2.
- New algorithm for multiplication.
- Conversion from Decimal to GR notation.
- Conversion from GR notation to Decimal.
- Conversion from GR notation to Signed Binary.
- Lexicographical Normalisation.
- Cases function.

Also I have re-designed, proved and implemented the following algorithms:

- Basic Operations.
- Intersection of nested sequences of intervals.
- Functions for calculating $e^x, \sin(x), \cos(x), \log(x)$ and x^y
- Definite Integration.

An explanation of each of the algorithms together with a proof that they work is given in the following chapters.

4. Manipulations of finite and infinite streams

4.1 Flip function for finite lists

Firstly we will motivate the definition of the flip function. If we want to construct the natural numbers in simplified GR notation. We can do this inductively. We know that $1 = [\![1]\!]$. By repeatedly applying the identity, we can turn digit representing $\phi^0 = 1$ to make it 0, then we can add one (by making this digit 1). Repeating this will give us the natural numbers. For example:

$$\begin{array}{ll} 1 & = [\![1.0]\!] = [\![0.11]\!] \\ 2 & = [\![1.11]\!] = [\![10.01]\!] \\ 3 & = [\![11.01]\!] = [\![100.01]\!] \\ 4 & = [\![101.01]\!] = [\![100.1111]\!] \\ 5 & = [\![101.1111]\!] = [\![110.0111]\!] \\ \vdots & \vdots \end{array}$$

This example suggests that it is possible to make the units digit, of a GR number 0 by applying appropriate identities. But it is not clear that this is true in general. If it is true then is there a function that can change any digit 1 to a digit 0 by applying appropriate identities (and possibly changing other digits, if necessary).

The flip function manipulates a finite stream of digits, in such a way as to set a specified digit to 0. In this section we will look at how this function works, see why we cannot extend this to infinite streams and discuss the possible applications for this.

We can define the flip function as follows, given a digit d and α, where α has finite length k with $\alpha_1 = 0, \alpha_{k-1} = 0$ and $\alpha_k = 0$. Then we require that flip(α, d) = β where β, for α, β with $[\![\alpha]\!]_s = [\![\beta]\!]_s$, and $\beta_d = 0$.

Proposition 3 *There exists a computable function flip* : $\mathbf{2}^* \to \mathbf{2}^*$*, that will satisfy the following,* $[\![\alpha]\!]_s = [\![\beta]\!]_s$*, where* $\beta =$ *flip*(α, d) *and* $\beta_d = 0$ *given that* $\alpha = 0\alpha'00$*, for some* $\alpha' \in \mathbf{2}^*$*.*

Proof

To prove the proposition we will first define the function flip and then prove that this will give the correct output.

We define flip, as follows:

$$\begin{aligned}
\text{flip}(01^n0\alpha', d) &\Rightarrow & 01^n\text{flip}(0\alpha', d-n-1) & \quad \text{if } d > n+1 & (4.1)\\
\text{flip}(01^n\alpha', d) &\Rightarrow & (10)^p01^{n-d+1}\alpha' & \quad \text{if } d \leq n+1, d-1 = 2p & (4.2)\\
\text{flip}(01^n\alpha', d) &\Rightarrow & (10)^{p+1}01^{n-d}\alpha' & \quad \text{if } d < n+1, d-1 = 2p+1 & (4.3)\\
\text{flip}(01^n\alpha', d) &\Rightarrow & (10)^p\text{flip}(010\alpha', 2) & \quad \text{if } d = n+1, d-1 = 2p+1 & (4.4)
\end{aligned}$$

We will now prove that flip satisfy the above conditions, by induction on the size of k. At each step of the induction we will assume α begins with 0 and ends with 00.

Two facts are used in this proof:

- By induction on n, repeatedly using the identity $\phi^2 = \phi + 1$, we can show $[\![0(11)^n]\!]_s = [\![(10)^n0]\!]_s$.
- We can think of a finite stream of digits as a sequence of smaller streams of the form 01^n. Therefore we can write α as $\alpha = 01^{n_1}01^{n_2}\dots01^{n_k}00$.

The base case of this induction is when $k = 1$. In this case $\alpha = 01^n00$, if d is $1, n+2, n+3$ or $n = 0$ then we can return α. If $d = n+1$ then by applying the identity once we can get $[\![01^n00]\!]_s = [\![01^{n-1}011]\!]_s$, this is now of the correct form. Now we have that $\alpha = 0(11)^l1^{n-2l}00$ or $\alpha = 0(11)^l1^{n-2l-1}00$ where l is defined so that $d = 2l+1$ or $d = 2l$ respectively. Note that one of the last 2 digits of 01^{2l} is the d th digit of α. Now by applying the above lemma to $0(11)^l$ giving $(10)^l0$. Obviously the last 2 digits of this are 0. Therefore $\alpha = (10)^l01^{n-2l}00$ or $\alpha = (10)^0l1^{n-2l-1}00$. This is now of the correct form.

Proof of flip function

Inductive hypothesis:

Assume $\forall i \leq p, \alpha = 01^{n_1}01^{n_2}\dots01^{n_i}00$ can be written using the flip function as β where $\beta_l = 0, [\![\alpha]\!]_s = [\![\beta]\!]_s$.

Proof of 4.1

By our inductive hypothesis if flip$(0\alpha', d-n-1)$ returns β' with $\beta'_{d-n-1} = 0$ then flip($01^n0\alpha', d$) will return β with $\beta_d = 0$, notice that if α ends in 00 then α' will also. It is obvious that if $[\![\beta']\!]_s = [\![\alpha']\!]_s$ then $[\![01^n\beta']\!]_s = [\![01^n\alpha']\!]_s$.

Proof of 4.2

If $d \leq n+1$ then we have reached the section with the digit that we want to change. If $d-1$ is even then we can write $[\![01^n\alpha']\!]_s = [\![0(11)^p1^{n-d+1}\alpha']\!]_s = [\![(10)^p01^{n-2p}\alpha']\!]_s = [\![(10)^p01^{n-d+1}\alpha']\!]_s$ note that $|(10)^p0| = d$ and therefore the d'th digit this is 0.

Proof of 4.3 If $d-1$ is odd, and $d < n+1$ then we can write $[\![01^n\alpha']\!]_s = [\![0(11)^p1^{n-d}]\!]_s = [\![(10)^{p+1}01^{n-d}\alpha']\!]_s$, notice that if $d = n+1$ then this is not true. We can see that $|(10)^{p+1}| = 2p+2 = d$ and therefore the d'th digit of the above expression is 0.

Proof of 4.4

If $d = n + 1$ and $d - 1$ is odd then we can write $[\![01^d 0\alpha']\!]_s = [\![0(11)^p 10\alpha']\!]_s = [\![(10)^p 010\alpha']\!]_s$, the d'th digit of this is the second digit of this $010\alpha'$, if α' ended in 00 then this will end in 00. Trivially this begins with 0. Therefore if flip$(010\alpha', 2) = \beta$, with $\beta_d = 0$. Then $[\![\beta']\!]_s = [\![010\alpha']\!]_s$ and $[\![(10)^p\beta']\!]_s = [\![01^n\alpha']\!]_s$. □

Here is a useful property that we can exploit using the flip function:

$$\begin{array}{rrcl} & [\![0.0(1)^\omega]\!]_s & = & [\![1.00(0)^\omega]\!]_s \\ \text{Therefore} & [\![0.0(1)^*0(1)^\omega]\!]_s & < & [\![1.00(0)^\omega]\!]_s \end{array}$$

This still holds if we replace ω with $*$. So if we have a stream of the form $0.0(1)^*0(1)^\omega$, for example $0.0(1)^\omega$ but with at least one of the 1's 0, then we can say that this will be less than one. We can now apply the flip function on the first digit of the finite stream α as follows.

$$\begin{array}{llllllllll} \text{convert from} & 0 & \cdot & \alpha_1 & \alpha_2 & \alpha_3 & \ldots & \alpha_n & 0 & 0 \\ \text{to} & \alpha'_0 & \cdot & 0 & \alpha'_2 & \alpha'_3 & \ldots & \alpha'_n & \alpha'_{n+1} & \alpha'_{n+2} \end{array}$$

Now α'_0 determines if α is less than or greater than or equal to 1 by the value α'_0 (if it is 0 then $\alpha < 1$, if it is 1 then $\alpha \geq 1$).

We can prove that the flip function on a finite part of a stream is the best that we can guarantee to do.

Proposition 4 *There is no computable function flip* : $\mathbf{2}^\omega \to \mathbf{2}^\omega$.

Proof

We show that if this was not the case then we would be able to construct a function that converts from GR notation to unsigned binary notation. Assuming that we are given a stream α, in simplified notation the algorithm that we could use is described as follows. First we find $2 \times [\![\alpha]\!]_s$. Then we can apply the flip function as described above to determine if the value of this stream is ≥ 1 or < 1. If the list is ≥ 1 then the original value of the list was ≥ 0.5, therefore the next digit of the binary answer is 1. We now have to subtract this from our stream, to cancel out the possible effect of taking out the digit 1. This is easy because in this case the first digit in our stream would be 1. In the case that the list is < 1 then we can say that the next digit of the output would be a 0. □

Therefore this algorithm cannot be extended to use infinite streams of input, because to make a digit that is 1, into 0 we have to at some point apply the identity. But if the stream is of the form $(01)^\omega$ we will never be able to apply the identity, and applying the flip function would result in us looping forever. Therefore the flip function is not of finite character. The cases where the flip function will loop are relatively small, for example: problem cases are of the form $0\alpha(01)^\omega$ where $\alpha \in \mathbf{2}^*$, $|\alpha| = n$ and $d = n + 2$.

The following functions all use the flip function: conversion to Decimal from GR notation, conversion from any integer to a finite GR notation, conversion from decimal to GR notation and integration. Also we implemented a function using this flip function that can convert any integer into its GR notation.

4.2 Lexicographical Normalisation

In this section we try to find a way of determining if $[\![\alpha]\!]_s < [\![\beta]\!]_s$. Unfortunately if $\alpha \equiv \beta$ we will never terminate, because to verify this we would have to check that each of the digits in each stream are the same. This is not possible since the steams are infinite. Therefore we can only find $\alpha <_\perp \beta$ where $<_\perp$ is defined as follows:

$$\alpha <_\perp \beta = \begin{cases} \text{true} & \text{if } [\![\alpha]\!]_s < [\![\beta]\!]_s \\ \text{false} & \text{if } [\![\alpha]\!]_s > [\![\beta]\!]_s \\ \perp & \text{if } [\![\alpha]\!]_s = [\![\beta]\!]_s \end{cases}$$

$\perp$ means non-termination.

A naive algorithm computing $\alpha <_\perp \beta$ is as follows:

$$\alpha <_\perp \beta = \begin{cases} \text{true} & \text{if } \exists k \text{ such that } \alpha_i = \beta_i \forall i = 1 \ldots k \text{ and } \alpha_{k+1} < \beta_{k+1} \\ \text{false} & \text{if } \exists k \text{ such that } \alpha_i = \beta_i \forall i = 1 \ldots k \text{ and } \alpha_{k+1} > \beta_{k+1} \\ \perp & \text{if } \alpha_i = \beta_i \forall i \in \mathbb{N} \end{cases}$$

This algorithm works when dealing with notations normally used, for example decimal, or binary, but does not work for GR numbers, for example:

$$\begin{array}{l} \alpha = 0.1000000000\ldots, [\![\alpha]\!]_s = \frac{1}{\phi} \\ \beta = 0.0111111111\ldots, [\![\beta]\!]_s = 1 \end{array}$$

Here we can see $[\![\alpha]\!]_s > [\![\beta]\!]_s$ but using the $<_\perp$ algorithm we would get that $\alpha <_\perp \beta$.

However Escardó [5] shows that if we lexicographically normalise two streams in an appropriate sense, then if the above algorithm terminates, it will always return the correct result.

Lexicographical normalisation on SB streams was defined for infinite streams of SB digits. Since the identities in SB and GR notation are different the same algorithm would not work in GR notation, although the method is similar. We have designed, proved and implemented an algorithm for lexicographical normalisation in GR notation.

Proposition 5 *There exists a computable function lex*: $2^\omega \times 2^\omega \to 2^\omega \times 2^\omega$, *such that if lex(*$\alpha, \beta$*)=(*$\alpha', \beta'$*) then* $[\![\alpha]\!]_s = [\![\alpha']\!]_s$ *and* $[\![\beta]\!]_s = [\![\beta']\!]_s$ *and if* $[\![\alpha]\!]_s < [\![\beta]\!]_s$ *then* $\alpha' <_\perp \beta'$, *and if* $[\![\alpha]\!]_s > [\![\beta]\!]_s$ *then* $\alpha' >_\perp \beta'$

Proof

We will define the algorithm for lex as follows:

$$\text{lex}(a :: \alpha, a :: \beta) \;\Rightarrow\; a :: \text{lex}(\alpha, \beta) \tag{4.5}$$

$$\text{lex}(1 :: \alpha, 0 :: \beta) \;\Rightarrow\; \text{swap}(\text{lex}(0 :: \beta, 1 :: \alpha)) \tag{4.6}$$

$$\begin{aligned}
\mathrm{lex}(0::0::a_2::\alpha, 1::1::b_2::\beta) &\Rightarrow (0::0::a_2::\alpha, 1::1::b_2::\beta) &(4.7)\\
\mathrm{lex}(0::n::0::\alpha, 1::n::b_2::\beta) &\Rightarrow (0::n::0::\alpha, 1::n::b_2::\beta) &(4.8)\\
\mathrm{lex}(0::n::a_2::\alpha, 1::n::1::\beta) &\Rightarrow (0::n::a_2::\alpha, 1::n::1::\beta) &(4.9)\\
\mathrm{lex}(0::1::1::\alpha, 1::1::0::\beta) &\Rightarrow 1::\mathrm{lex}(0::0::\alpha, 1::0::\beta) &(4.10)\\
\mathrm{lex}(0::0::1::\alpha, 1::0::0::\beta) &\Rightarrow 0::\mathrm{lex}(0::1::\alpha, 1::1::\beta) &(4.11)\\
\mathrm{lex}(0::1::1::\alpha, 1::0::1::\beta) &\Rightarrow 1::0::\mathrm{lex}(0::\alpha, 1::\beta) &(4.12)\\
\mathrm{lex}(0::1::0::\alpha, 1::0::0::\beta) &\Rightarrow 0::1::\mathrm{lex}(0::\alpha, 1::\beta) &(4.13)\\
\mathrm{lex}(0::1::1::\alpha, 1::0::0::\beta) &\Rightarrow 1::0::0::\mathrm{lex}(\alpha, \beta) &(4.14)\\
\mathrm{lex}(0::1::0::\alpha, 1::0::1::\beta) &\Rightarrow (0::1::0::\alpha, 1::0::1::\beta) &(4.15)
\end{aligned}$$

where $\mathrm{swap}(\alpha, \beta) = (\beta, \alpha)$.

In this algorithm when we add digits to the front of a call to lex, for example in rules 4.5, 4.10, 4.11, 4.12, 4.13 and 4.14, this means that we add a digit to both streams of the result:

$$\begin{aligned}
\text{If } \mathrm{lex}(\alpha, \beta) &= (\alpha', \beta'), \text{ then}\\
\mathrm{lex}(a::\alpha, a::\beta) &= a::\mathrm{lex}(\alpha, \beta) \text{ by rule 4.5}\\
&= a::(\alpha', \beta')\\
&= (a::\alpha', a::\beta')
\end{aligned}$$

An interesting practical problem that we encountered when implementing this particular part of the algorithm was that we could not use the map functions since we are using stream notation and not infinite list notation. The way that this would normally be done would be to evaluate the remaining part of the lex function in a **let ... in ... end** rule and then pattern match the answer. Doing this would cause the lex function to be evaluated, but because this works on an infinite stream this will never terminate. Another way of doing this would be to evaluate lex twice and each time pick one part of the result out. This works but is very inefficient, because at every step we will have to evaluate the lex function twice, not only is this inefficient because we are evaluating the same function twice, but each of these functions will then call the lex function. Therefore after n digits of output 2^n lex functions (all the same) may have to be evaluated. This is very inefficient since we should only need to evaluate one lex function to get the answer. We solved this problem by instead of using two streams with each part of the stream containing one digit. We use one stream, with each part containing a list of two digits. In practice we needed a third digit to deal with the swap part of the algorithm. After Lexicographical Normalisation we can convert this stream back into two separate streams, this now eliminates the problem of the number of processes exploding. Although to calculate the result this way we need to calculate lex twice, this is so that we can convert the result back into two streams of one digit.

Proof of lex algorithm

This algorithm is convergent since most of the rules produce at least one digit of output, some produce all future digits of output (when we can say $[\![\alpha]\!]_s < [\![\beta]\!]_s$) the only exception is the second rule, we can see that the next step of this algorithm then at least one digit of output will be produced.

The rules numbered 4.10, 4.11, 4.12, 4.13 and 4.14, all follow from applying the identity $\phi^2 = \phi + 1$ and then outputting digits that are in common with α and β.

In the rule 4.6, we will not output any digits but it is obvious that if $\mathrm{lex}(0 :: \beta, 1 :: \alpha)$ returns the correct answer then $\mathrm{lex}(1 :: \alpha, 0 :: \beta)$ will also.

In the remaining rules then we can defiantly say that $[\![\alpha]\!]_s < [\![\beta]\!]_s$, without looking at any more digits.

Proof of rule 4.7 In this rule we used the fact that $[\![00a_2\alpha]\!]_s < [\![11b_2\beta]\!]_s$, where $a_2, b_2 = 0$ or 1.

$$[\![00a_2\alpha]\!]_s \leq [\![001(1)^\omega]\!]_s = \frac{1}{\phi^2} + \frac{1}{\phi^3} < \frac{1}{\phi} + \frac{1}{\phi^2} = [\![110(0)^\omega]\!]_s \leq [\![11b_2\beta]\!]_s$$

Proof of rule 4.8 We use the fact that $[\![0n0\alpha]\!]_s < [\![1nb_2\beta]\!]_s$, we will now prove this.

$$[\![0n0\alpha]\!]_s \leq [\![0n0(1)^\omega]\!]_s = \frac{1}{\phi^2} + \frac{n}{\phi^2} < \frac{1}{\phi} + \frac{n}{\phi^2} \leq [\![0n0(0)^\omega]\!]_s \leq [\![0nb_2\beta]\!]_s$$

Proof of rule 4.9 We use the fact that $[\![0na_2\alpha]\!]_s < [\![1n1\beta]\!]_s$, we will now prove this.

$$[\![0na_2\alpha]\!]_s \leq [\![0n1(1)^\omega]\!]_s = \frac{1}{\phi^2} + \frac{1}{\phi^3} + \frac{n}{\phi^2} < \frac{1}{\phi} + \frac{1}{\phi^3} + \frac{n}{\phi^2} = [\![1n1(0)^\omega]\!]_s \leq [\![1n1\beta]\!]_s$$

Proof of rule 4.15 We use the fact that $[\![010\alpha]\!]_s < [\![101\beta]\!]_s$, we will now prove this.

$$[\![010\alpha]\!]_s \leq [\![010(1)^\omega]\!]_s = \frac{1}{\phi^2} + \frac{1}{\phi^2} = \frac{1}{\phi} + \frac{1}{\phi^4} < \frac{1}{\phi} + \frac{1}{\phi^3} \leq [\![101(0)^\omega]\!]_s \leq [\![101\beta]\!]_s$$

If the first digit in each stream are the same then we can just output this and repeat on the remainder of the stream. Otherwise by considering the upper and lower bounds of the remainder of the stream, we can either say α is not equal to β or α may still equal β. If α is not equal to β then we can make sure $<_\perp$ will terminate with the correct answer. Otherwise if α may still equal β, we use the identity to rewrite the digits such that at least the first digits in each stream are equal and then we repeat. □

For the above example, the algorithm would rewrite α as $\alpha = 0.0110000000\ldots$, and leave β unchanged. Now $\alpha <_\perp \beta$ will give the correct answer.

4.3 Cases function

Definition of Cases Function

As we saw in the above example it is impossible to determine if two infinite streams are equal, however it is possible to determine if two infinite streams are not equal.

Therefore as we might expect that it is not possible to define the following **if ... then ... else ...**, where the condition of the if part uses a conditional operator, like $<_\perp$. Since we cannot determine the case when the streams are equal.

However this is not always true. It was shown in Escardó [5], that it is possible to define the following function cases:

We assume that if $\alpha \equiv \delta$ then $\beta \equiv \gamma$.We will discuss why this assumption is necessary later.

$$\text{cases}(\alpha, \delta, \beta, \gamma) \equiv \begin{cases} \beta & \text{if } (\alpha <_\perp \delta) = \mathit{true}, \text{ or } \beta \equiv \gamma \\ \gamma & \text{if } (\alpha <_\perp \delta) = \mathit{false}, \text{ or } \beta \equiv \gamma \end{cases}$$

We will prove that this function is computable.

Proposition 6 *The cases function is computable, assuming that we have* $\alpha \equiv \delta \Rightarrow \beta \equiv \gamma$.

Proof

To prove this we will give the algorithm for calculating the cases function and then show that it satisfies the proposition.

The algorithm for the cases function is as follows:

Let $(\alpha', \delta') = \text{lex}(\alpha, \delta)$ and $(\beta', \gamma') = \text{lex}(\beta', \gamma')$.

$$\begin{aligned} \text{cases}(\alpha', \delta', \beta'_1 :: \beta'_{|2}, \gamma'_1 :: \gamma'_{|2}) &= \beta'_1 :: \text{cases}(\alpha', \delta', \beta'_{|2}, \gamma'_{|2}) && \text{if } \beta'_1 = \gamma'_1 \\ &= \beta'_1 :: \beta'_{|2} && \text{if } \beta'_1 \neq \gamma'_1 \text{ and } \alpha' <_\perp \delta' \\ &= \gamma'_1 :: \gamma'_{|2} && \text{if } \beta'_1 \neq \gamma'_1 \text{ and } \alpha' <_\perp \delta' = \mathit{false} \end{aligned}$$

We observe from our assumption that if $\alpha \equiv \delta$ then $\beta \equiv \gamma$, and if $\alpha \not\equiv \delta$ then $\beta \not\equiv \gamma$ (by negation of our assumption), then $\alpha <_\perp \delta$ is guaranteed to terminate.

The algorithm is basically as follows: Output the greatest common prefix of β and γ.

If this does not terminate, then $\beta \equiv \gamma$, and so irrespective of α and δ, the answer would be unchanged.

If this does terminate, then $\beta \not\equiv \gamma$, and so $\alpha \not\equiv \delta$, therefore $\alpha <_\perp \delta$ will terminate. We can then calculate this and depending on the answer we can output β or γ.

The proof that the algorithm will always give the correct output is as follows:

Proposition 7 *The cases algorithm will always give the correct answer.*

Proof

Case 1

Assume $\beta \equiv \gamma$ then the first rule of this algorithm will always be true since if $\beta \equiv \gamma$ then in particular $\beta_i = \gamma_i$, this follows from the definition of lexicographical normalisation. Therefore if $\beta \equiv \gamma$ then the output will be β which is exactly the same as γ.

Case 2

Assume $\beta \not\equiv \gamma$, then at some stage the first rule of this algorithm will not be satisfied, this follows again from the definition of lexicographical normalisation. Then by negation of the assumption if $\beta \not\equiv \gamma$ then $\alpha \not\equiv \delta$, now finding if $\alpha <_\perp \delta$ is guaranteed to terminate, by the definition of $<_\perp$. Therefore we calculate this and then return β or γ depending on this operation.

Therefore this algorithm will calculate the correct answer. □

If the assumption, if $\alpha \equiv \delta$ then $\beta \equiv \gamma$ does not hold then it is possible that $\beta \not\equiv \gamma$ and $\alpha \equiv \delta$, in this case we have a problem since the greatest common prefix of β and γ can only be of finite length, therefore at some point we will try to evaluate $\alpha <_\perp \delta$, however this will never terminate.

Applications of the Cases function

The cases function can be used to join together two different functions. The only prerequisite that we make is that at the point that we join the functions the values of the two functions are equal. Meaning that the new function is continuous, however it does not have to be continuously differentiable as we will see later.

We will consider four functions that we can implement using the cases function, these are: joining together two functions, finding the absolute value of a number, finding the maximum of a number and finding the minimum of a number.

Joining together two functions

If we have two functions $f(\alpha)$ and $g(\alpha)$ then assuming $f(\beta) = g(\beta)$, for some β, then we can join the two functions g and f at β, to make $h(\alpha)$ using:

$$h(\alpha) = \text{cases}(\alpha, \beta, f(\alpha), g(\alpha))$$

For absolute values we will use a specific case of this function, where $f(\alpha) = C'(\alpha), g(\alpha) = \alpha$ and $[\![\beta]\!]_f = 0$.

Absolute values

The absolute value of a number can be defined as follows:

Let *zero* be the stream defined such that $[\![zero]\!]_f = 0$.

$$\text{Abs}(\alpha) = \begin{cases} C'(\alpha) & \text{if } (\alpha <_\perp zero) = \mathit{true}, \text{ or } \alpha \equiv zero \\ \alpha & \text{if } (\alpha <_\perp zero) = \mathit{false}, \text{ or } \alpha \equiv zero \end{cases}$$

C' along with other basic operations will be defined in the next section. The effect of $C'(\alpha)$ is to negate the decimal value of α.

This definition of absolute values is analogous to the definition of absolute values for reals, it is obvious that this definition is correct. This can clearly be written as a particular case of the cases function, as follows:

$$\text{Abs}(\alpha) = \text{cases}(\alpha, zero, C'(\alpha), \alpha)$$

Maximum of two numbers

The maximum of two streams α and β can be defined as follows:

$$\text{Max}(\alpha, \beta) = \begin{cases} \beta & \text{if } (\alpha <_\perp \beta) = \mathit{true}, \text{ or } \alpha \equiv \beta \\ \alpha & \text{if } (\alpha <_\perp \beta) = \mathit{false}, \text{ or } \alpha \equiv \beta \end{cases}$$

This definition is analogous to the definition of Max with real numbers. It is clear that this defines the maximum of two streams.

We can write this equivalently as:

$$\text{Max}(\alpha, \beta) = \text{cases}(\alpha, \beta, \beta, \alpha)$$

Minimum of two numbers

The definition of Min is similar to that of Max, the definition of Min is:

$$\text{Min}(\alpha, \beta) = \begin{cases} \alpha & \text{if } (\alpha <_\perp \beta) = \mathit{true}, \text{ or } \alpha \equiv \beta \\ \beta & \text{if } (\alpha <_\perp \beta) = \mathit{false}, \text{ or } \alpha \equiv \beta \end{cases}$$

Therefore:

$$\text{Min}(\alpha, \beta) = \text{cases}(\alpha, \beta, \alpha, \beta)$$

5. Basic Operations

5.1 Implementation of Basic Operations

We have implemented the following basic operations: addition, negation, subtraction, multiplication and division.

In Pietro Di Gianantonio [6], the algorithms for calculating the functions addition, complement, subtraction, product and division are given the functions are called A, C, S, P and D respectively. Note a primed function means that it uses full notation:

$$\begin{array}{ll} [\![A(\alpha,\beta,a,b)]\!]_s = ([\![\alpha]\!]_s + [\![\beta]\!]_s + \frac{a}{\phi} + \frac{b}{\phi^2})/\phi^2, & [\![A'(z:\alpha,t:\beta)]\!]_f = [\![z:\alpha]\!]_f + [\![t:\beta]\!]_f \\ [\![C(\alpha)]\!]_s = \phi - [\![\alpha]\!]_s, & [\![C'(z:\alpha)]\!]_f = -[\![z:\alpha]\!]_f \\ [\![S'(z:\alpha,t:\beta)]\!]_f = [\![z:\alpha]\!]_f - [\![t:\beta]\!]_f & \\ [\![P(\alpha,\beta)]\!]_s = ([\![\alpha]\!]_s \times [\![\beta]\!]_s)/\phi^2, & [\![P'(z:\alpha,t:\beta)]\!]_f = [\![z:\alpha]\!]_f \times [\![t:\beta]\!]_f \\ [\![D(\alpha,\beta)]\!]_s = [\![\alpha]\!]_s/([\![\beta]\!]_s \times \phi), & [\![D'(z:\alpha,t:\beta)]\!]_f = [\![z:\alpha]\!]_f/[\![t:\beta]\!]_f \end{array}$$

Implementing these algorithms was non-trivial, since they were written with streams represented as infinite lists, and so they had to be rewrite them so that they used the new stream constructor.

Also we found a mistake in the proof of the function for product using full notation, the original proof of $P'(z:\alpha,t:\beta) \Rightarrow (z+t+2) : A(P(\alpha,\beta), C(A(\alpha,\beta,0,0)),1,0)$

contained the lines:

$$= (-1 + [\![\alpha]\!]_s \times [\![\beta]\!]_s - [\![\alpha]\!]_s - [\![\beta]\!]_s) \times \phi^{2z+2t}$$
$$= (-1 + [\![\alpha]\!]_s) \times (-1 + [\![\beta]\!]_s) \times \phi^{2z+2t}$$

This is false since $-1 \neq -1 \times -1$.

However we can define P' as follows:

$$P'(z:\alpha,t:\beta) \Rightarrow ((z+t+3) : A(A(C(A(\alpha,\beta,0,0)), P(\alpha,\beta),0,1), one, 1, 1))$$

where $[\![one]\!]_s = 1$. Here is the corrected proof:

Proposition 8 $[\![P'(z:\alpha,t:\beta)]\!]_f = [\![z:\alpha]\!]_f \times [\![t:\beta]\!]_f$

Proof

$$
\begin{aligned}
&[\![P'(z:\alpha,t:\beta)]\!]_f \\
&= [\![((z+t+3):A(A(C(A(\alpha,\beta,0,0)),P(\alpha,\beta),0,1),one,1,1))]\!]_f \\
&= (-1+[\![A(A(C(A(\alpha,\beta,0,0)),P(\alpha,\beta),0,1),one,1,1)]\!]_s)\times\phi^{2(z+t+3)} \\
&= (-1+([\![A(C(A(\alpha,\beta,0,0)),P(\alpha,\beta),0,1)]\!]_s+\tfrac{1}{\phi}+\tfrac{1}{\phi^2}+\tfrac{1}{\phi}+\tfrac{1}{\phi^2})/\phi^2)\times\phi^{2(z+t+3)} \\
&= (-\phi^2+\tfrac{1}{\phi}+\tfrac{1}{\phi^2}+\tfrac{1}{\phi}+\tfrac{1}{\phi^2}+([\![C(A(\alpha,\beta,0,0))]\!]_s+[\![P(\alpha,\beta)]\!]_s+\tfrac{1}{\phi^2})/\phi^2)\times\phi^{2(z+t+2)} \\
&= (-\phi^4+\phi+1+\phi+1+\tfrac{1}{\phi^2}+[\![C(A(\alpha,\beta,0,0))]\!]_s+[\![P(\alpha,\beta)]\!]_s)\times\phi^{2(z+t+1)} \\
&= (-\phi^4+\phi+1+\phi+1+\tfrac{1}{\phi^2}+\phi-[\![A(\alpha,\beta,0,0)]\!]_s+[\![P(\alpha,\beta)]\!]_s)\times\phi^{2(z+t+1)} \\
&= (-\phi^4+\phi+1+\phi+1+\tfrac{1}{\phi^2}+\phi-([\![\alpha]\!]_s+[\![\beta]\!]_s)/\phi^2+([\![\alpha]\!]_s\times[\![\beta]\!]_s)/\phi^2)\times\phi^{2(z+t+1)} \\
&= (1-\phi^6+\phi^3+\phi^2+\phi^3+\phi^2+\phi^3-[\![\alpha]\!]_s-[\![\beta]\!]_s+[\![\alpha]\!]_s\times[\![\beta]\!]_s)\times\phi^{2(z+t)} \\
&= (1-[\![\alpha]\!]_s-[\![\beta]\!]_s+[\![\alpha]\!]_s\times[\![\beta]\!]_s)\times\phi^{2z+2t} \\
&= (-1+[\![\alpha]\!]_s)\times(-1+[\![\beta]\!]_s)\phi^{2t+2z} \\
&= [\![z:\alpha]\!]_f\times[\![t:\beta]\!]_f
\end{aligned}
$$

In the following sections we show new algorithms for calculating results, all of these algorithms unless otherwise specified are only described in simplified notation. To convert these then into full notation is very simple and is described in Di Gianantonio [6].

5.2 New algorithm for addition

This new implementation of addition works differently to Di Gianantonio's algorithm by firstly adding together the two input streams, and then rewriting the stream so that the output is of the correct type.

The zip function takes in two streams α and β and adds together α_i and β_i, the algorithm is as follows:

$$\mathrm{zip}(a::\alpha,b::\beta)\Rightarrow(a+b)::\mathrm{zip}(\alpha,\beta)$$

Therefore zip takes in two steams of the form $\mathbf{2}^\omega$ and returns one stream of the form $\mathbf{3}^\omega$. We assume that α always begins with either a 00 or a 01.

Proposition 9 *There exists a computable function 3_to_2:$\mathbf{3}^\omega\to\mathbf{2}^\omega$, such that if 3_to_2($\alpha$) = β then $[\![\alpha]\!]_s=[\![\beta]\!]_s$. Also we assume that $\alpha_1=0$ and $\alpha_2=0$ or 1*

Proof

The algorithm for addition is as follows:

$$\begin{array}{ll}
\text{3_to_2}(0 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: \text{3_to_2}(0 :: 0 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 1 :: \alpha) & \Rightarrow 0 :: \text{3_to_2}(0 :: 1 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 2 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: 1 :: 0 :: \text{3_to_2}(0 :: 1 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 2 :: 0 :: 1 :: \alpha) & \Rightarrow 0 :: 1 :: \text{3_to_2}(0 :: 0 :: 2 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 2 :: 0 :: 2 :: \alpha) & \Rightarrow 0 :: 1 :: \text{3_to_2}(0 :: 0 :: 3 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 2 :: 1 :: \alpha) & \Rightarrow 1 :: 0 :: \text{3_to_2}(0 :: 0 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 2 :: 2 :: \alpha) & \Rightarrow 1 :: 0 :: \text{3_to_2}(0 :: 1 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 3 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: 1 :: 1 :: \text{3_to_2}(0 :: 1 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 3 :: 0 :: 1 :: \alpha) & \Rightarrow 1 :: 0 :: \text{3_to_2}(0 :: 0 :: 2 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 3 :: 0 :: 2 :: \alpha) & \Rightarrow 1 :: 0 :: \text{3_to_2}(0 :: 0 :: 3 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 3 :: 1 :: \alpha) & \Rightarrow 1 :: \text{3_to_2}(0 :: 1 :: 0 :: \alpha) \\
\text{3_to_2}(0 :: 0 :: 3 :: 2 :: \alpha) & \Rightarrow 1 :: 1 :: \text{3_to_2}(0 :: 0 :: \alpha) \\
\text{3_to_2}(0 :: 1 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: 1 :: \text{3_to_2}(0 :: 0 :: \alpha) \\
\text{3_to_2}(0 :: 1 :: 0 :: 1 :: \alpha) & \Rightarrow 0 :: 1 :: \text{3_to_2}(0 :: 1 :: \alpha) \\
\text{3_to_2}(0 :: 1 :: 0 :: 2 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: 1 :: 1 :: 0 :: \text{3_to_2}(0 :: 1 :: \alpha) \\
\text{3_to_2}(0 :: 1 :: 0 :: 2 :: 0 :: 1 :: \alpha) & \Rightarrow 1 :: 0 :: 0 :: \text{3_to_2}(0 :: 0 :: 2 :: \alpha) \\
\text{3_to_2}(0 :: 1 :: 0 :: 2 :: 0 :: 2 :: \alpha) & \Rightarrow 0 :: 1 :: 1 :: \text{3_to_2}(0 :: 0 :: 3 :: \alpha) \\
\text{3_to_2}(0 :: 1 :: 0 :: 2 :: 1 :: \alpha) & \Rightarrow 1 :: 0 :: \text{3_to_2}(0 :: 1 :: 0 :: \alpha) \\
\text{3_to_2}(0 :: 1 :: 0 :: 2 :: 2 :: \alpha) & \Rightarrow 1 :: 0 :: 1 :: \text{3_to_2}(0 :: 0 :: \alpha) \\
\text{3_to_2}(0 :: 1 :: 1 :: \alpha) & \Rightarrow 1 :: \text{3_to_2}(0 :: 0 :: \alpha) \\
\text{3_to_2}(0 :: 1 :: 2 :: \alpha) & \Rightarrow 1 :: \text{3_to_2}(0 :: 1 :: \alpha)
\end{array}$$

Notice that although at some points we put a the digit 3 back into the stream we only output digits of the form $\mathbf{2}^\omega$.

Proposition 10 *For any stream $\alpha \in \mathbf{3}^\omega$:*

1. *α begins with 00 or 01, is kept true in recursive calls .*

2. *3_to_2$(\alpha) \in \mathbf{2}^\omega$*

3. *$[\![\alpha]\!]_s = [\![$3_to_2$(\alpha)]\!]_s$*

Proof of Proposition 10
We will prove this by induction:

1. We can make the α given to 3_to_2 begin with 00 at the start then by noticing that all the rules will make the new call to 3_to_2 begin with either 00 or 01.

2. By looking at the digits returned by 3_to_2 we can see that they are either 0 or 1.

3. All these rules were constructed by looking at a finite part of the input and then repeatedly applying the identity $100 \equiv 011$. Therefore we are rewriting the first part of the input, without altering the value, so $[\![\alpha]\!]_s = [\![\text{3\_to\_2}(\alpha)]\!]_s$.

$\text{3\_to\_2}(0 :: 0 :: 0 :: \alpha)$	$\Rightarrow 0 :: \text{3\_to\_2}(0 :: 0 :: \alpha)$	Obvious
$\text{3\_to\_2}(0 :: 0 :: 1 :: \alpha)$	$\Rightarrow 0 :: \text{3\_to\_2}(0 :: 1 :: \alpha)$	Obvious
$\text{3\_to\_2}(0 :: 0 :: 2 :: 0 :: 0 :: \alpha)$	$\Rightarrow 0 :: 1 :: 0 :: \text{3\_to\_2}(0 :: 1 :: \alpha)$	since $00200 = 01001$
$\text{3\_to\_2}(0 :: 0 :: 2 :: 0 :: 1 :: \alpha)$	$\Rightarrow 0 :: 1 :: \text{3\_to\_2}(0 :: 0 :: 2 :: \alpha)$	since $00201 = 01002$
$\text{3\_to\_2}(0 :: 0 :: 2 :: 0 :: 2 :: \alpha)$	$\Rightarrow 0 :: 1 :: \text{3\_to\_2}(0 :: 0 :: 3 :: \alpha)$	since $00202 = 01003$
$\text{3\_to\_2}(0 :: 0 :: 2 :: 1 :: \alpha)$	$\Rightarrow 1 :: 0 :: \text{3\_to\_2}(0 :: 0 :: \alpha)$	since $0021 = 1000$
$\text{3\_to\_2}(0 :: 0 :: 2 :: 2 :: \alpha)$	$\Rightarrow 1 :: 0 :: \text{3\_to\_2}(0 :: 1 :: \alpha)$	since $0022 = 1001$
$\text{3\_to\_2}(0 :: 0 :: 3 :: 0 :: 0 :: \alpha)$	$\Rightarrow 0 :: 1 :: 1 :: \text{3\_to\_2}(0 :: 1 :: \alpha)$	since $00300 = 01101$
$\text{3\_to\_2}(0 :: 0 :: 3 :: 0 :: 1 :: \alpha)$	$\Rightarrow 1 :: 0 :: \text{3\_to\_2}(0 :: 0 :: 2 :: \alpha)$	since $00301 = 10002$
$\text{3\_to\_2}(0 :: 0 :: 3 :: 0 :: 2 :: \alpha)$	$\Rightarrow 1 :: 0 :: \text{3\_to\_2}(0 :: 0 :: 3 :: \alpha)$	since $00302 = 10003$
$\text{3\_to\_2}(0 :: 0 :: 3 :: 1 :: \alpha)$	$\Rightarrow 1 :: \text{3\_to\_2}(0 :: 1 :: 0 :: \alpha)$	since $0031 = 1010$
$\text{3\_to\_2}(0 :: 0 :: 3 :: 2 :: \alpha)$	$\Rightarrow 1 :: 1 :: \text{3\_to\_2}(0 :: 0 :: \alpha)$	since $0032 = 1100$
$\text{3\_to\_2}(0 :: 1 :: 0 :: 0 :: \alpha)$	$\Rightarrow 0 :: 1 :: \text{3\_to\_2}(0 :: 0 :: \alpha)$	Obvious
$\text{3\_to\_2}(0 :: 1 :: 0 :: 1 :: \alpha)$	$\Rightarrow 0 :: 1 :: \text{3\_to\_2}(0 :: 1 :: \alpha)$	Obvious
$\text{3\_to\_2}(0 :: 1 :: 0 :: 2 :: 0 :: 0 :: \alpha)$	$\Rightarrow 0 :: 1 :: 1 :: 0 :: \text{3\_to\_2}(0 :: 1 :: \alpha)$	since $010200 = 011001$
$\text{3\_to\_2}(0 :: 1 :: 0 :: 2 :: 0 :: 1 :: \alpha)$	$\Rightarrow 1 :: 0 :: 0 :: \text{3\_to\_2}(0 :: 0 :: 2 :: \alpha)$	since $010201 = 100002$
$\text{3\_to\_2}(0 :: 1 :: 0 :: 2 :: 0 :: 2 :: \alpha)$	$\Rightarrow 0 :: 1 :: 1 :: \text{3\_to\_2}(0 :: 0 :: 3 :: \alpha)$	since $010202 = 011003$
$\text{3\_to\_2}(0 :: 1 :: 0 :: 2 :: 1 :: \alpha)$	$\Rightarrow 1 :: 0 :: \text{3\_to\_2}(0 :: 1 :: 0 :: \alpha)$	since $01021 = 10010$
$\text{3\_to\_2}(0 :: 1 :: 0 :: 2 :: 2 :: \alpha)$	$\Rightarrow 1 :: 0 :: 1 :: \text{3\_to\_2}(0 :: 0 :: \alpha)$	since $01022 = 10100$
$\text{3\_to\_2}(0 :: 1 :: 1 :: \alpha)$	$\Rightarrow 1 :: \text{3\_to\_2}(0 :: 0 :: \alpha)$	since $011 = 100$
$\text{3\_to\_2}(0 :: 1 :: 2 :: \alpha)$	$\Rightarrow 1 :: \text{3\_to\_2}(0 :: 1 :: \alpha)$	since $012 = 101$

Therefore we can add using the following definition:

$$\text{3\_to\_2}'(\alpha, \beta) = \text{3\_to\_2}(0 :: 0 :: \text{zip}(\alpha, \beta)) = ([\![\alpha]\!]_s + [\![\beta]\!]_s)/\phi^2$$

Adding 00 to the result of the zip function guarantees that this will begin with 00.

We have proved proposition 10 and therefore proposition 9.

Unfortunately there are three reasons why this algorithm is not as good as the algorithm for addition designed by Di Gianantonio:

- Di Gianantonio's algorithm needs at most 2 digits of α and β to decide the next digit of the output. But this algorithm needs to look at between 3 and 6 digits of the input, meaning 3 to 6 digits of both α and β. Although we may be able to output more than one digit, after looking at several digits.

- There are more rules. Di Gianantonio's algorithm for addition used only 10 rules. However this algorithm uses 21 rules, although the rules that we used are easy to check, and it is clear that their work.

- It is not as easy to extend to full notation. Since the definition of addition in the full notation requires us to add a carry of $1/\phi^3$ onto the result. However this is possible and in fact we implemented this algorithm for the simplified notation and then extended it for the full notation.

5.3 Multiplication by 2

A useful function that we can have is multiplication by 2. This is not trivial to implement in GR notation because the base is not 2, but we can implement multiplication by 2, by adding the number to itself. This will mean that the input streams to the addition function are the same. However with a simple modification of the zip function we only need to look at the input stream once.

We define zip for multiplication by 2 as follows:

$$\text{double\_zip}(a :: \alpha) \Rightarrow (2 * a) :: \text{double\_zip}(\alpha)$$

Proposition 11 *There exists a computable function 02_to_2:*$\{0,2\}^\omega \rightarrow \mathbf{2}^\omega$*, such that if 02_to_2*$(\alpha) = \beta$ *then* $[\![\alpha]\!]_s = [\![\beta]\!]_s$.

The algorithm for multiplication by 2 is as follows:

$$
\begin{array}{ll}
\text{02\_to\_2}(0 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: \text{02\_to\_2}(0 :: 0 :: \alpha) \\
\text{02\_to\_2}(0 :: 0 :: 2 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: 1 :: 0 :: \text{02\_to\_2}(0 :: 1 :: \alpha) \\
\text{02\_to\_2}(0 :: 0 :: 2 :: 0 :: 2 :: \alpha) & \Rightarrow 0 :: 1 :: \text{02\_to\_2}(0 :: 0 :: 3 :: \alpha) \\
\text{02\_to\_2}(0 :: 0 :: 2 :: 2 :: \alpha) & \Rightarrow 1 :: 0 :: \text{02\_to\_2}(0 :: 1 :: \alpha) \\
\text{02\_to\_2}(0 :: 0 :: 3 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: 1 :: 1 :: \text{02\_to\_2}(0 :: 1 :: \alpha) \\
\text{02\_to\_2}(0 :: 0 :: 3 :: 0 :: 2 :: \alpha) & \Rightarrow 1 :: 0 :: \text{02\_to\_2}(0 :: 0 :: 3 :: \alpha) \\
\text{02\_to\_2}(0 :: 0 :: 3 :: 2 :: \alpha) & \Rightarrow 1 :: 1 :: \text{02\_to\_2}(0 :: 0 :: \alpha) \\
\text{02\_to\_2}(0 :: 1 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: 1 :: \text{02\_to\_2}(0 :: 0 :: \alpha) \\
\text{02\_to\_2}(0 :: 1 :: 0 :: 2 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: 1 :: 1 :: 0 :: \text{02\_to\_2}(0 :: 1 :: \alpha) \\
\text{02\_to\_2}(0 :: 1 :: 0 :: 2 :: 0 :: 2 :: \alpha) & \Rightarrow 0 :: 1 :: 1 :: \text{02\_to\_2}(0 :: 0 :: 3 :: \alpha) \\
\text{02\_to\_2}(0 :: 1 :: 0 :: 2 :: 2 :: \alpha) & \Rightarrow 1 :: 0 :: 1 :: \text{02\_to\_2}(0 :: 0 :: \alpha) \\
\text{02\_to\_2}(0 :: 1 :: 2 :: \alpha) & \Rightarrow 1 :: \text{02\_to\_2}(0 :: 1 :: \alpha)
\end{array}
$$

In fact we can use the same algorithm for addition, but we can remove some of the rules, because the input is now of the form $\{0,2\}^\omega$.

These are the subset of rules for addition that expect only the digits 0 or 2, though we may add on a prefix containing a 1 or 3 (as a carry).

Since this algorithm is basically addition we do not need to prove the algorithm is correct (we have already done this). We only need to show that one of the rules always succeeds, this can be easily checked.

The benefits of this algorithm are as follows:

- In both the addition and the multiplication by 2 algorithms it is relatively simple to see how the algorithm works, by rewriting a stream of the form $\mathbf{3}^\omega$ into one of the form $\mathbf{2}^\omega$.
- If we use the double_zip function instead of zip then this algorithm will look deeper into a stream to give the same output but will not calculate the same digits twice, however to multiply by 2 using the A algorithm implemented by Di Gianantonio [6] would mean that we separately look at digits of the same stream, which is very wasteful.
- Also when using finite lists, in the algorithm for integration and conversion from GR to decimal, we need to do addition on finite lists and it is much simpler to implement this form of addition and multiplication.

Therefore this algorithm has some benefits over Di Gianantonio's algorithm for addition.

5.4 Division by 2

In the previous section we defined a function 02_to_2:$\{0,2\}^\omega \rightarrow \mathbf{2}^\omega$ such that if 02_to_2$(\alpha) = \beta$ then $[\![\alpha]\!]_s = [\![\beta]\!]_s$, we used this function to multiply a stream by 2. We can define the inverse of this function, as 2_to_02:$\mathbf{2}^\omega \rightarrow \{0,2\}^\omega$, such that if 2_to_02$(\alpha) = \beta$ then $[\![\alpha]\!]_s = [\![\beta]\!]_s$, we can use this to divide a stream by 2.

Proposition 12 *There exists a computable function 2_to_02:*$\mathbf{2}^\omega \rightarrow \{0,2\}^\omega$*, such that if 2_to_02*$(\alpha) = \beta$ *then* $[\![\alpha]\!]_f = [\![\beta]\!]_f$.

The definition of 2_to_02 is as follows:

$$
\begin{array}{ll}
\text{2\_to\_02}(0 :: \alpha) & \Rightarrow 0 :: \text{2\_to\_02}(\alpha) \\
\text{2\_to\_02}(1 :: 0 :: 0 :: \alpha) & \Rightarrow 0 :: \text{2\_to\_02}(1 :: 1 :: \alpha) \\
\text{2\_to\_02}(1 :: 0 :: 1 :: \alpha) & \Rightarrow 0 :: \text{2\_to\_02}(1 :: 2 :: \alpha) \\
\text{2\_to\_02}(1 :: 1 :: 0 :: \alpha) & \Rightarrow 0 :: 2 :: \text{2\_to\_02}(1 :: \alpha) \\
\text{2\_to\_02}(1 :: 1 :: 1 :: \alpha) & \Rightarrow 2 :: 0 :: 0 :: \text{2\_to\_02}(\alpha) \\
\text{2\_to\_02}(1 :: 2 :: 0 :: 0 :: \alpha) & \Rightarrow 2 :: 0 :: 0 :: \text{2\_to\_02}(1 :: \alpha) \\
\text{2\_to\_02}(1 :: 2 :: 0 :: 1 :: \alpha) & \Rightarrow 2 :: 0 :: 0 :: 2 :: \text{2\_to\_02}(\alpha) \\
\text{2\_to\_02}(1 :: 2 :: 1 :: \alpha) & \Rightarrow 2 :: \text{2\_to\_02}(1 :: 0 :: \alpha)
\end{array}
$$

We can prove this algorithm using induction. We can show that all the rules in the algorithm do not change the overall value of the answer.

We can use the following facts, all provable from the identity $\phi^2 = \phi + 1$: $100 \equiv 011, 101 \equiv 012, 110 \equiv 021, 111 \equiv 200, 1200 \equiv 2001, 1201 \equiv 2002$ and $121 \equiv 210$.

Also we can see that although we may push the digit 2 onto the input stream all the digits that we output are either 0 or 2. Also we can see that after each application of this algorithm we will always output at least one digit of output, therefore this algorithm is convergent.

We can see that this function for dividing by two is much more compact than the algorithm for multiplying by two. Also this algorithm has a lower maximum lookahead, the algorithm for multiplication by two was at most six digits this needs to look at a maximum of only four digits.

To construct this algorithm we firstly consider the possible values that the stream can have, then we try and rewrite the beginning part of the stream so that it contains only 0's or 2's, but that it has the same value. If we can rewrite the first part of the stream then we can recurse on the remaining part. If we cannot rewrite the first part then we consider the possible next digits and then try and rewrite these. The simplest way to verify that the algorithm covers all possible inputs is to draw a tree-like structure of how the input can be de-constructed until it can then be rewritten.

The diagram in Figure 5.1 shows all the possible ways of expanding a stream and then rewriting it so that it only contains digits 0 and 2. When we can rewrite a portion of a stream then we can rewrite it as $\beta : \gamma\alpha$ where $\beta \in \{0, 2\}^*$ and $\gamma \in \mathbf{3}^*$, β the part to the left of the ":" is outputted and γ is put back onto α:

For any input, one of the cases will hold and we will be able to output a part of the answer.

For the algorithms for addition and multiplication by 2 we can also construct a tree to show that one case will always succeed. However since these algorithms will have more branches and will be deeper, this has not been included in this report.

5.5 New algorithm for multiplication

Multiplication of two streams α and β in simplified notation can be thought of as an infinite number of additions as follows:

$$\begin{array}{cccccccc}
= & \alpha_1 * & \cdot\beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \ldots \\
+ & \alpha_2 * & \cdot 0 & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \ldots \\
+ & \alpha_3 * & \cdot 0 & 0 & \beta_1 & \beta_2 & \beta_3 & \ldots \\
+ & \alpha_4 * & \cdot 0 & 0 & 0 & \beta_1 & \beta_2 & \ldots \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots
\end{array}$$

In this case we aligned values to the first digit in simplified notation. However we could align values in columns with respect to the same digits of the stream β as follows:

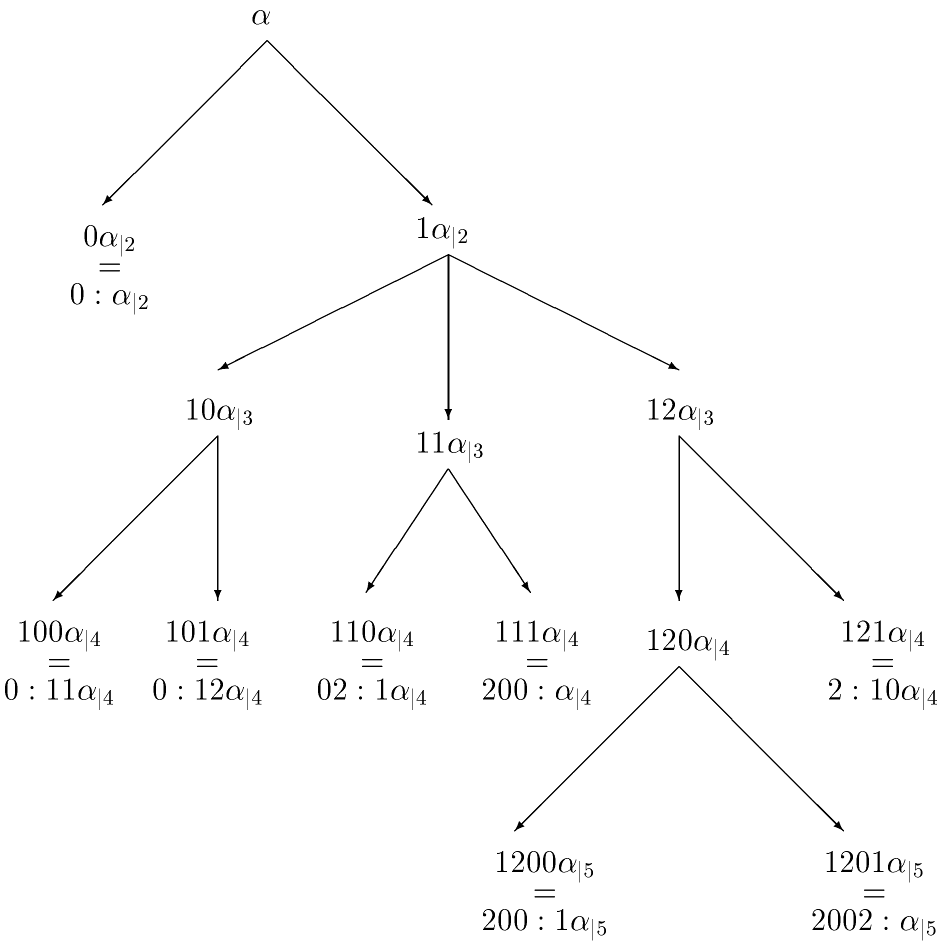

Figure 5.1: The tree of division by 2

$$
\begin{array}{cccccc|cccccc}
= & \alpha_1 * & & & & \cdot & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \dots \\
+ & \alpha_2 * & & & \cdot & 0 & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \dots \\
+ & \alpha_3 * & & \cdot & 0 & 0 & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \dots \\
+ & \alpha_4 * & \cdot & 0 & 0 & 0 & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \dots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}
$$

If we move down one element in this table then the magnitude of the β_i is divided by ϕ, moving on the diagonal from SW to NE we keep with values of the same magnitude. We can now remove the 0's to the left of the line to give the initial table that we will use to calculate the multiplication. The initial table will be of the form:

$$
\begin{array}{cc|cccccc}
= & \alpha_1 * & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \dots \\
+ & \alpha_2 * & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \dots \\
+ & \alpha_3 * & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \dots \\
+ & \alpha_4 * & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \dots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}
$$

However in general the column values will not all be equal, so we need to use a matrix notation for elements. Also when we are doing the calculation we will need 2 carries, a and b (which are initially set to 0). The beginning of all steps will be of this form:

$$
\begin{array}{lr|llllll}
 & a & b & & & & & \\
= \alpha_1 * & & \beta_{1,1} & \beta_{2,1} & \beta_{3,1} & \beta_{4,1} & \beta_{5,1} & \dots \\
+ \alpha_2 * & & \beta_{1,2} & \beta_{2,2} & \beta_{3,2} & \beta_{4,2} & \beta_{5,2} & \dots \\
+ \alpha_3 * & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \beta_{4,3} & \beta_{5,3} & \dots \\
+ \alpha_4 * & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \beta_{4,4} & \beta_{5,4} & \dots \\
\vdots & & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}
$$

Proposition 13 *There exists a computable function* $Prod : \mathbf{2}^\omega \times \mathbf{2}^\omega \to \mathbf{3}^\omega$, *such that*

$$[\![Prod(\alpha, \beta)]\!]_s = ([\![\alpha]\!]_s \times [\![\beta]\!]_s)/\phi$$

.

Proof

Firstly we construct the above table containing α and β. Then we can use the following 3 rules to calculate the next digit of output:

$$
\begin{array}{l}
Prod(0 :: \alpha, \beta_{i,1} :: \beta_{i,|2}, a, b) \Rightarrow a :: Prod(\alpha, \beta_{i,|2}, b, 0), \forall i \in \mathbb{N} \\
Prod(1 :: 0 :: \alpha, \beta_{i,1} :: \beta_{i,2} :: \beta_{i,|3}, a, b) \Rightarrow a :: Prod(1 :: \alpha, \beta_{|2,1} :: \beta_{i,|3}, b, \beta_{1,1}), \forall i \in \mathbb{N} \\
Prod(1 :: 1 :: \alpha, \beta_{i,1} :: \beta_{i,2} :: \beta_{i,|3}, a, b) \Rightarrow (a + \gamma_1) :: Prod(1 :: \alpha, \gamma_{|4} :: \beta_{i,|3}, \gamma_2, \gamma_3), \forall i \in \mathbb{N} \\
\text{where } \gamma = A(\beta_{i,1}, 0 :: \beta_{i,2}, b, b), \forall i \in \mathbb{N}
\end{array}
$$

We will prove this algorithm by using induction on the size of the stream, there is no base case since the stream is infinite. We will show that all the possible steps that we can apply will not change the output.

The first rule works as follows: if $\alpha = 0 :: \alpha_{|2}$ then

$$
\begin{array}{lr|llll}
 & a & b & & & \\
= 0* & & \beta_{1,1} & \beta_{2,1} & \beta_{3,1} & \cdots \\
+\alpha_2* & & \beta_{1,2} & \beta_{2,2} & \beta_{3,2} & \cdots \\
+\alpha_3* & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\
+\alpha_4* & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\
\vdots & & \vdots & \vdots & \vdots &
\end{array}
=
\begin{array}{lr|llll}
 & a & b & & & \\
= & \swarrow \ \swarrow & 0 & 0 & 0 & \cdots \\
+\alpha_2* & \swarrow & \beta_{1,2} & \beta_{2,2} & \beta_{3,2} & \cdots \\
+\alpha_3* & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\
+\alpha_4* & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\
\vdots & & \vdots & \vdots & \vdots &
\end{array}
$$

$$
=
\begin{array}{lrr|llll}
 & a & b & & & & \\
= \alpha_2* & & 0 & \beta_{1,2} & \beta_{2,2} & \beta_{3,2} & \cdots \\
+\alpha_3* & & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\
+\alpha_4* & & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\
\vdots & & & \vdots & \vdots & \vdots &
\end{array}
$$

In this case the first row was being multiplied by 0, all elements in the first row are 0. We can therefore move the carry and a 0 one row down and one row to the left. We have not altered the values of any of the digits in a way that will affect the result but we have removed one row, and there are now 3 carry, we can output a and relabel the remaining carry so that they are of the original form. The second rule works as follows: if $\alpha = 1 :: 0 :: \alpha_{|3}$ then

$$
\begin{array}{lr|llll}
 & a & b & & & \\
= 1* & & \beta_{1,1} & \beta_{2,1} & \beta_{3,1} & \cdots \\
+0* & & \beta_{1,2} & \beta_{2,2} & \beta_{3,2} & \cdots \\
+\alpha_3* & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\
+\alpha_4* & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\
\vdots & & \vdots & \vdots & \vdots &
\end{array}
=
\begin{array}{lr|llll}
 & a & b & & & \\
= & & \beta_{1,1} & \beta_{2,1} & \beta_{3,1} & \cdots \\
+ & & 0 & 0 & 0 & \cdots \\
+\alpha_3* & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\
+\alpha_4* & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\
\vdots & & \vdots & \vdots & \vdots &
\end{array}
$$

$$
=
\begin{array}{lr|llll}
 & a & b & & & \\
= & \swarrow \ \swarrow & \beta_{1,1} & \beta_{2,1} & \beta_{3,1} & \cdots \\
+ & & \swarrow & \swarrow & \swarrow & \cdots \\
+\alpha_3* & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\
+\alpha_4* & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\
\vdots & & \vdots & \vdots & \vdots &
\end{array}
=
\begin{array}{lrr|llll}
 & a & b & \beta_{1,1} & & & \\
= \alpha_2* & & & \beta_{2,1} & \beta_{3,1} & \beta_{4,1} & \cdots \\
+\alpha_3* & & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\
+\alpha_4* & & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\
\vdots & & & \vdots & \vdots & \vdots &
\end{array}
$$

If the second digit of α is 0 then the second row of digits will all be 0, then we can move the whole of the top row(except for $\beta_{1,1}$) and the carry down one column and to the left one row. This will not change any of the values but will result in an extra carry, $\beta_{1,1}$. We can now output a and then relabel the 2 remaining carry so that it is of the form of the input. The third rule works as follows: if $\alpha = 1 :: 1 :: \alpha_{|3}$ then

$$\begin{array}{ll|llll} & a & b & & & \\ = 1* & & \beta_{1,1} & \beta_{2,1} & \beta_{3,1} & \cdots \\ +1* & & \beta_{1,2} & \beta_{2,2} & \beta_{3,2} & \cdots \\ +\alpha_3* & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\ +\alpha_4* & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\ \vdots & & \vdots & \vdots & \vdots & \end{array} = \begin{array}{ll|llll} & a & b & & & \\ = & & \beta_{1,1} & \beta_{2,1} & \beta_{3,1} & \cdots \\ + & & \beta_{1,2} & \beta_{2,2} & \beta_{3,2} & \cdots \\ +\alpha_3* & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\ +\alpha_4* & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\ \vdots & & \vdots & \vdots & \vdots & \end{array}$$

$$= \begin{array}{lll|llll} & (a+\gamma_1) & \gamma_2 & \gamma_3 & & & \\ = \alpha_2* & & & \gamma_4 & \gamma_5 & \gamma_6 & \cdots \\ +\alpha_3* & & & \beta_{1,3} & \beta_{2,3} & \beta_{3,3} & \cdots \\ +\alpha_4* & & & \beta_{1,4} & \beta_{2,4} & \beta_{3,4} & \cdots \\ \vdots & & & \vdots & \vdots & \vdots & \end{array}$$

where

$$\begin{array}{lllllll} & a & b & \beta_{1,1} & \beta_{2,1} & \beta_{3,1} & \cdots \\ + & & & 0 & \beta_{2,2} & \beta_{3,2} & \cdots \end{array} = \begin{array}{c} a \\ \left.\begin{array}{llllll} & b & b & & & \\ + & \beta_{1,1} & \beta_{2,1} & \beta_{3,1} & \cdots & \\ + & 0 & \beta_{2,2} & \beta_{3,2} & \cdots & \end{array}\right\} = \gamma \end{array}$$

In this example then both the first and second digits of α are 1. In this case then we have to add together the first and the second rows (we have to put an extra 0 in the second row so that elements of the same order are being added. Then we add these together with b as a carry, to give γ the next digit of the output is $(a + \gamma_1)$ and the new carries are γ_2 and γ_3 respectively. Now that the first row is a combination of the first and the second row then we have to make the new $\alpha' = 1 :: \alpha_{|3}$.

At no stage in any of these steps did we alter the value of the result. We only removed the first row by manipulating the table. Therefore this algorithm must not alter value. Note that now the output is of the form $\mathbf{3}^\omega$, the first digit can be no more than 1. If we add the digit 0 to the front then the stream will begin with either 00 or 01. We can now use addition as defined in section 5.2 to convert this into a stream of the form $\mathbf{2}^\omega$.

By applying this to the result of Prod we can define a new function Prod' : $\mathbf{2}^\omega \times \mathbf{2}^\omega \to \mathbf{2}^\omega$. With $[\![\text{Prod'}(\alpha, \beta)]\!]_s = ([\![\alpha]\!]_s \times [\![\beta]\!]_s)/\phi^2$.

This algorithm for multiplication is less efficient than the method used by Di Gianantonio. The main reason that this algorithm is slow is that the number of digits that it needs to look at the next two digits of input.

Another problem with this algorithm is that the first rule is not often used. For example after we have encountered a digit 1 in α, we put the digit 1 back onto α because the top row of our table of β's are now a combination of previous rows depending on different α_i's.

Finally the worst possible case is $\alpha = 11\ldots$. This will mean that we have to do one addition for every digit of output that we get. We could try repeatedly applying the identity $100 = 011$ to guarantee that at least half of the digits of α will be 0. But the cost of looking at at least three digits of α to output one digit would remove any advantage gained.

The benefits of the algorithm is that the rules that we have constructed are very intuitive, and it uses an interesting property that is we can move the elements of a number diagonally. Also this algorithm only uses one addition, and in cases where α contains a lot of 0's then this could be very efficient.

6. Conversion

6.1 Conversion from Decimal to GR

For input and output purposes, it is important that we can convert from decimal notation to GR notation.

To convert from decimal we first have to separate a number into a list of digits e.g. from $0 \cdot d_1 d_2 d_3 \ldots d_n$ to $[0, d_1, d_2, d_3, \ldots, d_n]$, keeping a note of where the decimal point was. A simple way to do this is to multiply by 10 and then take the integer part of the result (this is d_1) then subtract d_1 and repeat to get all the digits.

$$\begin{aligned}
d = {} & 0 \cdot d_1 d_2 d_3 \ldots d_n & \\
10 \times d = {} & d_1 \cdot d_2 d_3 \ldots d_n & \text{digit} = d_1 = \text{floor}(10 \times d) \\
(10 \times d) - d_1 = {} & 0 \cdot d_2 d_3 \ldots d_n & \\
10 \times ((10 \times d) - d_1) = {} & d_2 \cdot d_3 \ldots d_n & \text{digit} = d_2 = \text{floor}(10 \times ((10 \times d) - d_1)) \\
\vdots \quad & \vdots &
\end{aligned}$$

This algorithm should terminate when n digits have been output, after the nth digit has been found the result should be 0.

To convert from a list of digits to GR we need to know the digits 1 to 10 in full GR notation. Then we can reconstruct the values by adding a value and then dividing by 10 (the opposite to de constructing the number) but we do the operations in GR notation with the digits converted to GR.

if $[\![z_{d_i} : \alpha_{d_i}]\!]_f = d_i$ then $[\![b]\!]_f = 10$ in this case, where b represents the base.

$$d = [\![z_{d_1} : \alpha_{d_1}]\!]_f \times ([\![b]\!]_f)^{-1} + [\![z_{d_2} : \alpha_{d_2}]\!]_f \times ([\![b]\!]_f)^{-2} + \ldots + [\![z_{d_n} : \alpha_{d_n}]\!]_f \times ([\![b]\!]_f)^{-n}$$

This algorithm uses n divisions and $(n-1)$ subtractions, where d has n digits. The time this takes to produce an answer therefore depends on the complexity of the number that it is given. We have not included the algorithm explicitly since it is very simple.

6.2 Conversion from GR to Decimal

This algorithm uses a similar idea as when we converted from a decimal number to a list of decimal digits. It is important to note that we cannot convert directly from infinite streams in GR notation to infinite streams of decimals, if this were possible we would be able to solve the problem of determining the first digit of the sum $\frac{4}{9} + \frac{5}{9}$. To convert to decimal we take a finite portion of the GR stream and then convert this to decimal.

Proposition 14 *In order to convert from GR to decimal, if we want n correct digits of output we need at most $5n$ digits of input.*

Proof

If we want n correct decimal digits of a number then we must be able to find a number within 10^n, the number of digits we have to look at in a GR stream will be proportional to n therefore we will take pn digits for some integer p.

We want

$$\begin{array}{ll} 10^{-n} \geq \phi^{-pn} & \\ 10^n \leq \phi^{pn} & \\ \log_{10} 10^n \leq \log_{10} \phi^{pn} & \text{taking log of both sides} \\ n \leq pn \log_{10} \phi & \\ p \geq \frac{1}{\log_{10} \phi} & \\ 5 \geq \frac{1}{\log_{10} \phi} & \text{therefore p = 5 satisfy this equation} \end{array}$$

We will consider converting from simplified notation to decimal. Then we will consider how to convert from full notation to simplified notation.

The idea of converting from GR notation to simplified notation is exactly the same method as when we calculated the digits of an decimal in the previous section. The only differences are that all the arithmetic that we are doing is in GR notation, and instead of just reading the next digit of the output we have to judge it by converting it to decimal and taking the integer part. This will mean that we will have to wait until we have calculated the last digit of our intended output before we print the first because we will have to allow for a carry.

Then we take the floor of this value. The value that we get from this may not be the actual digit, because the remaining digits in a stream may affect the value of the digit.

This algorithm could be applied to infinite streams, however this involves many multiplications and subtractions and so would take a long time. We can get exactly the same result by taking a sufficient finite number of digits of the input and then applying this algorithm, this means that algorithms for addition have to be able to work on a finite lists. This can easily be achieved.

The only problem with this algorithm is that to find n digits, we need n multiplications and $(n-1)$ subtractions. Multiplications are expensive and so this algorithm can be quite slow.

Now we consider converting from full notation to simplified notation. The aim of this algorithm is to write $z : \alpha$ as $0 : \alpha, k$ where $[\![z : \alpha]\!]_f = [\![0 : \alpha]\!]_f \times 10^k$. There are three particular cases that we have to deal with, initially k is 0:

- The exponent is already 0. In this case we can stop, with k unchanged.
- The exponent is negative. In this case we can repeatedly use the identity $[\![z : \alpha]\!]_f = [\![(z+1) : 1 :: 0 :: \alpha]\!]_f$, to make the exponent 0. Or we could multiply by 10 (and increase k by 1).

- The exponent is positive. In this case we have to divide the stream by 10, decreasing k by 1, and then carry on from the remainder of the stream.

6.3 Conversion from GR notation to Signed Binary

If we are able to convert from infinite streams in GR notation to infinite streams of SB notation than it means that it is possible to convert to, and then use all the algorithms implemented for SB notation. For example it would be possible to use the algorithms described by David Plume [14]. However for us to be able to implement these algorithms in ML we would have to re-design them to use the new stream constructor. This would be similar to the re-designing that was needed when we implemented Pietro Di Gianantonio's [6] algorithms.

In this example we convert from simplified GR notation to SB notation with no exponent. Though in practice we will want to convert from full GR notation to SB notation with an exponent. To expand the algorithm to cover this notation we will use the algorithm for converting to simplified notation. This is relatively simple and is described below.

Proposition 15 *There exists a computable function, GR_to_SB* $: \mathbf{2}^\omega \to \{-\mathbf{1}, \mathbf{0}, \mathbf{1}\}^\omega$, *such that if GR_to_SB*$(0 : \alpha) = \beta$ *then* $[\![0 : \alpha]\!]_f = [\![\beta]\!]_{SB}$. *Where* $[\![\beta]\!]_{SB} = \sum_{k=1}^{\infty} \beta_i \cdot \frac{1}{2^k}$

Proof

If there is a function GR_to_SB then it would satisfy the following properties:

$$\begin{array}{ll}
[\![\text{GR_to_SB}(0 :: 0 :: \alpha')]\!]_f & = [\![\bar{1} :: \text{GR_to_SB}(A(\alpha', \alpha', 0, 0))]\!]_{SB} \\
[\![\text{GR_to_SB}(0 :: 1 :: 1 :: \alpha')]\!]_f & = [\![\text{GR_to_SB}(1 :: 0 :: 0 :: \alpha')]\!]_{SB} \\
[\![\text{GR_to_SB}(0 :: 1 :: 0 :: \alpha')]\!]_f & = [\![\bar{1} :: \text{GR_to_SB}(1 :: A(\alpha', \alpha', 1, 0))]\!]_{SB} \\
[\![\text{GR_to_SB}(1 :: 0 :: \alpha')]\!]_f & = [\![0 :: \text{GR_to_SB}(A(\alpha', \alpha', 1, 0))]\!]_{SB} \\
[\![\text{GR_to_SB}(1 :: 1 :: \alpha')]\!]_f & = [\![1 :: \text{GR_to_SB}(A(\alpha', \alpha', 0, 0))]\!]_{SB}
\end{array}$$

The proof uses induction on the size of the input stream we will use the following facts about the SB notation:

$$\begin{array}{ll}
[\![\bar{1} :: \beta]\!]_{SB} = ([\![\beta]\!]_{SB} - 1)/2, & [\![\beta]\!]_{SB} = 2[\![\bar{1} :: \beta]\!]_{SB} + 1 \\
[\![0 :: \beta]\!]_{SB} = ([\![\beta]\!]_{SB})/2, & [\![\beta]\!]_{SB} = 2[\![0 :: \beta]\!]_{SB} \\
[\![1 :: \beta]\!]_{SB} = ([\![\beta]\!]_{SB} + 1)/2, & [\![\beta]\!]_{SB} = 2[\![1 :: \beta]\!]_{SB} - 1
\end{array}$$

Also $[\![0 : \alpha]\!]_f = (-1 + [\![\alpha]\!]_s).\phi^0 = -1 + [\![\alpha]\!]_s$.
Proof of GR_to_SB$(0 :: 0 :: \alpha') = \bar{1} ::$ GR_to_SB$(A(\alpha', \alpha', 0, 0))$
If $\alpha_1 = \alpha_2 = 0$

$$
\begin{aligned}
&[\![\mathrm{GR\_to\_SB}(0::0::\alpha')]\!]_f\\
&= [\![0::0::\alpha']\!]_s - 1\\
&= \tfrac{[\![\alpha']\!]_s}{\phi^2} - 1\\
&= (\tfrac{2[\![\alpha']\!]_s}{\phi^2} - 2)/2\\
&= [\![\bar{1} :: (\tfrac{2[\![\alpha']\!]_s}{\phi^2} - 1)]\!]_{SB}\\
&= [\![\bar{1} :: [\![\tfrac{2\alpha'}{\phi^2}]\!]_f]\!]_{SB}\\
&= [\![\bar{1} :: \mathrm{GR\_to\_SB}(A(\alpha',\alpha',0,0))]\!]_{SB}
\end{aligned}
$$

Proof of $\mathrm{GR\_to\_SB}(0::1::1::\alpha') = \mathrm{GR\_to\_SB}(1::0::0::\alpha')$
This is trivial since we are applying the identity to the input.
Proof of $\mathrm{GR\_to\_SB}(0::1::0::\alpha') = \bar{1} :: \mathrm{GR\_to\_SB}(A(1::\alpha', 1::\alpha', 0, 1))$

$$
\begin{aligned}
&[\![\mathrm{GR\_to\_SB}(0::1::0::\alpha')]\!]_f\\
&= [\![0::1::0::\alpha']\!]_s - 1\\
&= \tfrac{[\![\alpha']\!]_s}{\phi^3} + \tfrac{1}{\phi^2} - 1\\
&= (\tfrac{2[\![\alpha']\!]_s + 1/\phi}{\phi^3} + \tfrac{1}{\phi} - 1)/2 && \text{since } \tfrac{1}{2\phi^4} + \tfrac{1}{2\phi} = \tfrac{1}{\phi^2}\\
&= [\![\bar{1} :: (\tfrac{[\![A(\alpha',\alpha',1,0)]\!]_s}{\phi} + \tfrac{1}{\phi} - 1)]\!]_{SB}\\
&= [\![\bar{1} :: ([\![1 :: A(\alpha',\alpha',1,0)]\!]_s - 1)]\!]_{SB}\\
&= [\![\bar{1} :: \mathrm{GR\_to\_SB}(1 :: A(\alpha',\alpha',1,0))]\!]_{SB}
\end{aligned}
$$

Proof of $\mathrm{GR\_to\_SB}(1::0::\alpha') = 0 :: \mathrm{GR\_to\_SB}(A(\alpha',\alpha',1,0))$

$$
\begin{aligned}
&[\![\mathrm{GR\_to\_SB}(1::0::\alpha')]\!]_f\\
&= [\![1::0::\alpha']\!]_s - 1\\
&= \tfrac{[\![\alpha']\!]_s}{\phi^2} + \tfrac{1}{\phi} - 1\\
&= (\tfrac{2[\![\alpha']\!]_s + 1/\phi}{\phi^2} - 1)/2 && \text{since } \tfrac{1}{\phi} - 1 = \tfrac{1}{2\phi^3} - \tfrac{1}{2}\\
&= [\![0 :: (\tfrac{[\![A(\alpha',\alpha',1,0)]\!]_s}{\phi} - 1)]\!]_{SB}\\
&= [\![0 :: ([\![A(\alpha',\alpha',1,0)]\!]_s - 1)]\!]_{SB}\\
&= [\![0 :: \mathrm{GR\_to\_SB}(A(\alpha',\alpha',1,0))]\!]_{SB}
\end{aligned}
$$

Proof of $\mathrm{GR\_to\_SB}(1::1::\alpha') = 1 :: \mathrm{GR\_to\_SB}(A(\alpha',\alpha',0,0))$

$$
\begin{aligned}
&[\![\mathrm{GR\_to\_SB}(1::1::\alpha')]\!]_f\\
&= [\![1::1::\alpha']\!]_s - 1\\
&= \tfrac{[\![\alpha']\!]_s}{\phi^2} + \tfrac{1}{\phi} + \tfrac{1}{\phi^2} - 1\\
&= \tfrac{[\![\alpha']\!]_s}{\phi^2} && \text{since } \tfrac{1}{\phi} + \tfrac{1}{\phi^2} - 1 = 0\\
&= (\tfrac{2[\![\alpha']\!]_s}{\phi^2} - 1 + 1)/2\\
&= [\![1 :: ([\![A(\alpha',\alpha',0,0)]\!]_s - 1)]\!]_{SB}\\
&= [\![1 :: \mathrm{GR\_to\_SB}(A(\alpha',\alpha',0,0))]\!]_{SB}
\end{aligned}
$$

In fact not only does the function GR_to_SB have to satisfy the above equations, it is defined by the above equations. We can see that this is of finite character as we need to do at most two steps of the algorithm until we can output a SB digit. Also we have proved that the equations will not alter the value of the result therefore the function GR_to_SB can be defined as follows:

$$\begin{array}{lcl}
\mathrm{GR_to_SB}(0 :: 0 :: \alpha') & = & \bar{1} :: \mathrm{GR_to_SB}(A(\alpha', \alpha', 0, 0)) \\
\mathrm{GR_to_SB}(0 :: 1 :: 1 :: \alpha') & = & \mathrm{GR_to_SB}(1 :: 0 :: 0 :: \alpha') \\
\mathrm{GR_to_SB}(0 :: 1 :: 0 :: \alpha') & = & \bar{1} :: \mathrm{GR_to_SB}(1 :: A(\alpha', \alpha', 1, 0)) \\
\mathrm{GR_to_SB}(1 :: 0 :: \alpha') & = & 0 :: \mathrm{GR_to_SB}(A(\alpha', \alpha', 1, 0)) \\
\mathrm{GR_to_SB}(1 :: 1 :: \alpha') & = & 1 :: \mathrm{GR_to_SB}(A(\alpha', \alpha', 0, 0))
\end{array}$$

Conversion from GR notation to SB notation is relatively efficient, since we are only using repeated additions. The most expensive part of this algorithm is dealing with the exponent, which is nevertheless relatively simple.

The aim of this algorithm is to write $z : \alpha$ as $0 : \alpha, k$ where $[\![z : \alpha]\!]_f = [\![0 : \alpha]\!]_f \times 2^k$. There are three particular cases that we have to deal with, initially k is 0:

- The exponent is already 0. In this case we can stop, k is unchanged.
- The exponent is negative. In this case we can repeatedly use the identity $[\![z : \alpha]\!]_f = [\![(z+1) : 1 :: 0 :: \alpha]\!]_f$, to make the exponent 0. Or we could multiply by 2 (and increase k by 1).
- The exponent is positive. In this case we have to divide the stream by 2, decreasing k by 1, and then repeat.

In practice once we have converted to SB we would like to convert back to GR notation. Although this is possible we have not designed this algorithm since it would require the operators used in SB notation which we have not implemented.

We would need the value ϕ to implement this in SB notation. But it would be possible to convert from SB notation to GR notation in Plume‘s calculator. Calculation of the value of ϕ in SB notation can be achieved by just converting ϕ into SB notation as described above.

7. Intersection of nested sequences of intervals

To calculate functions like $\log, \sin$ and $\cos$ we have to calculate the sum of an infinite series. To find this we can calculate the intersection of nested sequences of intervals.

7.1 Finding the intersection of a nested sequence of intervals

Assume that we have an infinite number of closed intervals of the form $[\gamma^i, \delta^i]$, with the following properties:

$$\gamma^i \leq \gamma^{i+1} \tag{7.1}$$
$$\delta^i \geq \delta^{i+1} \tag{7.2}$$
$$\gamma^i \leq \delta^i \tag{7.3}$$
$$\forall \epsilon > 0 \exists i \text{ such that } \mid \gamma^i - \delta^i \mid < \epsilon \tag{7.4}$$

This is called a contracting sequence of intervals, since, by 7.1 and 7.2

$$[\gamma^1, \delta^1] \supseteq [\gamma^2, \delta^2] \supseteq \ldots \supseteq [\gamma^i, \delta^i] \supseteq [\gamma^{i+1}, \delta^{i+1}] \supseteq \ldots$$

By 7.4, the intersection is a singleton. The aim of this chapter is to compute the unique point in the intersection.

A diagram illustrating the relation between successive intervals, is given in Figure 7.1

Proposition 16 *There exists a computable function intersection:*$(\mathbf{2}^\omega)^\omega \times (\mathbf{2}^\omega)^\omega \to \mathbf{2}^\omega$, *such that if intersection*$(\Gamma, \Delta) = \alpha$ *then* $\alpha = \bigcap_{i=1}^{\infty}[\Gamma_i, \Delta_i]$, *note here that* Γ *and* Δ *are infinite streams of numbers, where each* Γ_i *and* Δ_i *are streams of digits.*

Proof

We want to find the intersection of a sequence of pairs of numbers, each of these numbers are an infinite stream of GR digits. We can construct two infinite streams of numbers as follows:

$$\Gamma = \gamma^1 :: \gamma^2 :: \ldots$$
$$\Delta = \delta^1 :: \delta^2 :: \ldots$$

The algorithm for calculating intersection is as follows, where lex is the function defined in section 4.2.

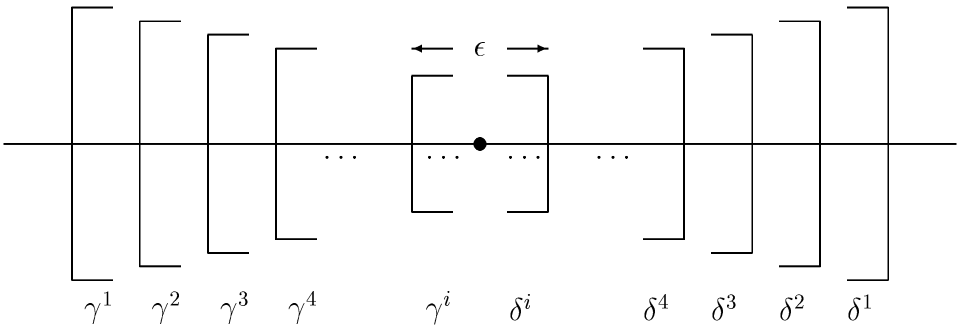

Figure 7.1: Diagram of a contracting sequence of intervals

Let $(\Gamma', \Delta') = \text{lex}(\Gamma_1, \Delta_1) :: \text{lex}(\Gamma_2, \Delta_2) :: \ldots \text{lex}(\Gamma_i, \Delta_i) :: \ldots$

$$\begin{array}{ll} \text{Intersection}(\gamma', \delta') :: (\Gamma'_{|2}, \Delta'_{|2}) & \\ = \text{Intersection}(\Gamma'_{|2}, \Delta'_{|2}) & \text{if } \gamma'_1 \neq \delta'_1 \\ = \gamma'_1 :: \text{Intersection}((\gamma'_{|2}, \delta'_{|2}) :: (\text{map\_p}(\gamma'_1, \Gamma'_{|2}), \text{map\_p}(\gamma'_1, \Delta'_{|2}))) & \text{if } \gamma'_1 = \delta'_1 \end{array}$$

Where the function $\text{map\_p}(\gamma'_1, \Gamma'_{|2})$ applies the function p to Γ'_i for all i's, where $p(\gamma'_1, \Gamma'_i) = \Gamma''_i$ such that $\gamma'_1 :: \Gamma''_i \equiv \Gamma'_i$.

If the function p exists then it satisfies the following equations:

$$p(0, 1 :: \gamma_{|2}) = \text{add\_1}(\gamma_{|2}) \tag{7.5}$$

$$p(1, 0 :: \gamma_{|2}) = \text{sub\_1}(\gamma_{|2}) \tag{7.6}$$

$$p(\gamma_1, \gamma_1 :: \gamma_{|2}) = \gamma_{|2} \tag{7.7}$$

Proof of 7.5

We can rewrite $1 :: \gamma_{|2}$, as follows:

$$\begin{array}{ll} [\![1 :: \gamma_{|2}]\!]_s & = (1 + [\![\gamma_{|2}]\!]_s)/\phi \\ & = (0 + [\![\text{add\_1}(\gamma_{|2})]\!]_s)/\phi \quad \text{since } [\![\text{add\_1}(\gamma_{|2})]\!]_s = [\![\gamma_{|2}]\!]_s + 1 \\ & = [\![0 :: \text{add\_1}(\gamma_{|2})]\!]_s \end{array}$$

This begins with the digit 0. If we remove it we get $\text{add\_1}(\gamma_{|2})$, which is the definition of rule 7.5.

Proof of 7.6

We can rewrite $0 :: \gamma_{|2}$, as follows:

$$\begin{array}{ll} [\![0 :: \gamma_{|2}]\!]_s & = [\![\gamma_{|2}]\!]_s/\phi \\ & = (1 + [\![\text{sub\_1}(\gamma_{|2})]\!]_s)/\phi \quad \text{Since } [\![\text{sub\_1}(\gamma_{|2})]\!]_s = [\![\gamma_{|2}]\!]_s - 1 \\ & = [\![1 :: \text{sub\_1}(\gamma_{|2})]\!]_s \end{array}$$

This begins with the digit 1, if we remove it we get, sub_1($\gamma_{|2}$), which is the definition of rule 7.6.

Proof of 7.7

In this case since the first digit of $\gamma_1 :: \gamma_{|2}$ is γ_1, the result of removing this digit is $\gamma_{|2}$. $\square$

Now we define an operation add_1 such that add_1(γ) $= \gamma'$ then $[\![\gamma']\!] = \min\{[\![\gamma]\!] + 1, \phi\}$:

$$\text{add\_1}(0 :: 0 :: \gamma_{|3}) \;\Rightarrow\; 1 :: 1 :: \gamma_{|3} \tag{7.8}$$
$$\text{add\_1}(0 :: 1 :: \gamma_{|3}) \;\Rightarrow\; 1 :: 1 :: \text{add\_1}(\gamma_{|3}) \tag{7.9}$$
$$\text{add\_1}(1 :: \gamma_{|2}) \;\Rightarrow\; 1^{\omega} \tag{7.10}$$

Proof of 7.8

$$[\![0 :: 0 :: \gamma_{|3}]\!]_s + 1 = [\![\gamma_{|3}]\!]_s/\phi^3 + \frac{1}{\phi} + \frac{1}{\phi^2} = [\![1 :: 1 :: \gamma_{|3}]\!]_s$$

Proof of 7.9

$$\begin{aligned} [\![0 :: 1 :: \gamma_{|3}]\!]_s + 1 \quad &= [\![\gamma_{|3}]\!]_s/\phi^3 + \tfrac{1}{\phi} + \tfrac{2}{\phi^2} \\ &= \tfrac{1}{\phi} + \tfrac{1}{\phi^2} + ([\![\gamma_{|3}]\!]_s + 1)/\phi^3 \\ &= [\![1 :: 1 :: \text{add\_1}(\gamma_{|3})]\!]_s \end{aligned}$$

Proof of 7.10

$$\begin{aligned} [\![1 :: \gamma_{|2}]\!]_s + 1 \quad &= 1 + \tfrac{1}{\phi} + [\![\gamma_{|2}]\!]_s/\phi \\ &\leq \phi \\ &= 1^{\omega} \end{aligned}$$

$\square$

Now we define an operation add_1 such that sub_1(γ) $= \gamma'$ then $[\![\gamma']\!] = \max\{[\![\gamma]\!] - 1, 0\}$.

$$\text{sub\_1}(0 :: \gamma_{|2}) \;\Rightarrow\; 0^{\omega} \tag{7.11}$$
$$\text{sub\_1}(1 :: 0 :: \gamma_{|3}) \;\Rightarrow\; 0 :: 0 :: \text{sub\_1}(\gamma_{|3}) \tag{7.12}$$
$$\text{sub\_1}(1 :: 1 :: \gamma_{|3}) \;\Rightarrow\; 0 :: 0 :: \gamma_{|3} \tag{7.13}$$

Proof of 7.11

$$\begin{aligned} [\![0 :: \gamma_{|2}]\!]_s - 1 \quad &= -1 + [\![\gamma_{|2}]\!]_s/\phi \\ &\leq -1 + \phi/\phi \qquad \text{Since } [\![\gamma_{|2}]\!]_s \leq \phi \\ &= 0 \\ &= 0^{\omega} \end{aligned}$$

Proof of 7.12

$$\begin{aligned}[\![1 :: 0 :: \gamma_{|3}]\!]_s - 1 &= [\![\gamma_{|3}]\!]_s/\phi^3 + \tfrac{1}{\phi} - \tfrac{1}{\phi} - \tfrac{1}{\phi^2} \\ &= -\tfrac{1}{\phi^2} + [\![\gamma_{|3}]\!]_s/\phi^3 \\ &= ([\![\gamma_{|3}]\!]_s - 1)/\phi^3 \\ &= [\![0 :: 0 :: \mathrm{sub\_1}(\gamma_{|3})]\!]_s\end{aligned}$$

Proof of 7.13

$$[\![1 :: 1 :: \gamma_{|3}]\!]_s - 1 = [\![\gamma_{|3}]\!]_s/\phi^3 + \frac{1}{\phi} + \frac{1}{\phi^2} - 1 = [\![0 :: 0 :: \gamma_{|3}]\!]_s$$

□

Proof of the algorithm

Case 1: $\gamma'_1 \neq \delta'_1$.

In this case then we know that $\gamma' \not\equiv \delta'$, therefore we can not say any more about α without looking at the next streams. Obviously by induction on the size of Γ and Δ then this will not change the α.

Case 2: $\gamma'_1 = \delta'_1$.

If γ and δ begin with the same digit implies that all future γ's and δ's will all begin with the same digit. Then it is obvious that the second rule of the algorithm will hold.

7.2 Applications of the intersection function

To calculate the functions $\sin, \cos$ and $\log$ we only have to generate a infinite sequence of closed intervals bounding the answer above and below and then we can use the intersection function to calculate the intersection of this nested sequence of intervals.

To generate these intervals we can use the Taylor series expansion of a function.

Calculating e^x

By using the Taylor series expansion of a function we can find that:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{7.14}$$

This holds for all real values of x. However, to find the the value of e^x using the intersection of a nested sequence of intervals, we need to construct a contracting sequence if intervals. We do this as follows. First we define

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

Proposition 17 *The following sequence of intervals forms a contracting sequence, when* $0 \leq x < 1$*:*

$$[S_i(x), S_i(x) + \frac{1}{\phi^i}], \forall i \in \mathbb{N},$$

with intersection $\{e^x\}$.

Proof

It is clear that if $x \geq 0$ then the sum 7.14 is always ≥ 0. Therefore this sum is always increasing, this means that:

$$S_i(x) \leq S_{i+1}(x), \forall i \in \mathbb{N}$$

Each $S_i(x)$ is a lower bound on the value of e^x.

$$\forall \epsilon > 0, \exists n \in \mathbb{N}, \text{ such that } \forall j, k \in \mathbb{N}, j \geq k, |s_j(x) - s_k(x)| < \epsilon$$

$$\begin{array}{rll}
s_j(x) \leq s_j(1) & = \frac{1}{1} + \ldots + \frac{1}{k!} + \ldots + \frac{1}{j!} & \\
s_k(x) \leq s_k(1) & = \frac{1}{1} + \ldots + \frac{1}{k!} & \\
s_j(x) - s_k(x) \leq s_j(1) - s_k(1) & = \frac{1}{(k+1)!} + \ldots + \frac{1}{j!} & \\
& < \frac{1}{\phi^{k+1}} + \ldots + \phi^j & \text{Since } \frac{1}{n!} < \frac{1}{\phi^n} \\
& = \frac{1}{\phi^k}.(\frac{1}{\phi} + \ldots + \frac{1}{\phi^{j-k}}) & \\
& < \frac{1}{\phi^k} &
\end{array}$$

As $j \to \infty$ then our estimation of an upper-bound of $s_k(x)$ will not change therefore

$$S_k(x) + \frac{1}{\phi^k}$$

is guaranteed to be an upper-bound of the value of e^x.

Calculating $\sin(x)$

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n.x^{2n+1}}{(2n+1)!}$$

$$\begin{array}{ll}
s_0(x) = x, & s_n(x) = \frac{(-1).x^2.s_{n-1}(x)}{(2n+3).(2n+2)} \\
S_0(x) = s_0, & S_n(x) = S_{n-1}(x) + s_n(x)
\end{array}$$

Proposition 18 $\forall k \in \mathbb{N}$ *and* $0 \leq x \leq \pi$.

$$S_{2(k+1)}(x) \leq S_{2k}(x) \quad (7.15)$$

$$S_{2k+1}(x) \leq S_{2(k+1)+1}(x) \quad (7.16)$$

$$S_{2k+1}(x) \leq \sin(x) \leq S_{2k}(x) \quad (7.17)$$

Proof

Proof of 7.15:

$$\begin{aligned}\sin(x) &= \sum_{n=0}^{\infty} \tfrac{(-1)^n.x^{2n+1}}{(2n+1)!}\\ &= \sum_{n=0}^{2k} \tfrac{(-1)^n.x^{2n+1}}{(2n+1)!} + \sum_{n=2k+1}^{\infty} \tfrac{(-1)^n.x^{2n+1}}{(2n+1)!}\end{aligned}$$

We will take the second part of this equation and show

$$\sum_{n=2k+1}^{\infty} \frac{(-1)^n.x^{2n+1}}{(2n+1)!} \leq 0, \forall k.$$

$$\begin{aligned}\sum_{n=2k+1}^{\infty} \tfrac{(-1)^n.x^{2n+1}}{(2n+1)!} &= \sum_{n=2k}^{\infty} \tfrac{(-1)^{n+1}.x^{2n+3}}{(2n+3)!}\\ &= \sum_{n=k}^{\infty} \big(\tfrac{(-1)^{2n+1}.x^{4n+3}}{(4n+3)!} + \tfrac{(-1)^{2n+2}.x^{4n+5}}{(4n+5)!}\big)\\ &= \sum_{n=k}^{\infty} \big(-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!}\big)\end{aligned}$$

But $\forall n \in \mathbb{N}$

$$\begin{aligned}-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!} &\leq 0\\ \tfrac{x^{4n+5}}{(4n+5)!} &\leq \tfrac{x^{4n+3}}{(4n+3)!}\\ \tfrac{x^{4n+5}}{x^{4n+3}} &\leq \tfrac{(4n+5)!}{(4n+3)!}\\ x^2 &\leq (4n+5)(4n+4) \quad \text{If } x = 0 \text{ then } 0 = 0\\ x^2 &\leq 16n^2 + 36n + 20\\ x^2 &\leq 20 \qquad\qquad\qquad \forall n\end{aligned}$$

This condition is satisfied when $0 \leq x \leq \pi$.

Therefore:

$$\begin{aligned}\sum_{n=k+1}^{\infty}\big(-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!}\big) - \tfrac{x^{4k+3}}{(4k+3)!} + \tfrac{x^{4k+5}}{(4k+5)!} &\leq \sum_{n=k+1}^{\infty}\big(-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!}\big)\\ \sum_{n=k}^{\infty}\big(-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!}\big) - \tfrac{x^{4(k+1)+3}}{(4(k+1)+3)!} &\leq \sum_{n=k+1}^{\infty}\big(-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!}\big)\\ -\sum_{n=k}^{\infty}\big(-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!}\big) - \tfrac{x^{4(k+1)+3}}{(4(k+1)+3)!} &\geq -\sum_{n=k+1}^{\infty}\big(-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!}\big)\\ \sin(x) - \sum_{n=k}^{\infty}\big(-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!}\big) - \tfrac{x^{4(k+1)+3}}{(4(k+1)+3)!} &\geq \sin(x) - \sum_{n=k+1}^{\infty}\big(-\tfrac{x^{4n+3}}{(4n+3)!} + \tfrac{x^{4n+5}}{(4n+5)!}\big)\\ S_{2k} &\geq S_{2(k+1)}\end{aligned}$$

We can also prove the equation 7.16, in a similar way.

Since $S_n(x) \to \sin(x)$ as $n \to \infty$. We can prove equation 7.17 using the previous two results.

Therefore

$$[S_{2k+1}, S_{2k}]$$

is a contracting sequence converging at $\sin(x)$.

In this sequence we are constricting x to be in $[0, \pi]$, if $x \notin [0, \pi]$ then we can apply the following identities to make $x \in [0, \pi]$:

$$\begin{aligned} \sin(x + 2\pi) &= \sin(x) \\ \sin(-x) &= -\sin(x) \end{aligned}$$

We can implement these identities by using the cases function (Chapter 4).

In this example we need to know the value of π. Using the facts:

$$\begin{aligned} \pi &= \arctan(1) \\ \arctan(x) &= \sum_{i=0}^{\infty} \frac{(-1)^i . x^{2i+1}}{2i+1} \end{aligned}$$

We can find that

$$\pi = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

Obviously this is a converging sequence, if $S_0 = 1, S_n = S_{n-1} + \frac{1}{2n+1}$. Then $[S_{2n+1}, S_{2n}]$ is a converging sequence.

This converges slowly and there are more efficient ways of calculating this.

If x was very large, it would be inefficient to repeatedly apply the identity $\sin(x + 2\pi) = \sin(x)$, when we could write $\sin(x + 2\pi n) = \sin(x)$ for some $n \in \mathbb{N}$. We could estimate the value n, for example by dividing x by 2π and then taking the integer part.

Calculating $\cos(x)$

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n . x^{2n}}{(2n)!}$$

$$\begin{aligned} s_0(x) &= 1, \quad & s_n(x) &= \tfrac{(-1).x^2.s_{n-1}(x)}{(2n+2).(2n+1)} \\ S_0(x) &= s_0, \quad & S_n(x) &= S_{n-1}(x) + s_n(x) \end{aligned}$$

Proposition 19 *$\forall k \in \mathbb{N}$ and $0 \leq x \leq \pi$.*

$$S_{2(k+1)}(x) \leq S_{2k}(x) \tag{7.18}$$
$$S_{2k+1}(x) \leq S_{2(k+1)+1}(x) \tag{7.19}$$
$$S_{2k+1}(x) \leq \sin(x) \leq S_{2k}(x) \tag{7.20}$$

Proof

Proof of 7.18

$$\begin{aligned} \cos(x) &= \sum_{n=0}^{\infty} \frac{(-1)^n . x^{2n}}{(2n)!} \\ &= \sum_{n=0}^{2k} \frac{(-1)^n . x^{2n}}{(2n)!} + \sum_{n=2k+1}^{\infty} \frac{(-1)^n . x^{2n}}{(2n)!} \end{aligned}$$

We can in an analogous way as in the proof for $\sin(x)$ we can show that

$$\sum_{n=2k+1}^{\infty} \frac{(-1)^n.x^{2n}}{(2n)!} \leq 0, \forall k.$$

We can also do this for the case that we take an odd number of terms and show that the remaining terms are increasing.

Combining these equations and letting $n \to \infty$ we can prove 7.20.

Therefore

$$[S_{2k+1}(x), S_{2k}(x)]$$

is a contracting sequence of $\cos(x)$ as $k \to \infty$.

Calculating $\ln(x)$

$$\ln(1+x) = \sum_{n=0}^{\infty} \frac{x^n}{n} \text{ for } 1 < x \leq 1$$

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

The closer that $|x|$ is to 1 the less efficiently this algorithm will work. But if we can force $x \in [-\frac{1}{\phi}, \frac{1}{\phi}]$, then this sequence will converge quickly to the answer. In fact,

$$S_k(x) - \frac{1}{\phi^{k-1}} < \ln(1+x) < S_k(x) + \frac{1}{\phi^{k-1}} \tag{7.21}$$

Proof of 7.21

Assume $x \in [-\frac{1}{\phi}, \frac{1}{\phi}]$. Simplifying 7.21 we get:

$$\begin{array}{rcll}
S_k(x) - \frac{1}{\phi^{k-1}} < & \ln(1+x) & < S_k(x) + \frac{1}{\phi^{k-1}} & \\
\sum_{i=0}^{\infty} \frac{x^i}{i} - \frac{1}{\phi^{k-1}} < & \sum_{i=0}^{\infty} \frac{x^i}{i} & < \sum_{i=0}^{\infty} \frac{x^i}{i} + \frac{1}{\phi^{k-1}} & \text{since } S_k = \sum_{i=k}^{\infty} \frac{x^i}{i} \\
-\frac{1}{\phi^{k-1}} < & \sum_{i=k+1}^{\infty} \frac{x^i}{i} & < \frac{1}{\phi^{k-1}} & \\
 & |\sum_{i=k+1}^{\infty} \frac{x^i}{i}| & < \frac{1}{\phi^{k-1}} & \\
 & \sum_{i=k+1}^{\infty} \frac{|x|^i}{i} & < \frac{1}{\phi^{k-1}} &
\end{array}$$

But

$$\sum_{i=k+1}^{\infty} \frac{|x|^i}{i} \leq \sum_{i=k+1}^{\infty} \frac{1}{\phi^i.i} < \sum_{i=k+1}^{\infty} \frac{1}{\phi^i} = \frac{1}{\phi^{k-1}}$$

as required. Therefore applying the intersection function on

$$[S_k(x) - \frac{1}{\phi^{k-1}}, S_k(x) + \frac{1}{\phi^{k-1}}]$$

will give us the value of $\ln(x)$.

This algorithm assumes that $-\frac{1}{\phi} < x \leq \frac{1}{\phi}$. We can use the following identities to get x int the desired form:

$$\begin{aligned} \ln(\tfrac{x}{e}) &= \ln(x) - 1 \\ \ln(x.e) &= \ln(x) + 1 \end{aligned}$$

Again, we can use the cases function in a similar way as in the algorithms for $\cos(x)$ and $\sin(x)$, to get x in the desired form.

8. Integration

In this chapter we will consider definite integration of a function defined in one variable.

The algorithm that we use was proposed by John Longly [10]. There are other ways to calculate the integral of a function. See for example [14] and [17]. However in practice these algorithms are very slow, because many intervals have to be examined in order to get a small number of digits of the output.

The aim of this algorithm is to calculate the area under a function $f(x)$ within some predetermined error $\epsilon = \phi^k$. The algorithm that we will use works by splitting the graph into Riemann strips. The diagram in Figure 8.1 demonstrates this. The shaded intervals are the part that we will calculate

For this section we will assume that the we only want to integrate from $[0, 1]$. We can then use this implementation and then 'gear-up' and 'gear-down' functions that are not in this range, as follows:

$$\int_a^b f(x)dx = \int_0^1 f(a + (b-a)x)dx$$

We will discuss this and another way of integrating from $[0, \delta]$, for some δ, in the next section.

However we have to ensure that the error of our result is at most ϵ and therefore we need to split the interval into strips that are at all points at most ϵ from the actual value.

If we partitioned the interval $[a, b]$ into n sections all with error at most ϵ, then we can find the integral over the whole function.

Proposition 20 *If we partition the interval* $[a, b]$ *into* n *disjoint intervals*

$$[a, b] = [a_0, a_1] \cup [a_1, a_2] \cup \ldots \cup [a_{n-1}, a_n] \text{ with } a_0 = a \text{ and } a_n = b$$

such that

$$\forall i = 1, \ldots, n \text{ and } \forall y \in [a_{i-1}, a_i] \text{ then } |f(a_{i-1}) - f(y)| \leq \epsilon.(b-a)$$

Then we can find

$$|\int_b^a f(x)dx - (F(b) - F(a))| \leq \epsilon$$

where $F(x)$ *is the integral of* f *at* x.

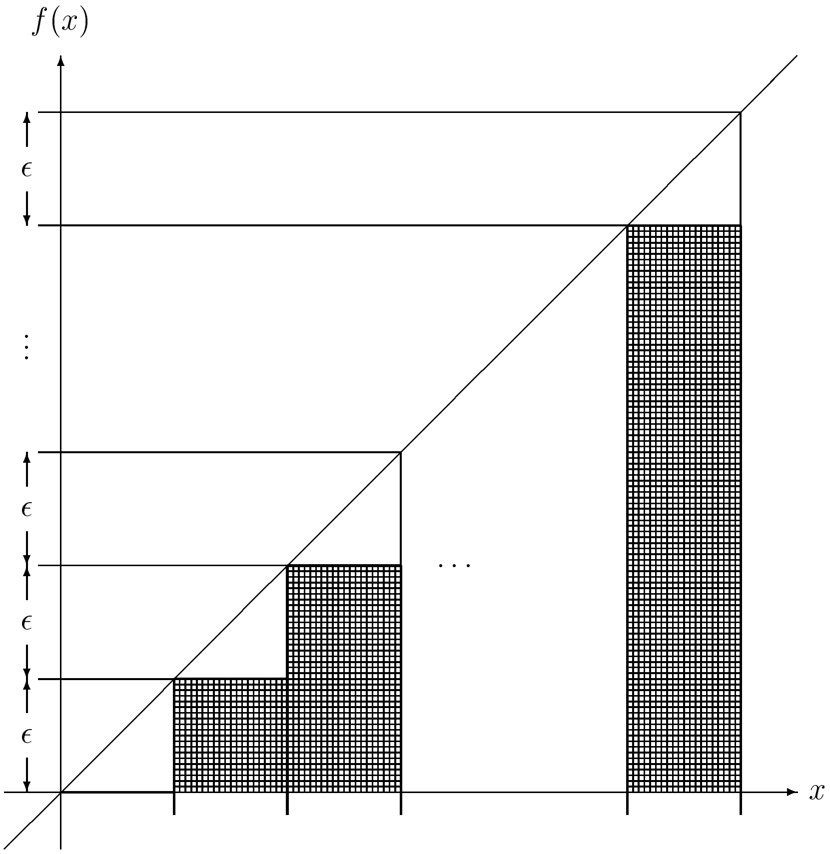

Figure 8.1: Integration on $[0, 1]$ using Riemann strips

Proof

By the above diagram we can see that we have constructed the shaded-in strips. We can find the sum of these values, and the area of $F(x)$ within $\epsilon.(b-a)$ as follows:

$$\begin{array}{ll}
\text{Estimated area of strip } [a_{i-1}, a_i] & = f(a_{i-1}).(a_i - a_{i-1}) \forall i = 1, \ldots, n \\
\text{Maximum error of strip } [a_{i-1}, a_i] & = \epsilon.(a_i - a_{i-1}) \forall i = 1, \ldots, n \\
\text{Estimated area of strip } [a, b] & = \sum_{i=1}^{n} (\text{Estimated area of strip } [a_{i-1}, a_i]) \\
\text{Maximum error of strip } [a, b] & = \sum_{i=1}^{n} (\text{Maximum error of strip } [a_{i-1}, a_i]) \\
& = \sum_{i=1}^{n} \epsilon.(a_i - a_{i-1}) \\
& = \epsilon.[(a_1 - a_0) + (a_2 - a_1) + \ldots + (a_n - a_{n-1})] \\
& = \epsilon.(a_n - a_0) \\
& = \epsilon.(b - a)
\end{array}$$

□

In this case $a = 0$ and $b = 1$. Therefore if we can split the interval $[0, 1]$ into any number of partitions each partition at most ϵ from the actual value then we have found the integral over $[0, 1]$ within ϵ.

Therefore we will now consider finding one partition that is at most ϵ from the function at any point in the partition.

8.1 Modulus of convergence

What we want:

Given α and ϵ find β and γ such that:

$$\gamma - \epsilon \leq f(x) \leq \gamma + \epsilon, \forall x \in [\alpha, \beta]$$

where $\epsilon = \frac{1}{\phi^p}$. A diagram of this can be seen in Figure 8.2

We assume that the function $f(\alpha)$ was of finite character.Recall that this means that if we want to know a finite number of output digits of the function, we will only have to consider a finite number of input digits.

Of the fact that f is of finite character we conclude that f is continuous on all $x \in \mathbf{2}^\omega$. For functions on reals a definition of uniform continuity is:

$$\forall \epsilon \exists \delta \text{ such that } |f(x - \delta) - f(x)| < \epsilon$$

We can define $x - \delta \geq [\![\alpha_1 \ldots \alpha_l (0)^\omega]\!]_f$, where $\phi^{-l+2} \leq \delta$. Therefore our definition of continuity is:

$$\forall \epsilon > 0 \exists l \text{ such that } |f(\alpha_1 \ldots \alpha_l (0)^\omega) - f(\alpha)| < \epsilon \tag{8.1}$$

In particular $\epsilon \leq \phi^{-m+2}$.

If we let m be the number of output digits then we call l the number of input digits that we have to look at.

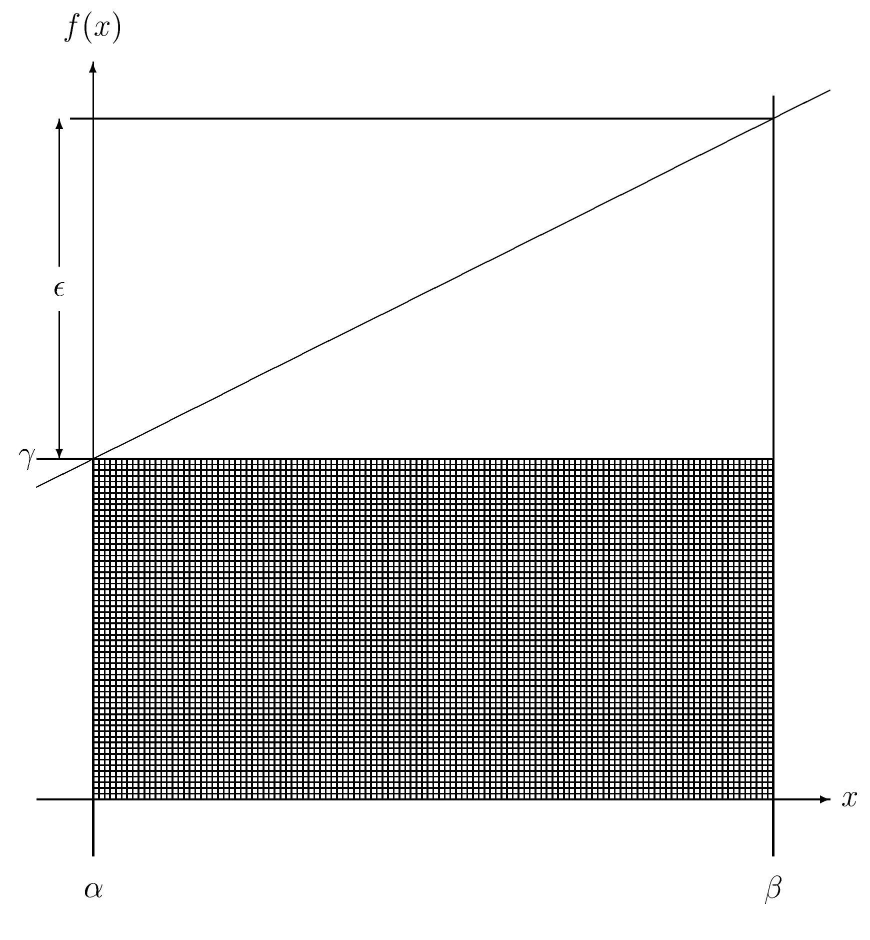

Figure 8.2: Riemann strip on one interval, $[\alpha, \beta]$

An example of 8.1 is as follows:

input (x)	$0000\,|^l\, 00\ldots$	but $0000\,|^l\, 11\ldots$
output $f(x_1 \ldots x_l(0)^\omega)$	$011\,|_m$	$011\,|_m$

Because the first l digits determined the first m digits of $f(\alpha)$, then it does not matter what the remaining digits of x (after the l'th digit) are. So we can set all of these digits to 1.

In the above example we have:

$$\gamma = f(x'_1 \ldots x'_l(0)^\omega), \beta = x' \text{ where } m \text{ is determined by } \epsilon$$

where $x = \alpha$.

Therefore if we can calculate l then we can calculate α and β.

8.2 Mod function

This function uses the fact that the language these algorithms were encoded in supports side effects in the form.

This is the reason why we used ML instead of a lazily evaluated language for example Haskell. This works by keeping track of the number of digits from the input that we have looked at.

There are several ways of storing this value for example in a memory location stored by a pointer, returned using exceptions or by using continuations. In the definition of Mod that we are using, we store the number of digits in a location addressed by a pointer.

In the idea proposed by John Longley a stream is defined as a function $\alpha : \mathbb{N} \to \mathbf{2}$, where $\alpha_i = \alpha(i)$. This definition simplify the algorithm however in this section we will use the definition of α as a sequence of digits to maintain consistency.

The definition of mod is as follows:

```
fun ModMax_f G f =
    let val log = ref ~2
        fun f'' s () =
            case s() of cons(y,s') =>
                (log := (!log)+1 ; cons(y, f'' s'))
        fun f' () =
            case f () of cons(y,s) =>
                (log := (!log)+1 ; cons(y, f'' s))
    in
        (G f', !log)
    end ;
```

The ModMax function takes two arguments G and f we are integrating the function G, f is a value written as a GR stream. The function G takes in the value f (a stream in GR notation) and outputs a value (another stream in GR notation). This is the normal result of giving the function G the argument f.

However as well as doing this the ModMax function counts the number of digits of the input that were used of f when finding G, the variable *log* keeps track of this.

Every time the function G requires the next digit it calls f with a unit value, to get the next digit of the stream f, the ModMax function will calculate this value but as a side effect the value *log* is increased by 1. This is the reason that we implemented in ML in the first place. When the result of the function is given the value *log* is also returned.

This function works on an infinite stream of digits however it could of been defined to work on a function as follows:

```
fun update NONE x = SOME x
  | update (SOME y) x = SOME (max(x,y));

fun ModMax G f =
    let val log = ref NONE
        fun f' x =
            case f x of y =>
                (log := update (!log) x ; y)
    in
        (G f', !log)
    end;
```

Both versions of the Mod function work in a similar way. In fact these definitions of the Mod function are interchangeable since we can write functions to convert from infinite streams of digits to functions from natural numbers to digits and vice versa using the following functions:

```
fun smap f (cons(a,s)) = cons(f(a),fn () => smap f (force s));

fun from n = cons(n, fn()=> from(n+1));

fun fun_to__stream f = smap f (from 0);

fun stream_to_fun (cons(a,s)) 0 = a
  | stream_to_fun (cons(_,s)) n = stream_to_fun (force s) (n-1);
```

Proposition 21 *Assume we know a_i and $f(x)$, assume that $f(x)$ is of computable(and specifically of finite character).*

Then we can find a a_{i+1} such that if $\forall x \in [a_i, a_{i+1}]$ then $|f(a_i) - f(x)| \leq \epsilon$.

Proof

Using the Mod function we can see that it is possible to find the value of an integral from $[a_i, a_{i+1}]$, where a_i and $\epsilon = \frac{1}{\phi^k}$ are specified. We can repeatedly use this starting with $[\![a_0]\!]_s = 0$, and terminate when $[\![a_0]\!]_s \geq 1$. The algorithm informally is as follows:

1. Set $[\![\alpha]\!] = 0$, α is the left hand side of the first interval. Set a value *acc* to be the sum of the area's so far, initially $[\![acc]\!] = 0$.
2. Calculate $\gamma = f(\alpha)_{|k+2}$, doing this will automatically satisfy $f(\alpha) - \epsilon \leq \gamma \leq f(\alpha) + \epsilon$, since $\epsilon = \frac{1}{\phi^k}$. We want to know l, the maximum number of digits of α that we need to look at. So we use the Mod function to do this.
3. Then $\beta = \alpha_{|l} + \frac{1}{\phi^{l+2}}$.
4. The area from α to β is
$$\begin{aligned}(\alpha - \beta).\gamma &= (\alpha - \alpha_{|l} - \tfrac{1}{\phi^{l+2}}).\gamma \\ &= \tfrac{1}{\phi^{l+2}}.\gamma\end{aligned}$$
5. Add this area to acc. If $\beta \geq 1$ then stop otherwise goto step 2.

This algorithm will obviously be correct because we can combine the Mod function as described in the previous section, and using proposition 20 we can combine all of these intervals together, to integrate over the integral $[0, 1]$. □

In this algorithm all the values are not infinite streams. Instead we use finite lists of digits, in this algorithm we only need to keep a finite number of digits of a number. However using finite lists instead of infinite streams, we cannot use the same algorithms for multiplication and for addition, both of which are used in the algorithm outlined above. In fact we only need to implement addition using finite lists because we can multiply a valve γ by $\frac{1}{\phi^k}$ in simplified notation simply by adding k zeros before γ. But notice that in full notation this can become quite complicated. The addition that we used was the addition defined in 5.2. Therefore we can guarantee termination.

The second problem that we have to deal with is to determine when $\beta \geq 1$. Since we cannot do this in practice for infinite streams. However we can use the flip function as described in chapter 4, to enforce this condition.

The modifications that we did to the algorithm originally proposed by John Longley was to change the algorithm so that it used full notation. This is not trivial since if we want the value of the stream to within ϕ^k then in the simple notation we only need to look at k digits, however with an exponent we have to take enough digits to represent the exponent and then take the next k digits.

z	α_1	$\ldots$	α_{2z}	α_{2z+1}	$\ldots$	α_{2z+n+1}	$\ldots$

$$\leftarrow \quad 2z \quad \rightarrow \leftarrow \qquad n \qquad \rightarrow$$

The second change that we made was to use streams to represent digits instead of a function. The advantage of this was that we could use all the functions that we have developed. But notice that we can use the function definition of streams and convert to and then from functions.

This is algorithm is considerably more efficient than the algorithm implemented by David Plume. Since we first implemented this function for integration, John Longly has, by modifying the definition of a stream, has been able to considerably improve the performance of this algorithm.

Another possible extension that we did not have time to implement was integration over an interval $[0, \delta]$ for some δ an infinite stream of digits. A problem that we encounter here is that we have to check if the β value has reached the end of the interval which is at δ. Normally this would not be possible to determine because as discussed previously we cannot guarantee $\beta <_{\perp} \delta$ will terminate. But in this case the value β that we are comparing with δ is a list of digits of finite length and there is no problem. We can compare β with δ, but we know after a finite number (the length of β) digits all the remaining digits are 0. Therefore if by the time we have looked at this many digits we have not terminated we can say that $[\![\beta]\!]_s \leq [\![\alpha]\!]_s$. This will terminate and we can apply the algorithm as described above to get the desired result.

9. Conclusions

9.1 Summary of work

Firstly in chapter 1 of this report we examined the problems with floating-point arithmetic. Then we considered possible ways of solving this, using infinite streams of digits in GR notation. We also discussed why we chose Golden Ratio notation.

In chapter 2 we looked at some of the theory behind the golden ratio, in particular its connection with the Fibonacci series. We proved facts that we used later in the project. We also proved an interesting property of the Fibonacci series. Using this property we proved that every number of the form $\frac{1}{2^k}$ has a periodic representation in GR notation. We then outlined the algorithm to calculate this. Though we have not used these properties in the rest of the report, they are on their own of interest.

In chapter 3 we considered the languages and representations that the calculator could be implemented in. We want to implement an interesting and more efficient method of calculating definite integration, there are several ways to solve this, the way that we chose was to implement it in a language language that allows side effects.

In chapter 4 we considered ways of rewriting streams in the same representation, whilst still representing the same number. We designed, proved and implemented the flip function which rewrites a finite portion of a stream making a specified digit 0. We then showed that this could not be extended to infinite streams.

Then we proved and implemented a lexicographical normalisation property for GR notation. The algorithm for lexicographical normalisation in SB notation was designed by Escardó in [5]. However since GR notation uses a different identity we re-designed, proved and implemented this algorithm for GR notation.

In chapter 3, we designed and implemented the basic arithmetic algorithms designed by Pietro Di Gianantonio. We also developed, proved and implemented new algorithms for multiplication, addition, multiplication by 2, and division by 2, and compared them to Di Gianantonio's algorithms.

A common theme of all the functions that we implemented is that they take in a stream in one representation, then they output a stream in a different representation without altering the value of the stream. For example one algorithm takes in a stream of the form $\mathbf{2}^\omega$ and outputs a stream of the form $\{0, 2\}^\omega$ with the same value, this can easily be used to divide the stream by 2.

We discussed conversion between representations in chapter 6. We designed, proved and implemented functions converting from: decimal to GR notation, GR notation to decimal and GR notation to SB notation. We also considered how to convert from SB notation into GR notation.

In chapter 7 we implemented and proved an algorithm for calculating the intersection of nested sequences of intervals. This function was then used to calculate the values of $\sin(x), \cos(x), e^x, \ln(x)$ and hence x^y, using the Taylor series expansion of a function and then bounding successive terms. We proved these bounds.

In chapter 8 we considered the Mod function and how this can be used to split the region [0,1] into strips, all within a given error. Then we considered how this can be used to integrate a function within a given accuracy. We looked at why this needed a language that supported side-effects. Finally, we considered possible extensions that could be made to the algorithms.

9.2 Original Material

The following algorithms are all original meaning they were designed, proved and implemented by me:

- Flip function on a finite list.
- New algorithms for addition and multiplication by 2.
- New algorithm for multiplication.
- Conversion from Decimal to GR notation.
- Conversion from GR notation to Decimal.
- Conversion from GR notation to Signed Binary.
- Lexicographical Normalisation, in GR notation.

The following algorithms have been designed, and in some cases implemented by other people. However I re-designed them to work on GR numbers and also proved that they were correct and implemented them.

- Basic Operations, these are described and proved in Di Gianantonio [6]
- Cases function, designed by Escardó [5].
- Intersection of nested sequences of intervals, the algorithm was described by Plume [14]
- Functions for calculating $e^x, \sin(x), \cos(x), \log(x)$ and x^y
- Definite Integration, designed by John Longley [10].

9.3 Discussion

The calculator that was implemented in this report is similar to the calculator that was implemented by David Plume [14]. The main differences between David Plume's and my calculator is that Plume's calculator uses Signed Binary notation with lazy evaluation and my calculator uses Golden Ratio notation with a language that supports side effects.

SB and GR notations are equivalent, since in chapter 6 we showed that we could convert from GR to SB notation and also from SB to GR notation. The main benefits of GR notation over SB is that there is only one identity (compared with two in SB notation) and every digit can only have two possibilities $\{0, 1\}$ (SB has three possible values $\{\bar{1}, 0, 1\}$). Also the ability to force a digit of a finite portion of a stream to be 0 (by using the flip function in chapter 4) is only possible in GR notation.

The main problems associated with a language that supports side effects (ML) as opposed to a lazy evaluated language (HASKELL), is that, as discussed in chapter 3, we need to use a new stream constructor. This means that we cannot use pattern matching and this results in long code that is difficult to debug and read.

There is another serious problem associated with this definition of streams, that results returned by functions like 'add' in lazy evaluation are a stream of actual digits. However in ML the result is given as one digit and a function that can then be evaluated to give the remaining digits. The problem with this is that if we knew α and calculated 'add(α, α)' in lazy evaluation we would calculate α first, and then substitute these values for each α, however 'add(α, α)' in ML will result in each digit of α being evaluated twice, once for the first α and then again for the second α in 'add(α, α)'. The reason for this is that when we are given α it has not been evaluated. This has the effect of slowing down computations, making it take much longer to calculate the basic operations in my calculator than in David Plume's calculator.

The advantage, and the reason for implementing this calculator in ML was that as well as calculating the answer as usual, we can make it possible to do other operations as a side effect. Therefore as the answer is calculated we can, for example count the number of digits of the input that were examined to get a result. The function that does this is called the 'Mod' function. It is impossible to implement the 'Mod' function in a lazily evaluated programming language. This is discussed in chapter 3. The advantage of having the 'Mod' function is that we can implement definite integration, using only finite lists which is significantly faster than the method used by David Plume.

The algorithm for definite integration, was proposed by John Longley [10], although this method had been implemented in SB notation, without an exponent. It had only been tested on simple functions, for example the constant and identity functions. We modified this algorithm to work on GR notation with an exponent and were therefore able to test it on polynomial functions.

We can also compare the algorithms that were designed, proved and implemented in section 5 with the algorithms designed by Pietro Di Gianantonio [6]. Discussed in this chapter were the algorithms for multiplication by 2 (using the algorithm for addition) and division by 2 were more efficient than by using Di Gianantonio's algorithms for example 'add(α, α)' would be less efficient than my multiplication algorithms, similarly 'div(α, two)' such that $[\![two]\!]_s = 2$ is less efficient then my algorithm for division by 2.

Finally the cases function and lexicographical normalisation cannot be compared with other implementations or algorithms since we have not found any other instances of these implementations of algorithms, as this is original material.

Bibliography

[1] M. Beeler, R. Gosper, R. Schroppel. *Hakmem.* Massachusetts Institute of Technology AI Laboratory. MIT AI memo 239 (HAKMEM). Item 101, p. 39-44. 1972.

[2] H. Boehm. R. Cartwright. *Exact Real Arithmetic, Formulating Real Numbers as Functions.* In Turner. D., editor, Research Topics in Functional Programming, pages 43-64. Addison-Wesley, 1990.

[3] H. Boehm. *Constructive Real Interpretation of Numerical Programs.* SIGPLAN Notices, 22(7):214-221, 1987.

[4] A. Edalat, P.J. Potts. *A New Representation for Exact Real Numbers.* Electronic Notes in Theoretical Computer Science, volume 6. http://www.elsevier.nl/locate/entcs/volume6.html. 1997.

[5] Martín Escardó. *Effective and sequential definition by cases on the reals via infinite signed-digit numerals.* In Electronic Notes in Theoretical Computer Science, volume 13. Available from http://www.elsevier.nl/locate/entcs/volume13.html. 1998.

[6] Pietro Di Gianantonio. *A Golden Ratio Notation for the Real Numbers.* CWI Technical Report. Available from http://www.dimi.uniud.it/~pietro/Papers/paper_arg.html.

[7] Pietro Di Gianantonio. *A Functional Approach to Computability on Real Numbers.* Ph.D. Thesis: Universita di Pisa-Genova-Udine. Available from http://www.dimi.uniud.it/~pietro/Papers/paper_arg.html.

[8] Graham, Knuth, Patashnik. *Concrete Mathematics: A foundation for Computer Science.* Addison Wesley. ISBN 0-201-14236-8.

[9] J. Hall. *Numerical Analysis and Optimisation.* Course Notes. Department of Mathematics, Edinburgh University. 1998.

[10] J.R. Longley. *When is a functional program not a functional program?: a walk through introduction to the sequentially realizable functions.* Standard ML source file, available from http://www.dcs.ed.ac.uk/home/jrl/, 1998.

[11] J.R. Longley. *When is a functional program not a functional program?* Paper available from http://www.dcs.ed.ac.uk/home/jrl/, March 1999.

[12] Valérie Ménissier-Morian. *Arbitrary precision real arithmetic: design and algorithms.* J. Symbolic Computation, 11, 1-000. 1996.

[13] A. Nielsen, P. Kornerup. *MSB-First Digit Serial Arithmetic.* Journal of Universal Computer Science, Vol 1, no 7, 527-547. 1995.

[14] David Plume. *A Calculator for Exact Real Number Computation.* BSc Project, School of Computer Science. Edinburgh University. Available from http://www.dcs.ed.ac.uk/~dbp/. 1998.

[15] P.J. Potts. *Computable Real Arithmetic Using Linear Fractional Transformations.* Electronic Notes in Theoretical Computer Science 6, 1997. Available from http://theory.doc.ic.ac.uk/~pjp/.

[16] P.J. Potts, A. Edalat. *Exact Real Computer Arithmetic.* Department of Computing Technical Report DOC 97/9, Imperial College, 1997. Available from http://theory.doc.ic.ac.uk/~pjp/.

[17] A. Simpson. *Lazy Functional Algorithms for Exact Real Functionals.* Mathematical Foundations of Computer Science 1998, Springer LNCS 1450, pp. 456-464, 1998.

[18] P. Sünderhauf. *Incremental Addition in Exact Real Arithmetic.* In R. Harmer et al., editors, Advances in Theory and Formal Methods of Computing: Proceedings of the fourth Imperial College Workshop, IC Press. Available from http://theory.doc.ic.ac.uk/~ps15/abstracts/incr.html. December 1997.

[19] E. Wiedmer. *Computing with Infinite Objects.* Theoretical Computer Science 10:133-155. 1980.