

Extraction of Programs for Exact Real Number Computation Using Agda



Swansea University
Prifysgol Abertawe

Chi Ming Chuang

Department of Computer Science

Swansea University

Submitted to Swansea University in fulfilment of
the requirements for the Degree of

Doctor of Philosophy

2011

Abstract

This thesis contains the to our knowledge first research project to extract in the theorem prover Agda programs from proofs involving postulated axioms. Our method doesn't require to write a Meta program for extracting programs from proofs. It shows as well the correctness of the machinery.

This method has been applied to the extraction of programs about real number computation. The method has been used for showing that the signed digit approximable real numbers are closed under addition, multiplication, and contain the rational numbers. Therefore we obtain in Agda a provably correct program which executes the corresponding operations on signed digit streams.

The first part of the thesis introduces axioms about real numbers using postulated data types and functions in Agda without giving any computational rules. Then we investigate some properties of real numbers constructed by Cauchy sequences: we introduce the set of real numbers which are limits of Cauchy sequences of rational numbers (Cauchy Reals) and show that they are closed under addition and multiplication. We also prove that Cauchy Reals are Cauchy complete.

Furthermore, we introduce the real numbers in the interval $[-1,1]$, which have a binary signed digit representation, i.e. $r = 0.d_0d_1d_2\dots$, where $d_i \in \{-1, 0, 1\}$. This set of real numbers is given as a codata type (SDR). We determine for rational numbers in the interval $[-1, 1]$ their SDR and show that SDRs are closed under the average function and the multiplication function. Besides, a finding digit function is

defined which determines the first n digits of a stream of signed digits.

In the second part of the thesis, a theorem is given which shows the correctness of our method. It shows that under certain conditions our method always normalises and doesn't make use of the axioms. The conditions mainly guarantee that a postulated function or axiom has as result type only a postulated type, so the reduction of elements of algebraic data types to head normal form will not refer to these postulates.

Because of our theorem the finding digit function applied to a real number r s.t. SDR r holds normalises to $[d_0, d_1, \dots, d_{n-1}]$ for the first n digit $d_0d_1 \dots d_{n-1}$ of r . Therefore, we can compute the SDR of rational numbers and from SDRs of real numbers the SDR of their average and product.

Declaration

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed (candidate)

Date

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Where correction services have been used, the extent and nature of the correction is clearly marked in a footnote(s).

Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed (candidate)

Date

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed (candidate)

Date

Contents

Contents	iv
1 Introduction	1
1.1 Program Extraction	2
1.2 Main Result of Thesis: Internalisation of Program Extraction into Agda together with Correctness Proof	5
1.3 The Structure of the Thesis	6
1.4 Related Work	7
1.5 More Details on Interval Arithmetic and Program Extraction . . .	11
1.6 Talk, Publication	15
2 Introduction to Agda	16
2.1 The Language of Agda	16
2.1.1 Basic Principles	17
2.1.2 Postulated Types and Terms	17
2.1.3 Dependent Functions	18
2.1.4 Data type and Pattern Matching	19
2.1.5 Inductive and Coinductive Data type	21
2.1.6 Equality	23
2.1.7 Let, Where-expressions and Mutual Definitions	24
2.1.8 BUILTIN and Primitive	25
2.1.9 Modules	27
2.1.10 Compiled Version of Agda	29

3	Real Numbers and Their Axiomatisation	31
3.1	Natural Numbers, Integers, and Rational Numbers	32
3.2	Real Numbers	33
3.2.1	Dedekind Cuts	34
3.2.2	Cauchy Sequences	36
4	Coalgebras and Coinduction	41
4.1	Initial Algebras and Final Coalgebras	43
4.2	Coalgebras and Codata in Agda	50
4.3	Least and Greatest Fixed Points in Set Theory	55
5	Cauchy Reals in Agda	58
5.1	Axioms	58
5.2	\mathbb{Q}' Closure Under Addition	66
5.3	\mathbb{Q}' Closure Under Multiplication	67
5.4	\mathbb{Q}' is Cauchy-complete	71
6	Signed Digit Representation of Real Numbers in Classical Mathematics	74
6.1	Signed Digit Representation	75
6.2	The signed Digit Representations of Real Numbers -1, 0 and 1	83
6.3	The signed Digit Representations of Rational Numbers	85
6.4	Average	85
6.4.1	Function <code>avaux</code>	86
6.4.2	Function <code>av</code>	87
6.5	Multiplication	88
6.5.1	Function <code>mpaux</code>	89
6.5.2	Function <code>addR</code>	92
6.5.3	Function <code>mp</code>	95
7	Signed Digit Representation of Real Numbers in Agda	96
7.1	SDR	96
7.2	SDR as Codata Type $\sim\mathbb{R}$	102
7.2.1	Proof of $\sim\mathbb{R}(-^r1)$, $\sim\mathbb{R}^r0$, $\sim\mathbb{R}^r1$	103

7.2.2	The Function transfer $\sim\mathbb{R}$	107
7.2.3	The Function $\sim\mathbb{R}$ embed \mathbb{Q}	108
7.2.4	Examples of Function $\sim\mathbb{R}$ embed \mathbb{Q}	112
7.3	$\sim\mathbb{R}$ Is Closed Under the Average Function av	113
7.3.1	$\sim\mathbb{R}$ Is Closed Under the Function avaux	114
7.3.2	$\sim\mathbb{R}$ Closure Under av	120
7.3.3	Examples of the Average Function	121
7.4	$\sim\mathbb{R}$ Is Closed Under the Multiplication Function mp	121
7.4.1	$\sim\mathbb{R}$ Is Closed Under the mpaux Function	124
7.4.2	$\sim\mathbb{R}$ Is Closed Under the add \mathbb{R} Function	130
7.4.3	$\sim\mathbb{R}$ Is Closed Under the Function scale ⁿ	132
7.4.4	$\sim\mathbb{R}$ Closure Under mp	134
7.4.5	Examples of the Multiplication Function	135
7.5	Defining $\sim\mathbb{R}$ Using the New Representation of $\sim\mathbb{R}$	135
7.6	Computing the Extracted Program	136
7.6.1	Testing	141
8	Extraction of Programs from Proofs in Agda	143
8.1	Program Extraction	143
8.2	Main Theorem: The Correctness of Program Extraction	144
8.2.1	Mathematical Preliminaries on Multisets	145
8.2.2	Agda Normalises Elements of Algebraic Data Types to Normal Form	151
8.2.2.1	Global Assumption - Restrictions on Agda Code	152
8.2.2.2	Pattern Matching Can Be Restricted to Simple Patterns	163
8.2.2.3	Proof That Agda Normalises Elements of Algebraic Data Types to Head Normal Form	179
8.2.2.4	Extensions of this Theorem	183
9	Conclusion	186
9.1	Achievements	186
9.1.1	Our Program Extraction Method	188

9.1.2 Correctness Theorem	189
9.2 Future Work	189
9.3 Possible Simplification	191
Appendix A	193
Bibliography	196

Acknowledgements

I would like to thank my supervisors, Dr Anton Setzer and Dr Ulrich Berger for their guidance, supervision and academic advice of this research project, and their support with patience and knowledge throughout my thesis.

I am also grateful to my family, Chua's family, Molly and Charles for their fully support and caring along the years in Swansea. Thanks to the Lord.

Chapter 1

Introduction

In real number computation the most common approach is to use floating point numbers which have limited approximations (which are in fact certain rational numbers) and calculate the accumulation of the rounding errors. For instance, if one calculates the product c of two approximated floating point numbers a, b then one obtains $|c - a * b| < \epsilon$ for some error margin ϵ . Then one calculates all the errors accumulated in the calculation and concludes what the overall error is. Such calculations are quite complicated and have usually to be done by hand. Moreover, errors might have occurred during the calculation (people make mistakes) which might cause serious consequences [Krä98]. Therefore, different approaches have been proposed such as interval arithmetic (see Section 1.5) and exact real arithmetic [YD94, PEE97].

Exact Real Number Computation. We will concentrate in this thesis on the exact real number computation. In exact real arithmetic real numbers are treated as infinite objects instead of finite objects. Since real numbers are infinite objects an arbitrarily good approximation of each real number can be given. Computational functions over real numbers request sufficiently good approximations of inputs to be able to yield the desired accuracy of outputs, i.e. these functions need to be continuous. For instance, if computing the real number $c = a * b$, then c is an algorithm (or a program) which is able to compute c arbitrary precisely by requesting sufficiently precise approximations of a and b . So if we can get a and b arbitrary precisely, then we are able to do the same for c . In contrast with the

floating point approach exact real arithmetic gives no errors and there is no need for calculating the error. On the other hand this approach is computationally much more expensive.

1.1 Program Extraction

The problem is that algorithms for exact real computation are more complicated. How do we know that the results we have computed are correct? How do we obtain such algorithms? The problem of program correctness applies to other programs as well, e.g. critical systems. There are several approaches for developing correct programs:

- Proving correctness by hand which is the most common way of writing programs. (The problem is that errors occur and this is not suitable for verifying larger programs or systems)
- Proving with some machine support, e.g. like most specification languages, without theorem proving. Although less errors occur and the user is forced to obey a certain syntax, theorem proving still needs to be done by hand.
- Interactive theorem proving, e.g. Coq [Coq09], Isabelle [Isa09]. Everything, which can be proved by hand, should be possible to be proved in such systems and proofs are fully checked by the system, therefore the correctness is guaranteed (provided the theorem prover is correct). However, it usually takes substantially longer than proving theorems by hand and for most mathematical proofs this is still infeasible.
- Automated theorem Proving, which usually is faster than interactive theorem proving. There are limits on what can be done. Most infinitary problems cannot be proved automatically.

When using interactive theorem proving there are two ways of obtaining a correct program:

- Write a program and then verify its correctness. The problem is that when the program changes, the correctness often has to be redone again.

- Write a proof and extract a program from it.

We will follow the last approach.

Why is program extraction interesting? Apart from obtaining verified programs, one could rediscover existing algorithms in an unexpected way and discover new algorithms.

One example is done by Berger [Ber05a], he considers the function which reverses a list

```
reverse : List → List
reverse [] = []
reverse (a : l) = (reverse l) ++ [a]
```

This is inefficient since in each step it needs to append $[a]$ to $(\text{reverse } l)$ which is linear in length of l . We need to do this length of l times ($l = \text{list to be reversed}$). So in total $O(n^2)$ steps are required if $n = \text{length}(l)$. Berger was able to obtain the (previously known) optimised version which is linear in l :

```
reverse l = reverseaux l []
```

The idea is that $\text{reverseaux } l \ l' = (\text{reverse } l) ++ l'$, reverseaux is defined as follows

```
reverseaux : List → List → List
reverseaux [] l = l
reverseaux (a : l) l' = reverseaux l (a : l')
```

Berger obtained this efficient algorithm from a classical proof using a refinement of the Friedman-Draglin A-translation by program extraction [BSB02]. Berger found as well a novel efficient higher types version for computing the Fibonacci numbers by using program extraction (details can be found in Section 1.5).

Another example is extraction of algorithms for normalisation. Normalisation means you have a rewriting system and show $\forall s \exists t. s \longrightarrow^* t \wedge \text{NF}(t)$ where $\text{NF}(t)$ means t is in normal form. From this proof one could obtain an algorithm which computes for s its normal form. An inefficient algorithm for normalising λ -terms is to normalise a term step by step which creates lots of unnecessary work e.g. in

case of the simple λ -calculus

$$\begin{aligned}
 & (\lambda x.f\ x\ x)\ ((\lambda xy.x)\ x) \\
 \longrightarrow & f\ ((\lambda xy.x)\ x)\ ((\lambda xy.x)\ x) \\
 \longrightarrow & f\ (\lambda y.x)\ ((\lambda xy.x)\ x) \\
 \longrightarrow & f\ (\lambda y.x)\ (\lambda y.x)
 \end{aligned}$$

So $(\lambda xy.x)\ x$ is evaluated twice. By contrast, in an efficient implementation it is evaluated only once. Berger [Ber93] was able to obtain such efficient algorithm for Gödel's system T. This was extended to system F by Matthias Eberl [Ebe02].

Our goal is to extract good algorithms for exact real number computation. So in our work we might obtain better or unexpected algorithms and a proof of correctness. We do everything inside Agda [Nor09]. One should note that at present we are not able to see the resulting algorithm, so we don't know what the extracted algorithm is. We will mention in future work an approach to make algorithms visible inside Agda.

The usual approach in program extraction is to take the formal proof and write a Meta program which from the proof obtains the program. For instance, assume a formal proof of $\forall x\exists y.\varphi(x, y)$ where $\varphi(x, y)$ is a property which expresses some specification for any natural numbers x, y . For a natural number n we obtain a proof of $\exists y\varphi(n, y)$ e.g. for $n = 17$ we obtain a proof of $\exists y\varphi(17, y)$. This proof need not be in normal form, for instance it might end in

$$\frac{B \rightarrow \exists y\varphi(17, y) \quad B}{\exists y\varphi(17, y)}$$

From a non-normal proof we cannot determine the instance y needed. Using normalisation or cut eliminations we obtain a normal form or cut-free proof (depending on the system) and the proof in the example will for instance be as follows:

$$\frac{\varphi(17, 6)}{\exists y\varphi(17, y)}$$

Then we obtain the desired result as 6. In real situations it is much more complicated. This requires a Meta program which takes a proof $p : \forall x\exists y.\varphi(x, y)$ and

extracts a program from p which computes y from x . In our work we take an approach to program extraction which doesn't require a Meta program: we construct a proof $b : B$ inside Agda and evaluate it by normalisation. This is possible since there is no difference between proofs and programs in Agda and Agda has builtin the axiom of choice. So from a proof $p : \forall x : A. \exists y : B. \varphi(x, y)$ we can define inside type theory the function $f : A \rightarrow B$ s.t. $\forall x : A. \varphi(x, f x)$ holds. In fact, $f = \lambda x. \pi_0(p x)$ and $\lambda x. \pi_1(p x)$ is a proof of $\forall x : A. \varphi(x, f x)$. So f can be evaluated inside Agda. For instance, if $A = B = \mathbb{N}$ we can apply f to a natural number n and evaluate $f n$ in Agda. One goal of this thesis is to show that $f n$ evaluates to a natural number under certain conditions.

1.2 Main Result of Thesis: Internalisation of Program Extraction into Agda together with Correctness Proof

We carry out the to our knowledge first approach of extracting programs from proofs involving postulated axioms in Agda and show the correctness of the machinery. Here, postulated axioms are constants with no reduction rules.

Agda [Nor09] is an interactive theorem prover based on dependent type theory which has the advantage of using a dependent type system: the proof of correctness can be written in the same language as the program. So a proof of a property of a function is based on the actual implementation internally, proofs and programs in Agda are really the same. Moreover, Agda has a novel approach, it doesn't force the user to prove given goals using logical rules. It is like functional programming and allows programmers to use their programming skills for proving theorems.

Agda is also a paradigm for programming with dependent types which can be used for developing correct programs and writing more generic programs. It provides a better type system than standard programming languages based on simple types. For instance, to define a matrix multiplication, we can take the function which takes three natural number n , m and k , an $n \times m$ -matrix and an

$m \times k$ -matrix, and has as result an $n \times k$ -matrix. This is in standard programming language based on simple types usually defined as a function taking matrices of arbitrary dimensions which returns a matrix of appropriate dimensions and checks the correctness of the dimensions at run-time, so errors are not detected at compile time. Let $(\text{Mat } n \ m)$ be the type of $n \times m$ -matrices in Agda. In Agda the matrix multiplication can get the correct function type $(n \ m \ k : \mathbb{N}) \rightarrow \text{Mat } n \ m \rightarrow \text{Mat } m \ k \rightarrow \text{Mat } n \ k$. So, that the dimensions are correct is checked at compiled time. .

Unlike other theorem provers Agda doesn't require a Meta program for program extraction, which generates programs outside the system. In Coq [Coq09], program extraction requires a Meta program that extracts the computational parts from the proof and generates an ML [ML190] or Haskell [Has] program.

Furthermore, Agda allows us to encode infinite objects inhabiting the coinductive data type of streams. Therefore, we can represent real numbers as potentially infinite sequences.

In this thesis we explore Agda in the presence of postulated axioms: we introduce the real numbers (by using postulated data types and functions in Agda without giving any computational rules) in the interval $[-1,1]$, which have a binary signed digit representation [BH08]. Since we axiomatise real numbers using postulated axioms, program extraction becomes more complicated. The extracted function might make use of the axioms which might prevent normalisation to its head normal form. We provide a correctness theorem which guarantees that under certain conditions the extracted function always normalises to head normal form.

1.3 The Structure of the Thesis

In **Chapter 2** we give an introduction to the features of the language Agda, which have been used in this thesis. In **Chapter 3** we present an overview over the literature of Cauchy sequences, and show how to construct the real numbers by Cauchy sequences. In **Chapter 4** we present the background on coinductive data types (codata) based on the use of F-coalgebras.

In **Chapter 5** we will introduce the axioms of the real numbers by using postulated data types and functions. Then we will investigate some properties of real numbers constructed by Cauchy sequences: we will prove that the Cauchy reals (which are the real numbers which are limits of Cauchy sequences of rational numbers) are closed under addition and multiplication and show that the Cauchy reals are Cauchy complete.

In **Chapter 6 and 7** we will introduce the real numbers in the interval $[-1,1]$, which have a binary signed digit stream representation (SDR) in classical mathematics (**Chapter 6**) and in Agda (**Chapter 7**), i.e. are of the form $0.d_0d_1d_2\cdots$ where $d_i \in \{-1, 0, 1\}$, as a codata type. We will show that the real numbers with SDR are Cauchy reals and the signed digit approximable real numbers are closed under average, multiplication, and contain the rational numbers in the interval $[-1,1]$. (The signed digit representation of irrational numbers e.g. $\sqrt{2}$ are left for future work). Furthermore, we will define a function `toList` which for $s : \text{SDR}$, $n : \mathbb{N}$ returns the list of the first n digits of s .

In **Chapter 8** we give the details of our program extraction method. This chapter will also provide a correctness theorem of our method showing that under certain conditions (`toList s n`) will always normalise to a list of signed digits and therefore won't make use of the axioms. The conditions will mainly guarantee that a postulated function or theorem has as result type only a postulated type, so the computation of elements of algebraic data types to head normal form will not refer to these postulates. Therefore, (`toList s n`) returns a list of n digits.

Use of mathematics. In Chapters 3 (excluding 3.1), 4.3, 6 and 8 we work in classical mathematics. Proofs in Chapters 3.1, 4.1, 4.2, 5 and 7 are carried out in type theory using in most cases (not in 4.1 and 4.2) the theorem prover Agda.

1.4 Related Work

To our knowledge the use of programs which make explicit use of postulates as axioms is new. After some extensive research we couldn't detect any articles which deal directly with program extraction into Agda. However extensive research has

been done on the development of programs in Agda.

Nuo [Nuo10] defines integers, rational numbers, real number, complex numbers and proves the basic properties of them as the tools for theorem proving. He also investigates the construction of real numbers based on Bishops real number system and implementation of real numbers in Coq and LEGO. Mu, Ko and Jansson [MKJ09] have developed a library called AoPA (Algebra of Programming in Agda) which allows to encode relational derivations in Agda by stepwise refinement. They also have shown how to translate Haskell programs using the monads into Agda, and carried out some case studies in verification of properties of Haskell programs in Agda [MKJ08]. Alonzo and Agate [Ben07, OTK09] are two compilers. They allow to translate Agda into fast executable programs. (Agda terms can be evaluated but Agda is slow - using this compilation we get relatively efficient programs from Agda). Alonzo is now integrated into the Agda framework.

Theoretical models of higher type real number computation. Marcial-Romero and Escardo [MRE07] present the description and semantics of RealPCF which is an extension of PCF by a real number type and operations on real numbers. The semantics is based on domain theory. PCF was introduced by Milner and Plotkin. It is a mathematical model of a higher-order functional programming language. The most important paper on PCF is [Plo77]. In this paper Plotkin proves completeness and adequacy results for extensions of PCF by parallel operators (parallel if) w.r.t. a domain semantics.

General approaches to real number computation. Edalat and Heckmann [EH02] give a detailed presentation of the LFT (Linear Fractional Transformation) approach to exact real number computation. An LFT is a function of the form $f(x) = (ax + b)/(cx + d)$ where a, b, c, d are rational numbers. It can be represented by the matrix

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

Composition of LFTs can be done by matrix multiplication. An LFT representation of a real number is an infinite composition of LFTs. Special cases of LFTs

are:

$$f(x) = (x + d)/2, \quad \text{where } d \text{ is in } \{-1, 0, 1\}.$$

This yields the signed digit representation

$$f(x) = 1/(x + n), \quad \text{where } n \in \mathbb{N}$$

This yields continued fraction representations.

Konecny [Kon04] proves that linear affine function (functions of the form

$$f(x_1, \dots, x_n) = a_1 * x + \dots + a_n * x_n + b$$

where a_i, b are rational numbers) are the only real functions that can be implemented as finite automata (i.e. with finite memory).

Blanck [Bla05] introduces a fast C implementation of high iterations of real functions of the form $f : [0, 1] \rightarrow [0, 1]$, $f(x) = ax(1 - x)$ where a is a rational number. Blank presents a detailed efficiency analysis and provided informal correctness proofs. Plume implements real arithmetic in Haskell w.r.t. the signed digit representation. Plume's implementation is available on website [Plu]. It also contains the iterated functions studied by Blanck [Bla05].

Implementation of real number computation by coinduction using Coq and other theorem provers. Hancock and Setzer introduce the principle of guarded induction and weakly final coalgebras in dependent type theory. This work is based on their study of interactive programs in dependent type theory, which is a special case of a coalgebra [HS04, HS05, HS99, HS00a, HS00b]. The book [BC04] by Bertot and Castran, which is a thorough textbook on the theorem prover Coq, contains in Chapter 1.3 a long and detailed introduction into using coalgebras in Coq. Bertot [Ber05b] contains a general introduction to coinduction in Coq with applications to exact real number computation. Geuvers, Niqui, Spitters and Wiedijk develop constructive analysis and exact real number computation in Coq. Mainly foundational aspects are addressed in [HGW07]. Ciaffaglione and Gianantonio [CG06] show the verification of exact real number algorithms in Coq w.r.t. to the signed digit representation. Bertot [Ber07] extends Ciaffaglione/Gianantonio's work using infinite streams of digits implemented as a co-inductive type in Coq. Berger and Hou [BH08] also extend Ciaffaglione/-

Gianantonio's work, but use a different notion of coinductive proof: whereas in Coq coinductive proofs are infinite guarded expressions, they work with finite proofs that use axioms expressing that a coinductive predicate is a greatest fixed point of some monotone operator. Addition and multiplication are treated in their article [BH08]. Niqui [Niq08] discusses coinductive formal reasoning in exact real number computation.

Program extraction in general. Kreisel [Kre59] introduces modified realisability which is the basis for many approaches to program extraction. Troelstra [Tro73] contains a very detailed account of realisability and a summary of the research that has been done. Tatsuta [Tat98] seems to be the first who did program extraction for coinductive definitions. He uses a version of q-realisation. It contains only very simple applications (e.g. pointwise inversion of an infinite bit stream). The article by Benl, Berger, Schwichtenberg, Seisenberger and Zuber [BBS⁺98] contains an explanation how to carry out program extraction in Minlog. In [Ber93] Berger shows that a normalisation program can be extracted from Tait's strong normalisation proof for the simply typed lambda calculus and that the extracted program is normalisation-by-evaluation. The paper also introduces non-computational quantifiers that yield simpler extracted programs. Normalisation-by-evaluation has also been extracted formally in Coq (Letouzey), Isabelle (Berghofer) and Minlog (Schwichtenberg/ Berger), see the joint paper [BBL06]. Berger, Schwichtenberg and Buchholz develop in [BSB02] the theory and application of program extraction from classical, i.e. non-constructive proofs using a refined version of the Friedman/Dragalin A-translation. Matthias Eberl [Ebe02] extends the work by Berger [Ber93] in his PhD thesis and obtained an efficient algorithm for normalisation for system F. Seisenberger has in her PhD thesis [Sei03] (see as well articles [Sei02, Sei01, BS05]) used A-translation in order to extract a program from the Nash-Williams proof of Higman's lemma. She [Sei08] has generalised the refined A-translation method for extracting programs from classical proofs to include choice principles such as classical dependent choice.

Program extraction for exact real number computation. In Berger's recent papers [Ber09a, Ber09b] he gives detailed proofs of the correctness of real-

isability, i.e. Soundness Theorem, Adequacy Theorem for untyped realisers and an introduction to realisability for exact real number computation with simple, but detailed examples. Together with Seisenberger he extends this work to realisability with typed realisers which correspond to lazy functional programming language such as Haskell. They describe in detail how to extract a program for the addition function for real numbers [BS10b, BS10a]. Schwichtenberg [Sch08] extracts a program from a constructive version of the Inverse Function Theorem in Minlog.

Alternative approaches to type theory: explicit mathematics and Frege structures. An alternative approach to using type theory for formulating constructive mathematics is explicit mathematics, introduced by Feferman in [Fef75]. Kahle [Kah99] has studied Frege structures in partial applicative theories. This approach allows to define a certain notion of a pointer, a concept closely related to the concept of promises in Scheme. Promises are used to introduce streams in functional programming languages with strictness.

1.5 More Details on Interval Arithmetic and Program Extraction

Interval arithmetic is an approach to real number computation without rounding errors. In interval arithmetic [Kea96] real numbers are approximated by rational or floating point intervals e.g. π is approximated by $[3.1415, 3.1416]$.

If r is approximated by $[a, b]$
and s is approximated by $[c, d]$
then $r + s$ is approximated by $[a + c, b + d]$

where in case of using floating point numbers one makes sure that $a + c$ is rounded down and $b + d$ is rounded up. For instance, if we assume precision of decimal numbers up to 10 digits (that is not what is happening on the computer which

uses binary numbers) then

$$0.\underbrace{1000000005}_{10 \text{ digits}} + 1.0 = 1.\underbrace{1000000005}_{10 \text{ digits}}$$

would be rounded up to

$$1.\underbrace{1000000001}_{9 \text{ digits}}$$

and

$$0.\underbrace{1000000014}_{10 \text{ digits}} + 1.0 = 1.\underbrace{1000000014}_{10 \text{ digits}}$$

would be rounded down to

$$1.\underbrace{1000000001}_{9 \text{ digits}}$$

but

$$\begin{aligned} & [0.\underbrace{1000000005}_{10 \text{ digits}}, 0.\underbrace{1000000014}_{10 \text{ digits}}] + [1.0, 1.0] \\ &= [1.\underbrace{1000000000}_{9 \text{ digits}}, 1.\underbrace{1000000002}_{9 \text{ digits}}] \end{aligned}$$

where 1.1000000005 has been rounded down to 1000000000 and 1.1000000014 has been rounded up to 1.000000002. Then at the end one knows how precise the result is so calculation of rounding error is not necessary. However, we don't get any information on how precisely we should have calculated the input and intermediate values in order to obtain the desired precision. By contrast, exact real arithmetic will automatically determine the precision of all inputs and intermediate calculations in order to obtain the desired precision of the output. Since we need to compute pairs of floating point numbers rather than single floating point numbers, interval arithmetic takes twice as long (unless it is supported by the machine) as simple floating point arithmetic which is slightly more expensive but in many cases acceptable.

An example of using Berger's Method of Program Extraction. Berger [BSB02, Ber05a] found a novel efficient version of the Fibonacci function using

higher types for encoding Pairs by program extraction. The naive implementation of the Fibonacci function is defined as follows:

$$\begin{aligned} \text{fib} &: \mathbb{N} \rightarrow \mathbb{N} \\ \text{fib } 0 &= 1 \\ \text{fib } 1 &= 1 \\ \text{fib } (n + 2) &= \text{fib } n + \text{fib } (n + 1) \end{aligned}$$

With this naive definition

$$\begin{aligned} \text{fib } 5 &= \text{fib } 4 + \text{fib } 3 \\ &= (\text{fib } 3 + \text{fib } 2) + (\text{fib } 2 + \text{fib } 1) \\ &= ((\text{fib } 2 + \text{fib } 1) + (\text{fib } 1 + \text{fib } 0)) + ((\text{fib } 1 + \text{fib } 0) + 1) \\ &= \dots \end{aligned}$$

which requires $O(2^n)$ many steps. E.g. above fib 3 is calculated twice, fib 2 is calculated three times and so on. In general the calculation of fib n requires fib (k) many calculations of fib $(n - k)$.

The efficient algorithm is to calculate $g(n) := \langle \text{fib}(n), \text{fib}(n + 1) \rangle$. Then

$$\begin{aligned} g(0) &= \langle 1, 1 \rangle \\ g(n + 1) &= \langle \text{fib}(n + 1), \text{fib}(n + 2) \rangle \\ &= \langle \text{fib}(n + 1), \text{fib}(n) + \text{fib}(n + 1) \rangle \\ &= \langle b, a + b \rangle \\ &\text{where } \langle a, b \rangle = g(n) \end{aligned}$$

Note $a = \text{fib}(n)$ and $b = \text{fib}(n + 1)$. $g(n)$ is calculated in linear time and we obtain $\text{fib}(n) = \pi_0(g \ n)$. This is not the optimal algorithm. There exists an algorithm (which uses matrix multiplication and repeated squaring), which computes (fib n) in $O(\log(n))$ many steps.

Berger discovered a higher type version of the linear algorithm based on pairing. Pairs can be encoded by higher types. The pair $\langle n, m \rangle$ is encoded as

$$\pi(n, m) := \lambda f : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} . f \ n \ m : (\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$$

We obtain projections by

$$\begin{aligned}\pi_0(F) &= F(\lambda xy.x) \\ \pi_1(F) &= F(\lambda xy.y)\end{aligned}$$

and see immediately $\pi_0(\pi(n.m)) = n$ and $\pi_1(\pi(n.m)) = m$. Now we apply this to the function g above so we define g s.t.

$$g\ n = \pi(\text{fib } b, \text{fib } (n + 1)) : (\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$$

$$\begin{aligned}g &: \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \\ g\ 0 &= \pi(\text{fib}0, \text{fib}1) = \pi\ 1\ 1 = \lambda f.f\ 1\ 1 \\ g\ (n + 1) &= \pi(b, a + b) \\ &\text{where } a = \pi_0(g\ n) \\ &\quad b = \pi_1(g\ n) \\ &\text{so} \\ g\ (n + 1) &= \pi(\pi_1(g\ n), \pi_0(g\ n) + \pi_1(g\ n)) \\ &= \lambda f.f(\pi_1(g\ n))(\pi_0(g\ n) + \pi_1(g\ n)) \\ &= \lambda f.f(g\ n(\lambda xy.y))((g\ n(\lambda xy.x)) + (g\ n(\lambda xy.y)))\end{aligned}$$

Now we can define

$$\text{fib } n = \pi_0(g\ n) = g\ n(\lambda xy.x)$$

Berger extracted essentially the above program from a classical proof using a refinement of the Friedman-Draglin A-translation. He took a proof of $\forall n \exists m. \text{Fib}(n, m)$

where $\text{Fib}(n, m)$ expresses $m = \text{fib}(n)$, which is axiomatised as follows:

$\text{Fib}(0, 1)$ because $(\text{fib}(0) = 1)$

$\text{Fib}(1, 1)$ because $(\text{fib}(1) = 1)$

$\text{Fib}(n, m) \rightarrow \text{Fib}(n + 1, k) \rightarrow \text{Fib}(n + 2, m + k)$

(because if $m = \text{fib}(n)$

$k = \text{fib}(n + 1)$

then $m + k = \text{fib}(n + 2)$)

Then he extracted an algorithm. Magically, the above mentioned algorithm came out.

1.6 Talk, Publication

This research (especially Chapter 8, Program Extraction and Correctness) was presented in an invited talk at a workshop on program extraction associated with CSL 2010 and MFCS 2010 [Set10c] and we have been invited to submit an article for the post-proceedings of that workshop.

Chapter 2

Introduction to Agda

In this thesis our work is carried out in the theorem prover Agda. Agda is based on Martin-Löf’s intuitionistic type theory [ML84]. It is a descendant of Cayenne [Aug98] and Alf [MN94]. The current version is Agda2 [Nor09] and the previous version is Agda1 [Coq05]. Agda2 was implemented by Ulf Norell at Chalmers University in Gothenburg and its syntax is distinct from Agda1. It is not only a proof assistant but also a functional programming language with dependent types [Nor08] and it is possible to compile Agda programs into Haskell [Has]. When it is used as a proof assistant terms are proofs. Agda2 looks more like a programming language rather than a proof assistant, and it is similar to the programming language Epigram [MM08]. Furthermore, its Emacs-based interface and its module system allow users to construct extensive proofs/programs interactively. Unlike other tactic-based theorem provers such as Coq it always shows full proof terms. In this thesis we work in version 2.2.4 of Agda 2 and in the following, when we mention Agda, we mean that version of Agda 2.

2.1 The Language of Agda

In this section we briefly introduce basic features of Agda used in this thesis and introduce the syntax of Agda by using as examples Agda code from this thesis. The full details of the language Agda can be found on the Agda Wiki [Nor09] and in Ulf Norell’s PhD thesis [Nor07, Nor08] .

2.1.1 Basic Principles

In Agda types are for history reason denoted by Agda's reserved key word `Set`. So $A : \text{Set}$ means A is a type, $a : A$ means a is an element of type A . Agda follows the propositions-as-types paradigm. Propositions and data types are both represented by elements of `Set`. For instance the decidable $<$ -relation on \mathbb{N} is `_Bool<n_` : $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Bool}$, which via an operation `Atom` : `Bool` \rightarrow `Set` (for the definition of `_Bool<n_` and `Atom` see p.20) converts into $x <^n y := \text{Atom} (x \text{ Bool}<^n y) : \text{Set}$. An element of $p : x <^n y$ is a proof that $x <^n y$ holds. In case of \mathbb{R} $<$ is undecidable, we only define directly `_<_` : $\mathbb{R} \rightarrow \mathbb{R} \rightarrow \text{Set}$ where underscore "`_`" is used to form mixfix symbols. The symbol `_` denotes where the arguments are placed. Therefore, the function `_<_` can be used as an infix operator writing $n < m$ instead of `_<_ n m`.

Sets in Agda can be data types and formulas. There is no formal distinction in Agda between the two. A formula A considered as true if there is an element $p : A$ of it. Then p is considered a proof of A .

2.1.2 Postulated Types and Terms

Agda allows users to postulate a type or a function by using Agda's reserved key word **`postulate`**. (In the following we use bold font to indicate Agda's reserved key words). This means that a constant of this type is introduced without any reduction rules. For instance, in this thesis the set of real numbers \mathbb{R} with elements 0 and 1 are introduced as postulates as follows:

```
postulate  ℝ : Set
postulate  0 : ℝ
postulate  1 : ℝ
```

where ⁰ and ¹ are constants denoting 0 and 1. The relations and operations over \mathbb{R} are as well introduced as postulated types and functions

postulate

<code>_==_</code>	<code>: ℝ → ℝ → Set</code>	{- Equal -}
<code>_<_</code>	<code>: ℝ → ℝ → Set</code>	{- Less -}
<code>_≤_</code>	<code>: ℝ → ℝ → Set</code>	{- Less or equal -}
<code>_</code>	<code>: ℝ → ℝ</code>	{- Negation -}
<code>_#_</code>	<code>: ℝ → ℝ → ℝ</code>	{- Apartness -}
<code>_+_</code>	<code>: ℝ → ℝ → ℝ</code>	{- Addition -}
<code>_*_</code>	<code>: ℝ → ℝ → ℝ</code>	{- Multiplication -}
<code> _</code>	<code>: ℝ → ℝ</code>	{- Absolute value -}

One can declare the precedence and fixity of an operator by using Agda's reserved key words **infix**, **infixl**, and **infixr**, e.g.

```
infix 40 _==_ _≤_
infixl 60 _+_
infixl 70 _*_
```

The number denotes the priority. Here `_*_` binds more than `_+_` which in turn binds more than `_==_`, `_≤_`. `_+_` is left associative by Agda's reserved key word **infixl** (so $a + b + c$ is parsed as $(a + b) + c$). **infixr** denotes right associative operations, e.g. the cons operation on lists `_: A → List A → List A` is denoted as **infixr** `_:` and $a :: b :: l$ is parsed as $a :: (b :: l)$.

2.1.3 Dependent Functions

In Agda one can define functions which have result types depending on their arguments. For instance, one can define

$$f : (r : \mathbb{R}) \rightarrow (Q : \mathbb{R} \rightarrow \text{Set}) \rightarrow \text{Set}$$

$$f \ r \ Q = Q \ r$$

where f depends on r and Q . One also can write $(r : \mathbb{R})(Q : \mathbb{R} \rightarrow \text{Set}) \rightarrow \text{Set}$ instead of $(r : \mathbb{R}) \rightarrow (Q : \mathbb{R} \rightarrow \text{Set}) \rightarrow \text{Set}$. Similarly for $(r : \mathbb{R}) \rightarrow (s : \mathbb{R}) \rightarrow \text{Set}$ one can write $(r \ s : \mathbb{R}) \rightarrow \text{Set}$ instead. Agda allows users to use implicit

arguments. An example would be $_++_ : \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \text{List } A \rightarrow \text{List } A$ which appends two lists. Now the argument A is omitted when using $_++_$, one writes $l ++ l'$ or $_++_ l l'$ instead of $_++_ A l l'$. The hidden argument can be made explicit by using $\{_ \}$. E.g. one writes $_++_ \{A\} l l'$ if one wants to write down the argument A explicitly.

Since the elements of dependent function types are lambda terms, one can define f above instead as follows (" \backslash " denotes " λ ")

$$\begin{aligned} f &: (r : \mathbb{R}) \rightarrow (\mathbb{Q} : \mathbb{R} \rightarrow \text{Set}) \rightarrow \text{Set} \\ f \ r &= \backslash \mathbb{Q} \rightarrow \mathbb{Q} \ r \end{aligned}$$

or

$$\begin{aligned} f &: (r : \mathbb{R}) \rightarrow (\mathbb{Q} : \mathbb{R} \rightarrow \text{Set}) \rightarrow \text{Set} \\ f &= \backslash r \rightarrow \backslash \mathbb{Q} \rightarrow \mathbb{Q} \ r \end{aligned}$$

xf or

$$\begin{aligned} f &: (r : \mathbb{R}) \rightarrow (\mathbb{Q} : \mathbb{R} \rightarrow \text{Set}) \rightarrow \text{Set} \\ f &= \backslash (r : \mathbb{R}) \rightarrow \backslash (\mathbb{Q} : \mathbb{R} \rightarrow \text{Set}) \rightarrow \mathbb{Q} \ r \end{aligned}$$

An implicit argument can be abstracted explicitly by using the notation $\backslash \{ _ \} \rightarrow$, e.g. $_++_ = \backslash \{A\} \rightarrow \backslash l l' \rightarrow \dots$. One can easily define a function which is identified with f by $f'' = f$ but f and f'' will have the same implicit and explicit arguments.

2.1.4 Data type and Pattern Matching

In general one can define an algebraic data type consisting of elements introduced by constructors using Agda's reserved key word **data**. For instance, in this thesis we define a data type `Digit` in Agda as follows

```
data Digit : Set where
  (d)0 : Digit
  (d)1 : Digit
  (d)-1 : Digit
```

This is the type of signed digits consisting of $0, 1, -1$. Another example is the data type \top which is defined as follows

```
data  $\top$  : Set where  
  triv :  $\top$ 
```

where \top is the true formula, which has a trivial proof $\text{triv} : \top$.

One can as well define an empty data type. For instance, \perp is defined as follows

```
data  $\perp$  : Set where
```

where \perp is the false formula, which has no element. So \perp is the data type with no constructor.

Furthermore, functions over algebraic data types can be defined by pattern matching. For instance, a function embedding `Digit` into \mathbb{R} is defined as follows

```
embedD : Digit  $\rightarrow$   $\mathbb{R}$   
embedD (d)0 = r0  
embedD (d)1 = r1  
embedD (d)-1 = - r1
```

What will happen if we make pattern matching on \perp ? We will get an absurd pattern `()`, i.e.

```
efq : {A : Set}  $\rightarrow$   $\perp$  - > A  
efq ()
```

The absurd pattern `()` indicates that there is no constructor of \perp so all cases are covered. In Agda functions defined by pattern matching must cover all cases. If there are missing cases the coverage checker of Agda will raise a error.

There is another situation where an absurd pattern `()` occurs, namely when an argument of a function in one specific case has no valid constructor pattern. We demonstrate this case but first we need to introduce the less than function

on natural numbers \mathbb{N} (see next section for the definition of \mathbb{N} in Agda)

$\text{Atom} : \text{Bool} \rightarrow \text{Set}$

$\text{Atom true} = \top$

$\text{Atom false} = \perp$

- - Atom converts a Boolean value into the formula expressing

- - that this Boolean value is true

$_ \text{Bool} <^n _ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Bool}$

$n \text{ Bool} <^n 0 = \text{false}$

$0 \text{ Bool} <^n \text{suc } m = \text{true}$

$\text{suc } n \text{ Bool} <^n \text{suc } m = n \text{ Bool} <^n m$

$_ <^n _ : (x \ y : \mathbb{N}) \rightarrow \text{Set}$

$x <^n y = \text{Atom } (x \text{ Bool} <^n y)$

Now we can define a function as follows

$$g : (p : 4 <^n 3) \rightarrow \mathbb{N}$$
$$g ()$$

where p is a proof that 4 is less than 3 which doesn't exist, since $4 <^n 3 = \perp$. Therefore, one can use the absurd pattern $()$ to indicate that there is no proof of $4 <^n 3$. Agda allows one to introduce pattern matching automatically. If one does this on the argument p , Agda will show the absurd pattern.

Another special pattern apart from $()$ is the dot pattern. Details of the dot pattern can be found in next section.

2.1.5 Inductive and Coinductive Data type

Agda allows users to define inductive, coinductive and record data types. (We will not make use of record types in this thesis and therefore won't introduce them). For instance, one can define the inductive data type of the natural numbers by

using **data** as follows

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

where `zero` and `suc` are constructors of natural numbers. So \mathbb{N} is the least set containing `zero` which is closed under `suc`. Another example is the type of Booleans, i.e.

```
data Bool : Set where
  true  : Bool
  false : Bool
```

One can define recursive functions on algebraic data types. An example is the function `_Bool<n_` above. The termination checker of Agda will guarantee that all functions are terminating and therefore Agda is normalising.

Furthermore, one can introduce data type families in Agda. For instance, the family of predicates over natural numbers n of expressing that n is even can be introduced as follows

```
data IsEven : ℕ → Set where
  evenZ : IsEven zero
  evenSS : (n : ℕ) → IsEven n → IsEven (suc (suc n))
```

This is an indexed algebraic data type, where n is the index of `IsEven n`. When one makes pattern matching on an argument which is an element of `(IsEven n)` one gets the information about the index which indicates that n is either `zero` or `suc (suc n)`. For instance, the following function, which shows that the sum of two even numbers is also even, can be defined as follows:

```
evenadd : (n m : ℕ) → IsEven n → IsEven m → IsEven (n +n m)
evenadd n .zero en evenZ = en
evenadd n .(suc (suc m)) en (evenSS m em) =
  evenSS (n +n m) (evenadd n m en em)
```

Here, `.zero` indicates that if the argument of `(IsEven m)` is `evenZ` then m must be zero. Similarly, `.(suc (suc m))` indicates that if the argument of `(IsEven m)`

is (`evenSS m em`) then m must be `(suc (suc m))`. So the dot patterns are not actual patterns of `IsEven` but they determine the correct type of values for the arguments of `IsEven`. The dot patterns are so called inaccessible patterns. In order to distinguish the inaccessible patterns and the actual patterns of `IsEven`, the inaccessible patterns are prefixed with a dot.

One can as well define coinductive or infinite data type in Agda by using Agda's reserved key word `codata`. For instance, the data type `Stream` can be defined as follows:

```
codata Stream (A : Set) : Set where
  _ :: _ : A → Stream A → Stream A
```

The intuition is that elements of `Stream A` are formed by infinitely many applications of `_ :: _`. So they have the form $a_1 :: a_2 :: a_3 :: \dots$. For example, one can now introduce

```
inc : ℕ → Stream ℕ
inc n = n :: inc (suc n)
```

The termination checker of Agda will check that guarded recursion is used, i.e. the right hand side contains at least one constructor before making a recursive call, and no function except for constructors is applied to a recursive call. More details about the theory of codata types will be given in Section 4.2.

2.1.6 Equality

In type theory the two most important forms of equality are definitional and propositional equality. Definitional equality of $a : A$ and $b : A$ means that the judgement $a = b : A$ is provable. We won't elaborate this, see [ML84, NPS90] for details. In Agda this judgement is implicitly used during type checking (e.g. if type checking $\lambda x.x : B a \rightarrow B b$ where $B : A \rightarrow \text{Set}$, Agda checks that $a = b : A$). $a = b : A$ in Agda means that a and b have the same normal form up to α -conversion.

Propositional equality is given by the data type

```
data _==_ {A : Set} (a : A) : A → Set where
  refl : a == a
```

That $a : A$ and $b : A$ are equal w.r.t. propositional equality means that $a == b$ is provable in type theory, i.e. there exist $p : a == b$.

2.1.7 Let, Where-expressions and Mutual Definitions

In Agda `let` and `where`-expressions are used for declaring local definitions. The difference between them is that pattern matching or recursive functions are not allowed in `let`-expressions. For example,

```
f : ℕ
f =let n : ℕ
    n = 4
in n +n 5
```

```
reverse : A : Set → List A → List A
reverse A l = reverseaux l []
  where
    reverseaux : List A → List A → List A
    reverseaux [] ys = ys
    reverseaux (x :: xs) ys = reverseaux xs (x :: ys)
```

Both expressions can always be omitted by defining corresponding global definitions.

One can use Agda's reserved key word **mutual** to define two functions or data types that depend to each other. For example, consider

```
mutual
  data Even : Set where
    Z : Even
    S : Odd → Even

  data Odd : Set where
    S : Even → Odd
```

Here the data types `Even` and `Odd` refer to each other in their definition.

2.1.8 BUILTIN and Primitive

BUILTIN and **primitive** are Agda's reserved key words for Agda built in types. Some inductive data types are built into Agda. The natural number data type is one of them, so in this case we are able to use the Agda built-in natural number data type, if we add the following declaration after the definition of \mathbb{N}

```
{-# BUILTIN NATURAL  $\mathbb{N}$  #-}
{-# BUILTIN SUC suc #-}
{-# BUILTIN ZERO zero #-}
```

where **NATURAL**, **SUC** and **ZERO** are exactly the names of Agda built in types for \mathbb{N} , `suc` and `zero` respectively. This means that Agda will use internally Haskell's efficient native natural numbers rather than working with natural numbers build-in from `suc` and `zero`. After this definition we can write as well `356 : \mathbb{N}` instead of

$$\underbrace{\text{suc (suc } (\dots (\text{suc zero}) \dots))}_{356}$$

which allow us as well to write `0` for the constructor `zero : \mathbb{N}` even in patterns, and Agda uses the built-in natural numbers used by the processor efficiently. Another example of a **BUILTIN** type in Agda is `List`

```

data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

{-# BUILTIN LIST List #-}
{-# BUILTIN NIL [] #-}
{-# BUILTIN CONS _::_ #-}

```

One can also use Agda built-in functions. For instance, the addition function for natural numbers is defined by recursion as follows

```

_+n_ : ℕ → ℕ → ℕ
n +n 0 = n
n +n suc m = suc (n +n m)

{-# BUILTIN NATPLUS _+n_ #-}

```

Once we add the **BUILTIN** declaration, Agda will treat $_+^n_$ as a built in function type which uses Haskell’s native operation $_+_$ rather than Agda’s inefficient $_+^n_$. Agda will check when reaching the **BUILTIN** statement that the definition of $_+^n_$ fulfils the equations of $_+^n_$.

Another way to use Agda’s built-in type is by Agda’s reserved key word **primitive**. For instance, `primIntegerPlus` is the addition function for **BUILTIN** Integers defined by

```

primitive
  primIntegerPlus : Int → Int → Int

```

Note that here `Int` is different to \mathbb{Z} in this thesis (which doesn’t make use of **BUILTIN**) since we use a different definition for `Integer`.

We use **BUILTIN** when we have an explicit definition and a directive which says: if the term is closed then use the native definition. If the term is not closed then use the explicit definition. For instance the addition function for natural numbers above ($2 +^n \text{ suc } x$) computes using the explicit definition to

$(\text{suc } (2 +^n x))$ and $(2 +^n 4)$ computes to 6 using the native implementation. **primitive** is like a **BUILTIN** without a definition, so the definition is like a **postulate**. In Agda we use **primitive** when one postulates an element of a function type and states that if the element is closed, then a native definition is executed. Otherwise it stays as a **postulate**. For instance, the addition function for **BUILTIN** Integer above $(\text{primIntegerPlus } (-2) 2)$ computes to 0 using the native definition. On the other hand $(\text{primIntegerPlus } (-2) x)$ evaluates to itself.

However, not many data types or functions are **BUILTIN** and **primitive** in Agda, one can check the Agda standard library in the Agda wiki [Nor09] for details.

2.1.9 Modules

In Agda each Agda file is demanded to be a single top-level module which contains all the declarations in the file. These declarations can also in turn be modules (a module can contain modules) by using Agda's reserved key word **module**. For instance, the Agda file `Nat.agda` which contains the definition of natural numbers is declared by having at the top of the file

```
module Nat where
```

One can also define a module which is parametrised by arbitrary types in the same way as **data** types can be parametrised. For instance, consider a parametrised module `A` as follows:

```
module A (r : ℝ)
  (Q : ℝ → Set)
where
  q : Digit → ℝ
  q = embedD

  one : ℕ
  one = 1

  -- position 1 --
  p1 : (r : ℝ) → (Q : ℝ → Set) → Digit → ℝ
  p1 = A.q
```

At position 1 $A.q$ has type of $(r : \mathbb{R}) \rightarrow (Q : \mathbb{R} \rightarrow \text{Set}) \rightarrow \text{Digit} \rightarrow \mathbb{R}$ and $A.one$ is has type $(r : \mathbb{R}) \rightarrow (Q : \mathbb{R} \rightarrow \text{Set}) \rightarrow \mathbb{N}$. One can give proof of $p1$ using function q in the module A since looking at the function q in module A from the outside, q takes the module parameters as additional arguments. Therefore, $q : (r : \mathbb{R}) \rightarrow (Q : \mathbb{R} \rightarrow \text{Set}) \rightarrow \text{Digit} \rightarrow \mathbb{R}$. Modules can also be opened by using Agda's reserved key word **open**. The functions inside the module become visible, i.e.

```
open A

  p2 : (r : ℝ) → (Q : ℝ → Set) → ℕ
  p2 = one
```

If one wants to export the contents of another module from the current module one can use Agda's reserved key word **public**, i.e.

```
open A public
```

Now we look at importing modules from other files. Using Agda's reserved key word **import** allows users to import modules (or files). However, it doesn't open the file automatically. One would need to **open** it if the definitions from that imported module is going to be used. Instead of writing two statements one can

just use the short form **open import**. For instance, in the Agda file `Nat.Agda` the two modules `Bool` and `Logic` are imported by the declaration

```
open import Bool
open import Logic using (flip; flip')
```

Agda's reserved key word **using** indicates that only functions `flip` and `flip'` are imported from the file `Logic.Agda`.

2.1.10 Compiled Version of Agda

Agda file can be compiled into Haskell in order to be executed more effectively (due to lazy evaluation and omission of computations needed for type checking only). One can import freely and has access to all Haskell data types and functions using Agda's reserved key words **COMPILED_DATA**, **COMPILED_TYPE** and **COMPILED**.

COMPILED_DATA is used for importing Haskell data types which requires that both the data type and its constructors need to be matched, i.e.

```
data Unit : Set where
  unit : Unit

{-# COMPILED_DATA Unit () () #-}
```

where the first argument to **COMPILED_DATA** is the name of the Agda data type (`Unit`) and the second is the corresponding Haskell type (`()`) which has only one constructor (`()`), the third argument.

There are as well abstract Haskell types exported by some libraries, that have no corresponding type definitions in Agda. One of these is the IO monad. In order to import such a Haskell type one can use Agda's reserved key word **COMPILED_TYPE** and the corresponding Agda type is simply postulated,

i.e

```
postulate
```

```
IO : Set → Set
```

```
{-# COMPILED_TYPE IO IO #-}
```

COMPILED is used when one wants to import a Haskell function. Similarly, the corresponding Agda type can be postulated in Agda, i.e.

```
postulate
```

```
String : Set
```

```
putStrLn : String → IO Unit
```

```
{-# COMPILED_TYPE String String #-}
```

```
{-# COMPILED putStrLn putStrLn #-}
```


Chapter 3

Real Numbers and Their Axiomatisation

If we want to prove theorems in formal systems we need a description of what a correct proof is. Then we can derive proofs from the given context by using logical rules and logical axioms which deal with the logical connectives such as $\vee, \wedge, \rightarrow, \forall, \exists$. Non-logical axioms are about content. A simple example would be a statement such as "John studies computer science". For instance, by giving two statements, "John studies computer science", "John lives in Swansea" with a logical connective \wedge (means "and"), we can form another statement "John studies computer science and John lives in Swansea" and using the \wedge - introduction rule we can derive this statement from the two axioms given before. Formally,

$$\frac{\text{"John studies computer science"} \quad \text{"John lives in Swansea"}}{\text{"John studies computer science"} \wedge \text{"John lives in Swansea"}} (\wedge \text{ - introduction rule})$$

In order to carry out proofs of theorems based on numbers, we need to characterise numbers and axiomatise them. In this chapter we will first look at the axioms for natural numbers, integers and rational numbers, then we will investigate axiomatisations of the real numbers. We will in this Chapter (except Section 3.1 when referring to Agda code) work in classical mathematics.

3.1 Natural Numbers, Integers, and Rational Numbers

The statement "John studies computer science" is an axiom in some specific context. When we talk about natural numbers, we talk about a non-logical object without context. There are different axiomatisations of natural numbers, from which we can derive different statements. The same applies to logic (there are different systems such as classical, intuitionistic or minimal logic which allow to derive different statements). Natural numbers based on the Peano axioms are a generally agreed axiomatisation of the natural numbers. The natural numbers can be characterized by the Peano axioms for \mathbb{N} , 0 and S ([Ebb91]):

- $0 \in \mathbb{N}$.
- if $n \in \mathbb{N}$ then $S(n) \in \mathbb{N}$.
- if $n \in \mathbb{N}$ then $S(n) \neq 0$.
- if $0 \in E$ and if it always follows from $n \in E$ that $S(n) \in E$ then $\mathbb{N} \subseteq E$.
- if $m, n \in \mathbb{N}$ then $S(m) = S(n)$ implies that $m = n$.

In Agda, our data type of natural numbers is given as follows

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

So \mathbb{N} is the set containing an element zero and which is closed under suc. The builtin recursion principle of Agda expresses that it is the least set with these properties and using case distinction we can prove $zero \neq suc\ n$ and $suc\ n == suc\ m$ implies $n == m$.

Integers \mathbb{Z} can be expressed as a pairs (s, n) s.t. $s \in \{+, -\}$ (the sign) and $n \in \mathbb{N}$ and such that if $s = -n$ then $n \neq 0$ (in order to avoid two notations for 0). For instance, $(+, 3)$ stands for +3, $(-, 5)$ for -5. In our definition of integers, instead of using natural numbers we introduce \mathbb{N}^+ (all natural numbers except

0) and represent integers as a union of three disjoint sets i.e. for $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$, $\mathbb{Z} = -\mathbb{N}^+ \cup \{0\} \cup +\mathbb{N}^+$. In Agda \mathbb{N}^+ , which is defined as the image of the $+1$ function applied to \mathbb{N} , and \mathbb{Z} are defined as follows :

```
data N+ : Set where
  _+1 : N -> N+
```

```
data Z : Set where
  pos : N+ -> Z
  neg : N+ -> Z
  zero : Z
```

Rational numbers \mathbb{Q} can be seen as the quotient a/b of two integers with the denominator b not equal to zero. For two rationals a/b and a'/b' , $a/b = a'/b'$ if and only if $a * b' = a' * b$. Since $a/b = -a / -b$ we can restrict b to be an element of \mathbb{N}^+ , and defined $\mathbb{Q} = \{z/n \mid z \in \mathbb{Z}, n \in \mathbb{N}^+\}$. In Agda, \mathbb{Q} is defined as follows :

```
data Q : Set where
  _%_ : Z -> N+ -> Q
```

We can see that there exist surjections $f : \mathbb{N} \rightarrow \mathbb{Z}$ and $f' : \mathbb{N} \rightarrow \mathbb{Q}$. So by using natural numbers we can denote integers and rational numbers, there are finite notations for integers and rationals. There is no surjection $f : \mathbb{N} \rightarrow \mathbb{R}$, so not all real numbers can be denoted finitely by natural numbers. We will not be able to give notations for real numbers, the axiomatisations are more complicated.

3.2 Real Numbers

In decimal representation real numbers are usually written as an infinite string of digits, in which a decimal numeral point is placed within. We can see it as an infinite sequence of digits with one dot. For instance, $\sqrt{2} = 1.41421356237\dots = a_0.a_1a_2a_3a_4a_5\dots$ where

$$a_0 = 1, a_1 = 4, a_2 = 1, a_3 = 4, a_4 = 2, a_5 = 1, a_6 = 3, a_7 = 5, \dots$$

However, there is not always a unique representation e.g. $1.0 = 0.99999999\dots$ (similar for binary representation). When the axiomatisation of the real numbers is based on an idea of what the real numbers are, decimal (or binary) representation is not suitable, since it is a priori not clear that all real numbers have a decimal representation (see the discussion at the beginning of Chapter 6). In fact constructively not all real numbers have a decimal representation. Instead one axiomatises abstractly the concept of real numbers. Then one can investigate using this axiomatisation whether each real number has a decimal representation or not, and with classical logic this will be the case.

3.2.1 Dedekind Cuts

Now we briefly discuss Dedekind cuts following the book by Ebbinghaus [Ebb91]. The first idea for defining the real numbers using Dedekind cuts is that a real number r is given by two nonempty sets of real numbers A, B such that $A = \{a \in \mathbb{R} \mid a < r\}$ and $B = \{a \in \mathbb{R} \mid r \geq a\}$. We cannot use this definition since we do not know what r is and what all $a \in \mathbb{R}$ are. One step is to replace \mathbb{R} by \mathbb{Q} so we get $A = \{a \in \mathbb{Q} \mid a < r\}$ and $B = \{b \in \mathbb{Q} \mid r \geq b\}$. We still don't know what r is. Instead we introduce axioms which characterise the sets of rationals which are formed this way [Ebb91]:

- Neither A nor B are empty.
- Every rational number belongs to one of two sets A, B .
- every element of A is less than every element of B .
- A has no largest element.

These axioms define what a Dedekind cut is. Note: This is the classical notion of a Dedekind cut. Constructively it is better not to assume that B is the complement of A . Instead one says that B has no minimal element, and (A, B) is located, i.e. for $p, q \in \mathbb{Q}$ s.t. $p < q$ we have $p \in A$ or $q \in B$. We will in this thesis only consider classical Dedekind cuts.

A Dedekind cut is a pair (A, B) of sets of nonempty rationals such that the above properties hold. Every Dedekind cut denotes a real number which is the

least real number not in A (so if it is a rational then it is in B). The cut number dividing them is the corresponding real number which is the least upper bound of A and also the greatest lower bound of B . The real numbers \mathbb{R} can be defined as the set of all Dedekind cuts of rationals. An example of a Dedekind cut for a real number is $\sqrt{2}$ which is written as a pair (A, B) such that $A = \{a \in \mathbb{Q} : a^2 < 2 \vee a \leq 0\}$ and $B = \{b \in \mathbb{Q} : b^2 \geq 2 \wedge b > 0\}$.

Definition 3.2.1. 1. A Dedekind cut is a pair of nonempty subsets A, B of \mathbb{Q} such that:

- $A \cup B = \mathbb{Q}$.
- If $a \in A$ and $b \in B$ then $a < b$.
- A contains no largest element.

2. A real number is a Dedekind cut.

Since B is the complement of A , B is determined by A . So we can identify a Dedekind cut with a set A s.t. $(A, \mathbb{Q} \setminus A)$ is a Dedekind cut. So by saying for a Dedekind cut α that $q \in \alpha$ we mean that if $\alpha = (A, B)$ then $q \in A$. Instead of writing (A, B) for a Dedekind cut, we write just A . (So for instance the Dedekind cut $\{q \mid q < 0\}$ denotes $(\{q \mid q < 0\}, \{q \mid q \geq 0\})$).

We can show using classical logic that the following is equivalent for $A \subseteq \mathbb{Q}$:

- (i) $(A, \mathbb{Q} \setminus A)$ is a Dedekind cut
- (ii) The following holds:
 - (1) $A \neq \emptyset, A \neq \mathbb{Q}$.
 - (2) If $a \in A, b \in \mathbb{Q}, b < a$, then $b \in A$.
 - (3) A has no largest element.

We call a set A fulfilling (ii) a Dedekind cut in the second sense. This equivalence doesn't hold constructively. If we define $B := \mathbb{Q} \setminus A$ then $\forall q \in \mathbb{Q}. q \in A \vee q \in B$ means $\forall q \in \mathbb{Q}. q \in A \vee \neg(q \in A)$ which is an instance of the principle of excluded middle.

Definition 3.2.2. For two Dedekind cuts $\alpha, \beta \in \mathbb{R}$

1. $0 := \{x \mid x < 0\}$. Then $0 < \alpha$ means $0 \subseteq \alpha \wedge 0 \neq \alpha$.

2. $\alpha < \beta$ if and only if $\alpha \subseteq \beta \wedge \alpha \neq \beta$.

3. $\alpha + \beta := \{x + y \mid x \in \alpha, y \in \beta\}$.

4. $\alpha - \beta := \{x - y \mid x \in \alpha, y \notin \beta\}$.

5. $-\alpha := \{-x \mid x > x' \notin \alpha\}$

6. $\alpha * \beta := \begin{cases} \{x * y \mid x \in \alpha, x \geq 0, y \in \beta\} & \text{if } \alpha > 0 \wedge \beta > 0; \\ \{x \mid x < 0\} & \text{if } \alpha = 0 \vee \beta = 0; \\ -((-\alpha) * \beta) & \text{if } \alpha < 0, \beta > 0; \\ -(\alpha * (-\beta)) & \text{if } \alpha > 0, \beta < 0; \\ (-\alpha) * (-\beta) & \text{if } \alpha < 0, \beta < 0; \end{cases}$

where $-((-\alpha) * \beta)$, $-(\alpha * (-\beta))$, $(-\alpha) * (-\beta)$ are defined by the first case and $-$ is defined as before.

7. We define the embedding $f : \mathbb{Q} \rightarrow \mathbb{R}$ be $f(q) = \{q' \in \mathbb{Q} \mid q' < q\}$, and identify q with this Dedekind cut.

We can show: if α, β are Dedekind cuts in the second sense, so are $\alpha + \beta$, $\alpha - \beta$, $-\alpha$, $\alpha * \beta$.

3.2.2 Cauchy Sequences

The second axiomatisation of real numbers is based on Cauchy sequences. We would like to construct real numbers by using sequences: a real number is given by a sequence of real numbers converging to it. A sequence $(a_n)_{n \in \mathbb{N}}$ of real numbers converges to the number r if and only if for each real number $\epsilon > 0$, there exists $N \in \mathbb{N}$ such that whenever $n \geq N$, $|a_n - r| < \epsilon$. Now $\epsilon \in \mathbb{R}$ can be replaced by $\epsilon' \in \mathbb{Q}$ since for every real number $\epsilon > 0$ there exists a rational $\epsilon' > 0$ s.t. $0 < \epsilon' < \epsilon$. Since we haven't defined \mathbb{R} yet, we replace it by: a real number is given by a sequence of rationals which has this property. (We will afterwards

show that if real numbers are given by Cauchy sequences of rationals, all Cauchy sequences of real numbers have a limit. (This is the proof that $\mathbb{Q}'' \subseteq \mathbb{Q}'$, see Chapter 5, Cauchy Reals).

We still have the problem that without having real numbers, we don't know what it means for a sequence to converge to a real number. Instead we investigate the property needed in order for a sequence to converge without having its limit available. This notion is called Cauchy sequence. A Cauchy sequence is a sequence whose elements all become arbitrarily close to one another eventually. More precisely, a Cauchy sequence is a sequence $(a_n)_{n \in \mathbb{N}}$ s.t. for any positive number $\epsilon > 0$, there is an integer N such that for any n and m are larger then N , the distance from a_n to a_m is always less than ϵ .

Definition 3.2.3. 1. A sequence $(a_n)_{n \in \mathbb{N}}$ of real numbers converges to the number r if and only if

$$\forall \epsilon \in \mathbb{Q}. \epsilon > 0 \rightarrow \exists N \in \mathbb{N}. \forall n \geq N. |a_n - r| < \epsilon$$

2. A sequence $(a_n)_{n \in \mathbb{N}}$ is called a Cauchy sequence (or Cauchy) if and only if

$$\forall \epsilon \in \mathbb{Q}. \epsilon > 0 \rightarrow \exists N \in \mathbb{N}. \forall n, m \in \mathbb{N}. \forall n, m \geq N. |a_n - a_m| < \epsilon$$

3. The Cauchy reals are given as sequences $(a_n)_{n \in \mathbb{N}}$ such that $(a_n)_{n \in \mathbb{N}}$ is a Cauchy sequence of rationals.

4. For two Cauchy real numbers $(a_n)_{n \in \mathbb{N}}$ and $(b_n)_{n \in \mathbb{N}}$, we define $(a_n)_{n \in \mathbb{N}}$ is equal to $(b_n)_{n \in \mathbb{N}}$ if and only if

$$\forall \epsilon \in \mathbb{Q}. \epsilon > 0 \rightarrow \exists N \in \mathbb{N}. \forall n \geq N. |a_n - b_n| < \epsilon$$

5. We define $+$, $-$, $*$ on the Cauchy reals and embedding of \mathbb{Q} into the Cauchy reals as follows:

- $((a_n)_{n \in \mathbb{N}}) + ((b_n)_{n \in \mathbb{N}}) := (a_n + b_n)_{n \in \mathbb{N}}$.
- $((a_n)_{n \in \mathbb{N}}) - ((b_n)_{n \in \mathbb{N}}) := (a_n - b_n)_{n \in \mathbb{N}}$.

- $((a_n)_{n \in \mathbb{N}}) * ((b_n)_{n \in \mathbb{N}}) := (a_n * b_n)_{n \in \mathbb{N}}$.
- We define the embedding $f : \mathbb{Q} \rightarrow \mathbb{R}$ by $f(q) = (q)_{n \in \mathbb{N}}$, and identify q with the Cauchy real $f(q) = (q)_{n \in \mathbb{N}}$.

6. We say a set A is Cauchy complete if every Cauchy sequence within A has a limit within A itself.

Cauchy reals are (classically) isomorphic to the Dedekind reals. This equivalence holds since both are Archimedean complete ordered field. We therefore introduce the notions field, ordered field, complete ordered field and Archimedean field in the following

Definition 3.2.4. 1. A field is a set F together with operations

- $_{-}+_{} : F \rightarrow F \rightarrow F$
- $_{-}*_{-} : F \rightarrow F \rightarrow F$
- $0 : F$
- $1 : F$
- $_{-}- : F \rightarrow F$
- $_{-}^{-1} : \{f : F \mid f \neq 0\} \rightarrow F$

such that for all x, y , and z in F the following axioms hold:

- $0 \neq 1$
- $x + (y + z) = (x + y) + z$ and $x * (y * z) = (x * y) * z$ (associativity)
- $x + y = y + x$ and $x * y = y * x$ (commutativity)
- $x * (y + z) = (x * y) + (x * z)$ (distributivity)
- $x + 0 = x$ and $x * 1 = x$ (identity)
- $x + (-x) = 0$ (inverse for +)
- if $x \neq 0$ then $x * x^{-1} = 1$ (inverse for *)

Definition 3.2.5. 1. A set F is linearly ordered under \leq , where \leq is a binary relation on F , if for all x, y and z in F the following statements hold:

- (a) $x \leq y$ or $y \leq x$ (totality)
- (b) if $x \leq y$ and $y \leq z$ then $x \leq z$ (transitivity)
- (c) If $x \leq y$ and $y \leq x$ then $x = y$ (antisymmetry)

2. A field $(F, +, *)$ with a linear order \leq on F is a linearly ordered field $(F, +, *, \leq)$ if the following holds:

- $x \leq y$ then $z + x \leq z + y$.
- $0 \leq x$ and $0 \leq y$ then $0 \leq x * y$.

Definition 3.2.6. A complete ordered field is a set $(F, +, *, \leq)$ which satisfies the following properties:

- $(F, +, *, \leq)$ is a linearly ordered field.
- Completeness: every non-empty subset of F , bounded above, has a supremum in F .

Definition 3.2.7. Let $(F, +, *, \leq)$ be a complete ordered field. F is Archimedean, if for $r, s \in F$ s.t. $r > 0$ there is a natural number n such that

$$s < \underbrace{r + \dots + r}_{n \text{ times}}$$

From the definition of Archimedean property we get the following property:

$$\forall \epsilon, r \in F. \epsilon > 0 \rightarrow \exists n \in \mathbb{N}. r/f(n) < \epsilon, \text{ where } f(n) = \underbrace{1 + \dots + 1}_{n \text{ times}}$$

There is only one complete ordered Archimedean field up to isomorphism.

So far we have demonstrated two ways for constructing real numbers, Dedekind cuts and Cauchy sequences. In the book [Ebb91]P.47 §5.2, for ordered fields it is shown that the following statements are equivalent

- if (α, β) is a cut in F (the definition of Dedekind cuts is satisfied when elements of F instead of rationals are taken) then β contains a minimum element.

- the ordering on F is Archimedean and every Cauchy sequence of elements of F converges in F .

[Ebb91]P.50 §5.3 contains a the proof that all Archimedean complete ordered fields are isomorphic and the isomorphisms are unique. The Dedekind reals and the Cauchy reals are both Archimedean complete ordered fields and therefore they are isomorphic.

In the following, we will axiomatise in Chapter 5 the real numbers \mathbb{R} in the language of Agda and investigate some properties of the Cauchy reals. In Chapter 6, we introduce binary signed digit real numbers presentations $\sim\mathbb{R}$ using a coinductive definition in Agda.

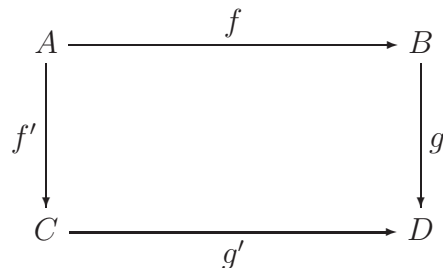
Chapter 4

Coalgebras and Coinduction

This Chapter presents a background knowledge of coalgebras and coinduction. The types $\sim\mathbb{R}$ and `Stream` in Chapter 7 will be defined as codata types which is Agda's version of coalgebras. In the following

- we introduce coalgebras as the dual of algebras (Section 4.1).
- we discuss the use of final and weakly final coalgebras as the dual of initial algebras which represents to algebraic data type.
- we introduce codata types as a the notation for coalgebras in Agda (Section 4.2).
- we discuss how to model coalgebras set theoretically. (Section 4.3).

We develop the theory of coalgebras in the context of type theory. Here the commutativity of a diagram such that



is to be understood as

$$\forall a : A. g (f a) == g' (f' a)$$

or written in type theory

$$(a : A) \rightarrow g (f a) == g' (f' a)$$

is provable, where $==$ is the standard propositional equality of type theory (see Section 2.1.6).

Uniqueness of a function $g : B \rightarrow C$ s.t. some property holds is to be understood as follows: If g, g' are two functions with this property then $(b : B) \rightarrow g b == g' b$ is provable.

By F being an endofunctor (or briefly a functor) we mean that we have the object part of F_{obj} s.t. $F_{obj} : \text{Set} \rightarrow \text{Set}$ (note that Set is the collection of small types in type theory) and a morphism part $F_{mor} : \{A B : \text{Set}\} \rightarrow (A \rightarrow B) \rightarrow (F_{mor} A \rightarrow F_{mor} B)$ s.t. the functor laws follow pointwise e.g.

$$F_{mor} (f \circ g) = (F_{mor} f) \circ (F_{mor} g)$$

means

$$(a : A) \rightarrow F_{mor} (f \circ g) a = ((F_{mor} f) \circ (F_{mor} g)) a$$

We write F instead of F_{obj} or F_{mor} . We usually define functors by giving their object part, the morphism part is usually obvious.

Let 1 be the one element set containing element $*$. $A + B$ is the disjoint union of A and B with elements $(\text{inl } a)$ for $a : A$ and $(\text{inr } b)$ for $b : B$.

Definition 4.0.8. *Strictly positive functor are defined as follows:*

- (a) $\lambda X.X$ and $\lambda X.A$ where A is a set A not depending on X , are strictly positive.
- (b) If $A : \text{Set}$ and F, G are strictly positive. so are

$$\begin{aligned} &\lambda X.F(X) + G(X) \\ &\lambda X.F(X) \times G(X) \\ &\lambda X.A \rightarrow F(X) \end{aligned}$$

We could develop this chapter as well in a general category-theoretic setting, then Set above would be replaced by a category with suitable conditions (e.g. we need the existence products, coproducts and a generalised version of equality set). We won't analyse this further.

In Section 4.3 we explore the relationship to set theory.

4.1 Initial Algebras and Final Coalgebras

In Computer Science initial algebras and final coalgebras provide the framework for inductive and coinductive data structures respectively. Let F be an endofunctor. An initial algebra is an initial object in the category of F -algebras and a final coalgebra is a final object in the category of F -coalgebras, where F -(co)algebras are defined below. In the category of ordered sets an initial object is a least element, whereas a final object is a largest element. Therefore, one can consider initial F -algebras as smallest F -algebras and final F -coalgebras as largest F -coalgebras. The initiality of an initial F -algebra means there exists exactly one morphism from it to any other F -algebra and the finality of a final coalgebra means there exists exactly one morphism from any other F -coalgebra to it.

Definition 4.1.1. 1 An F -algebra is a pair (A, f) where A is a set and $f : F(A) \rightarrow A$.

2 A morphism from an F -algebra (A, f) to an F -algebra (B, g) is a function $h : A \rightarrow B$ s.t. the following diagram commutes

$$\begin{array}{ccc}
 F(A) & \xrightarrow{f} & A \\
 \downarrow F(h) & & \downarrow h \\
 F(B) & \xrightarrow{g} & B
 \end{array}$$

3 A weakly initial F -algebra is an F -algebra (A, f) such that for any other F -algebra (B, g) , i.e. for any $B : \text{Set}$, $g : F(B) \rightarrow B$, there exists a morphism

$h : (A, f) \rightarrow (B, g)$, i.e a function $h : A \rightarrow B$ such that the following diagram commutes:

$$\begin{array}{ccc}
 F(A) & \xrightarrow{f} & A \\
 F(h) \downarrow & & \downarrow h \\
 F(B) & \xrightarrow{g} & B
 \end{array}$$

4 An initial F -algebra is defined as a weakly initial F -algebra but the morphism h given above which was supposed to exist is additionally assumed to be unique.

The set of natural numbers with zero and successor is an initial algebra of the functor $1 + _$ where 1 is one element set. Let for $a : X$, $g : X \rightarrow X$, $[a, g]$ be the function $f : 1 + X \rightarrow X$ s.t. $f(\text{inl } *) = a$ and $f(\text{inr } (x)) = g(x)$. Then we get $[\text{zero}, \text{suc}] : 1 + \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\text{zero} : \mathbb{N}$ and $\text{suc} : \mathbb{N} \rightarrow \mathbb{N}$. From the natural numbers being an F -algebra we can derive the principle of iteration, recursion and induction. We will in the following derive those principles for the natural numbers. Note: The iteration, recursion and induction principles can be defined for all other initial F -algebras for strictly positive F e.g. for $\text{List}(A)$. We consider here only the initial F -algebra for F as above, i.e. $F(X) = 1 + X$. The principle of iteration is as follows: Assume $a : X$ and $f : X \rightarrow X$. Let $\tilde{f} : 1 + X \rightarrow X$, $\tilde{f}(\text{inl } *) = a$ and $\tilde{f}(\text{inr } x) = f x$. Then by the initiality of \mathbb{N} there exists exactly one morphism $h : \mathbb{N} \rightarrow X$ such that the following diagram commutes:

$$\begin{array}{ccc}
 F(\mathbb{N}) = 1 + \mathbb{N} & \xrightarrow{[\text{zero}, \text{suc}]} & \mathbb{N} \\
 1 + h \downarrow & & \downarrow h \\
 F(X) = 1 + X & \xrightarrow{\tilde{f}} & X
 \end{array}$$

Therefore, there exists exactly a function $h : \mathbb{N} \rightarrow X$ s.t.

$$\begin{aligned} h(\text{zero}) &= h([\text{zero}, \text{suc}] (\text{inl } *)) = \tilde{f}((1 + h) (\text{inl } *)) \\ &= \tilde{f} (\text{inl } *) = a \\ h(\text{suc } x) &= h([\text{zero}, \text{suc}] (\text{inr } x)) = \tilde{f}((1 + h) (\text{inr } x)) \\ &= \tilde{f} (\text{inr } (h x)) = f (h x) \end{aligned}$$

So, if $X : \text{Set}$, $a : X$ and $f : X \rightarrow X$ then there exists a unique function $h : \mathbb{N} \rightarrow X$ s.t.

$$h(n) = f^n(a) = \underbrace{f(f(f(f(\dots f(a))))}_{n \text{ times}}$$

The principle of recursion is obtained from iteration. The principle of recursion is as follows: Assume $a : X$ and $f : \mathbb{N} \rightarrow X \rightarrow X$. Then there exists a unique function $g : \mathbb{N} \rightarrow X$ s.t. $g 0 = a$ and $g (n + 1) = f n (g n)$. It can be derived from the principle of iteration as follows: Assume a, f as before. Let $X' := \mathbb{N} \times X$, $a' := \langle 0, a \rangle : X'$ and $f' : 1 + X' \rightarrow X'$, $f' (\text{inl } *) = \langle 0, a \rangle$, $f' (\text{inr } (\langle n, x \rangle)) = \langle n + 1, f n x \rangle$. By iteration there exists a unique morphism $\tilde{g} : \mathbb{N} \rightarrow X'$ such that

$$\begin{array}{ccc} F(\mathbb{N}) = 1 + \mathbb{N} & \xrightarrow{[\text{zero}, \text{suc}]} & \mathbb{N} \\ \downarrow 1 + \tilde{g} & & \downarrow \tilde{g} \\ F(X') = 1 + X' & \xrightarrow{f'} & X' \end{array}$$

Since $X' := \mathbb{N} \times X$, we get

$$\begin{array}{ccc}
 F(\mathbb{N}) = 1 + \mathbb{N} & \xrightarrow{[\text{zero}, \text{suc}]} & \mathbb{N} \\
 \downarrow 1 + \tilde{g} & & \downarrow \tilde{g} \\
 F(X') = 1 + X' & \xrightarrow{f'} & X' \\
 \downarrow 1 + \pi_0 & & \downarrow \pi_0 \\
 F(\mathbb{N}) = 1 + \mathbb{N} & \xrightarrow{[\text{zero}, \text{suc}]} & \mathbb{N}
 \end{array}$$

Both diagrams commute, the top one is by the definition of \tilde{g} and the bottom one is by the definition of f' namely

$$\begin{aligned}
 \pi_0 (f' (\text{inl } *)) &= \pi_0 (< 0, a >) = 0 \\
 &= [\text{zero}, \text{suc}] (\text{inl } *) = [\text{zero}, \text{suc}] ((1 + \pi_0) (\text{inl } *)) \\
 \pi_0 (f' (\text{inr } (n, x))) &= \pi_0 (< n + 1, f \ n \ x >) = n + 1 \\
 &= [\text{zero}, \text{suc}] (\text{inr } n) = [\text{zero}, \text{suc}] ((1 + \pi_0) (\text{inr } < n, x >))
 \end{aligned}$$

$\pi_0 \circ \tilde{g}$ is an F-algebra morphism $\mathbb{N} \rightarrow \mathbb{N}$. id is another F-algebra morphism $\mathbb{N} \rightarrow \mathbb{N}$. By uniqueness there exists only one such morphism. Therefore, $\pi_0 \circ \tilde{g} = \text{id}$. We get $\pi_0 (\tilde{g} \ n) = n$. Connect

$$\begin{array}{ccc}
 F(\mathbb{N}) = 1 + \mathbb{N} & \xrightarrow{[\text{zero}, \text{suc}]} & \mathbb{N} \\
 \downarrow 1 + \tilde{g} & & \downarrow \tilde{g} \\
 F(X') = 1 + X' & \xrightarrow{f'} & X' \\
 \downarrow 1 + \pi_1 & & \downarrow \pi_1 \\
 1 + X & & X
 \end{array}$$

Let $g\ n := \pi_1 (\tilde{g}\ n)$. Since $\pi_0 (\tilde{g}\ n) = n$ we have $\tilde{g}\ n = \langle n, (g\ n) \rangle$ and we get

$$\begin{aligned} g\ 0 &= \pi_1 (\tilde{g}\ 0) = \pi_1 (\langle 0, a \rangle) = a \\ g\ (n + 1) &= \pi_1 (\tilde{g}\ ([\text{zero}, \text{suc}] (\text{inr}\ n))) = \pi_1 (f' ((1 + \tilde{g}) (\text{inr}\ n))) \\ &= \pi_1 (f' (\text{inr} (\tilde{g}\ n))) = \pi_1 (f' (\text{inr} (\langle n, g\ n \rangle))) \\ &= \pi_1 (\langle n + 1, f\ n\ (g\ n) \rangle) = f\ n\ (g\ n) \end{aligned}$$

The principle of induction is dependent recursion: if we have $C : \mathbb{N} \rightarrow \text{Set}$, $f_0 : C\ 0$ and $f_1 : (n : \mathbb{N}) \rightarrow C\ n \rightarrow C\ (\text{suc}\ n)$ then we get $g : (n : \mathbb{N}) \rightarrow C\ n$ s.t. $g\ 0 = f_0$, $g\ (\text{suc}\ n) = f_1\ n\ (g\ n)$. The induction principle is defined as follows:

$$\begin{array}{l} C : \mathbb{N} \rightarrow \text{Set} \\ \text{Base0} : C\ 0 \\ \text{StepS} : (n : \mathbb{N}) \rightarrow C\ n \rightarrow C\ (\text{S}\ n) \\ \hline \text{Ind}(C, \text{Step0}, \text{StepS}) : (n : \mathbb{N}) \rightarrow C\ n \end{array}$$

together with the following equalities

$$\begin{aligned} \text{Ind}(C, \text{Step0}, \text{StepS})\ 0 &= \text{Base0} \\ \text{Ind}(C, \text{Step0}, \text{StepS})\ (\text{S}\ n) &= \text{StepS}\ n\ (\text{Ind}(C, \text{Step0}, \text{StepS})\ n) \end{aligned}$$

The induction principle can be derived from \mathbb{N} being an initial algebra as follows: Let $X' := (n : \mathbb{N}) \times (C\ n)$, $f' : 1 + X' \rightarrow X'$ s.t. $f' (\text{inl}(\ast)) = \langle 0, \text{Base0} \rangle$ and $f' (\text{inr}(\langle n, x \rangle)) = \langle n + 1, \text{StepS}\ n\ x \rangle$. By \mathbb{N} being an initial F-algebra there exists a unique morphism $\tilde{g} : \mathbb{N} \rightarrow X'$ such that

$$\begin{array}{ccc} F(\mathbb{N}) = 1 + \mathbb{N} & \xrightarrow{[\text{zero}, \text{suc}]} & \mathbb{N} \\ \downarrow 1 + \tilde{g} & & \downarrow \tilde{g} \\ F(X') = 1 + X' & \xrightarrow{f'} & X' \end{array}$$

As for the derivation of the recursion principle above, it follows $\pi_0 \circ \tilde{g} = \text{id}$ i.e. $\pi_0 (\tilde{g}\ n) = n$. By the definition of X' we get $\tilde{g}\ n : X' := (n : \mathbb{N}) \times (C\ n)$ and $\pi_1 (\tilde{g}\ n) : C\ (\pi_0 (\tilde{g}\ n)) = C\ n$. Therefore, $g := \pi_1 \circ \tilde{g} : (n : \mathbb{N}) \rightarrow C\ n$ and one

easily verifies that $g\ 0 = \text{Base}0$ and $g\ (n + 1) = \text{Step}S\ n\ (g\ n)$. So the induction principle can be derived from the principle of initial F-algebras.

On the other hand if we have an F-algebra together with the induction principle (without any uniqueness condition) then we get an initial F-algebra. Assume an F-algebra (A, f) s.t. $A : \text{Set}$ and $f : F(A) \rightarrow A$. Then by the induction principle there exists a morphism $g : \mathbb{N} \rightarrow A$ such that the following commutes:

$$\begin{array}{ccc}
 F(\mathbb{N}) & \xrightarrow{[\text{zero}, \text{suc}]} & \mathbb{N} \\
 \downarrow F(g) & & \downarrow g \\
 F(A) & \xrightarrow{f} & A
 \end{array}$$

($g\ 0 = f\ (\text{inl}\ *)$, $g\ (n + 1) = f\ \text{inr}\ (g\ n)$) The uniqueness of g follows by induction. Assume g, g' both make the diagram above commute, i.e. $g\ 0 = f\ (\text{inl}\ *)$ and $g\ (n + 1) = f\ (\text{inr}\ (g\ n))$, similar for g' . By induction we can define $p : (n : \mathbb{N}) \rightarrow g\ n ==_A g'\ n$ such that

$$\begin{aligned}
 p\ 0 &= \text{refl} : \underbrace{g\ 0}_{f\ (\text{inl}\ *)} ==_A \underbrace{g'\ 0}_{f\ (\text{inl}\ *)} \\
 p\ (n + 1) &= \text{lem}\ n\ (p\ n) : \underbrace{g\ (n + 1)}_{f\ (\text{inr}\ (g\ n))} ==_A \underbrace{g'\ (n + 1)}_{f\ (\text{inr}\ (g'\ n))}
 \end{aligned}$$

where $\text{lem}\ n\ (p\ n)$ proves from $p\ n : g\ n ==_A g'\ n$ that $f\ (\text{inr}\ (g\ n)) ==_A f\ (\text{inr}\ (g'\ n))$. Therefore, induction is equivalent to the principle of the initial algebras.

(Weakly) final coalgebras are the dual of initial algebras (weakly final means the finality without the uniqueness).

Definition 4.1.2. 1 An F-coalgebra is a pair (A, f) such that $A : \text{Set}$ and $f : A \rightarrow F(A)$.

2 A morphism from an F-coalgebra (A, f) to an F-coalgebra (B, g) is a func-

tion $h : A \rightarrow B$ s.t. the following diagram commutes

$$\begin{array}{ccc}
 A & \xrightarrow{f} & F(A) \\
 \downarrow h & & \downarrow F(h) \\
 B & \xrightarrow{g} & F(B)
 \end{array}$$

3 A weakly final coalgebra is an F -coalgebra (A, f) where A is a set and $f : A \rightarrow F(A)$ such that for all F -coalgebras (B, g) , i.e. for any $B : \text{Set}$, $g : B \rightarrow F(B)$, there exists a morphism $h : (B, g) \rightarrow (A, f)$ which is a function $h : B \rightarrow A$ such that the following diagram commutes:

$$\begin{array}{ccc}
 B & \xrightarrow{g} & F(B) \\
 \downarrow h & & \downarrow F(h) \\
 A & \xrightarrow{f} & F(A)
 \end{array}$$

4 A final F -coalgebra is defined as a weakly final coalgebra but the morphism h given above is additionally assumed to be unique.

Let F be such that \mathbb{N} is the initial F -algebra. The final F -coalgebra is a set \mathbb{N}^∞ (\mathbb{N}^∞ is called the set of co-natural numbers) together with a function $\text{elim} : \mathbb{N}^\infty \rightarrow 1 + \mathbb{N}^\infty$ (\mathbb{N}^∞ is co-natural numbers) such that for any coalgebra (X, f) , i.e. $X : \text{Set}$, $f : X \rightarrow 1 + X$, there exists a unique morphism $h : (X, f) \rightarrow (\mathbb{N}^\infty, \text{elim})$, i.e. a function $h : X \rightarrow \mathbb{N}^\infty$ such that the following diagram commutes:

$$\begin{array}{ccc}
 X & \xrightarrow{f} & F(X) = 1 + X \\
 \downarrow h & & \downarrow 1 + h \\
 \mathbb{N}^\infty & \xrightarrow{\text{elim}} & F(\mathbb{N}^\infty) = 1 + \mathbb{N}^\infty
 \end{array}$$

Let $0' := \text{inl } *$ and $S' x = \text{inr } x$. So there exists a function $h : X \rightarrow \mathbb{N}^\infty$ s.t. $\text{elim } (h x) = 0'$, if $f x = 0'$, and $\text{elim } (h x) = S' (h x')$ if $f x = S' x'$. So elim defines for every $n : \mathbb{N}^\infty$ whether n has the shape $0'$ or $S' m$ for same $m : \mathbb{N}^\infty$. A discussion how to develop \mathbb{N}^∞ on Set theory can be found in Section 4.3.

4.2 Coalgebras and Codata in Agda

In Agda with final coalgebras equality becomes undecidable but with initial algebras equality stays decidable. In a recent talk [Set10a] there was some discussion about this namely that in type theory \mathbb{N} is only the initial algebra w.r.t. the extensional equality on the function space.

More precisely the situation is as follows: \mathbb{N} is an initial algebra with the decidable definitional equality on \mathbb{N} and referring to the equality of functions in the diagram of initial algebras as the extensional equality. So if $f, f' : \mathbb{N} \rightarrow X$ are two solutions of the diagram we get that for all $n : \mathbb{N}$ we have $f n == f' n$ w.r.t. propositional equality (see Section 2.1.6) but usually not $f == f'$.

\mathbb{N}^∞ is a final coalgebra only if we instead of having the definitional equality on it we have bisimulation as equality, which is undecidable. An example is as follow: we define $n^\infty : \mathbb{N}^\infty$ s.t. $\text{elim } n^\infty = S' n^\infty$ and define for $f : \mathbb{N} \rightarrow \mathbb{N}$, $g : \mathbb{N} \rightarrow \mathbb{N}^\infty$ s.t. $\text{elim } (g n) = 0'$ if $f n = 0$, $\text{elim } (g n) = S' (g (n + 1))$ if $f n > 0$. Then $(g 0)$ is bisimilar to n^∞ if and only if $f n > 0$ for all n , which is undecidable.

So, whereas \mathbb{N} is the initial algebra taking as equality on \mathbb{N} the definitional one and only on the arrows in the diagram extensional equality, \mathbb{N}^∞ is the final coalgebra only if we take an undecidable equality on \mathbb{N}^∞ , which means by enforcing special equality on \mathbb{N}^∞ .

Agda uses initial algebras and only weakly final coalgebras. (Weakly final coalgebras are the dual of a weakly initial algebra which is an initial algebra without uniqueness). We can define \mathbb{N}^∞ by using "codata" as a data type of

infinite objects as follows

$$\begin{aligned} \text{codata } \mathbb{N}^\infty : \text{Set where} \\ 0^\infty : \mathbb{N}^\infty \\ S^\infty : \mathbb{N}^\infty \rightarrow \mathbb{N}^\infty \end{aligned}$$

which stands for the weakly final coalgebra $(\mathbb{N}^\infty, \text{elim})$ s.t. $\text{elim} : \mathbb{N}^\infty \rightarrow 0' + S'(\mathbb{N}^\infty)$, i.e. $\text{elim } n = 0'$ or $\text{elim } n = S' n$. Here, $0' + S'(\mathbb{N}^\infty)$ stands for some set X such that in Agda

$$\begin{aligned} \text{data } X : \text{Set where} \\ 0' : X \\ S' : \mathbb{N}^\infty \rightarrow X \end{aligned}$$

which is the labelled disjoint union. For \mathbb{N}^∞ we have the new diagram as follows

$$\begin{array}{ccc} X & \xrightarrow{f} & 0' + S'(X) \\ \downarrow h & & \downarrow 0' + S'(h) \\ \mathbb{N}^\infty & \xrightarrow{\text{elim}} & 0' + S'(\mathbb{N}^\infty) \end{array}$$

Furthermore, the codata definition of \mathbb{N}^∞ means that one defines $0^\infty : \mathbb{N}^\infty$ s.t. $\text{elim } 0^\infty = 0'$ and $S^\infty : \mathbb{N}^\infty \rightarrow \mathbb{N}^\infty$ s.t. $\text{elim } (S^\infty n) = S' n$. So after applying elim every co-natural numbers is either $0'$ or $S' n$. The principle of coiteration essentially means guarded recursion, see [HS05]. Assume $f : X \rightarrow 1 + X$ as above, the commuting diagram shows that if $f x = 0'$ then

$$\text{elim } (h x) = (0' + S' h) (f x) = (0' + S' h) (0') = 0'$$

If $f x = S' (x')$ for some $x' : X$ then

$$\text{elim } (h x) = (0' + S' h) (f x) = (0' + S' h) (S' (x')) = S' (h x')$$

The principle of guarded recursion allows one to define $h : X \rightarrow \mathbb{N}^\infty$ such that

$$\text{elim } (h \ x) = 0'$$

or

$$\text{elim } (h \ x) = S' (h \ x')$$

We can generalise guarded recursion to the principle that we can define $h : X \rightarrow \mathbb{N}^\infty$ s.t.

$$\text{elim } (h \ x) = 0'$$

or

$$\text{elim } (h \ x) = S' (S^{\infty \ k} (h \ x'))$$

for some $k : \mathbb{N}$ and $x' : X$ or

$$\text{elim } (h \ x) = S' (S^{\infty \ k} (0^\infty))$$

In Agda one writes for this $h : X \rightarrow \mathbb{N}^\infty$ such that

$$h \ x = 0^\infty$$

or for some $k : \mathbb{N}$ and $x' : X$

$$h \ x = S^\infty (S^{\infty \ k} (h \ x'))$$

or

$$h \ x = S^\infty (S^{\infty \ k} (0^\infty))$$

However, if \mathbb{N}^∞ is only a weakly final coalgebra then there can be many $n' : \mathbb{N}^\infty$ s.t. $\text{elim } n' = S' \ n$ for some $n : \mathbb{N}^\infty$. So if $\text{elim } (h \ x) = S' (S^{\infty \ k} (h \ x'))$ for some $k : \mathbb{N}, x' : X$ it doesn't mean that $h \ x = S^\infty (S^{\infty \ k} (h \ x'))$. In a previous version of Agda, which we think had a better representation of weakly final coalgebras in dependent type theory, one wrote $a \sim b$ for $\text{elim } a = \text{elim } b$. Then $h \ x \sim S^\infty (S^{\infty \ k} (h \ x'))$ means the same as $\text{elim } (h \ x) = S' (S^{\infty \ k} (h \ x'))$ and one can omit

occurrences of S' : $h\ x \sim S^\infty (S^{\infty k} (h\ x'))$ means

$$\text{elim } (h\ x) = \text{elim } (S^\infty (S^{\infty k} (h\ x'))) = S' (S^{\infty k} (h\ x'))$$

So for instance,

$$\begin{aligned} f &: \mathbb{N} \rightarrow \mathbb{N}^\infty \\ f\ 0 &\sim 0^\infty \\ f\ (S\ n) &\sim S^\infty (f\ (S\ n)) \end{aligned}$$

means

$$\begin{aligned} f &: \mathbb{N} \rightarrow \mathbb{N}^\infty \\ \text{elim } (f\ 0) &= 0' \\ \text{elim } (f\ (S\ n)) &= S' (f\ (S\ n)) \end{aligned}$$

One can define $h : 1 \rightarrow \mathbb{N}^\infty$ s.t. $h\ (*) \sim S^\infty (h\ (*))$ which means if one make case distinction on $h\ (*)$ then it has the shape $S^\infty(h\ (*))$. One can apply elim several times and find out

$$\begin{aligned} h\ (*) &\text{ is of the shape } && : S^\infty \\ \text{where } n &\text{ is of the shape } && : S^\infty\ n' \\ \text{where } n' &\text{ is of the shape } && : S^\infty\ n'' \\ &&& \vdots \end{aligned}$$

but each unpacking requires to apply elim so we don't have

$$h\ (*) = S^\infty (S^\infty (S^\infty (S^\infty (\dots))))$$

which would violate normalisation. $h\ (*) \sim S^\infty (h\ (*))$ which means

$$\text{elim } (h\ (*)) = S' (h\ (*))$$

" \sim " was replaced by " $=$ " in Agda version 2.2.4 (which is used in this thesis). Without " \sim " we get $h\ (*) = S^\infty (h\ *) = S^\infty (S^\infty (h\ *)) = \dots = S^\infty (S^\infty (S^\infty (S^\infty (\dots)))$. This destroys normalisation since $h\ (*) \longrightarrow S^\infty (h\ (*)) \longrightarrow S^\infty (S^\infty (h\ (*))) \longrightarrow \dots$. Therefore, if we take this $=$ verbally, Agda is non-normalising [Set09]. The solution is that $=$ when used in guarded

recursion is not to be taken as definitional equality (and means essentially \sim).

A new approach to defining codata types is taken in the newest version of Agda (which isn't used in this thesis). In the Agda standard library version 0.3 which is tested with Agda version 2.2.6, "codata" defines the symbol " ∞ " in the library file "Coinduction.agda" as follows

```
codata  $\infty$  (A : Set) : Set  where
  #- : A  $\rightarrow$   $\infty$  A
```

```
b :  $\forall$  {A : Set}  $\rightarrow$   $\infty$  A  $\rightarrow$  A
b (# a) = a
```

where ∞A denotes arguments in a data type for which infinite recursion is possible. Since ∞A and A are different we need functions which covert between them such that $b : \infty A \rightarrow A$ and $\# : A \rightarrow \infty A$. So b is the morphism in the coalgebra and $\# a$ is an element of ∞A such that $b (\# a) = a$.

The idea is that the user, instead of using the codata definition of \mathbb{N}^∞ earlier, should define it as follows by importing the library file defining " ∞ ":

```
open import Coinduction
```

```
data  $\mathbb{N}^\infty$  : Set  where
  0 $^\infty$  :  $\mathbb{N}^\infty$ 
  S $^\infty$  : ( $\infty$   $\mathbb{N}^\infty$ )  $\rightarrow$   $\mathbb{N}^\infty$ 
```

So one can write that

```
inf :  $\mathbb{N}^\infty$ 
inf = S $^\infty$  (# inf)
```

Since \mathbb{N}^∞ is defined by "data", one can define deep elimination on \mathbb{N}^∞ . So one can write an infinite element as long as the recursion is after $\#$. However, the definition of ∞ , $\#_- : (x : A) \rightarrow \infty A$ means ∞A is isomorphic to A . Then the definition of \mathbb{N}^∞ is isomorphic to the definition of \mathbb{N} ($\infty (\mathbb{N}^\infty)$ is essentially the same as \mathbb{N}^∞ .) This is not what is intended but that would be the formal interpretation. What is intended is that the definition of \mathbb{N}^∞ is to be understood

as

mutual

data $\mathbb{N}^\infty : \text{Set}$ where

$0^\infty : \mathbb{N}^\infty$

$S^\infty : (\infty\mathbb{N}^\infty) \rightarrow \mathbb{N}^\infty$

codata $\infty\mathbb{N}^\infty : \text{Set}$ where

$\#_- : \mathbb{N}^\infty \rightarrow \infty\mathbb{N}^\infty$

Details have been discussed in [Set10b, Set10a].

4.3 Least and Greatest Fixed Points in Set Theory

In this section we consider algebras and coalgebras in set theory.

The initial algebra and final coalgebras are the least and the largest fixed points if the functor F is monotone. In case of strictly positive F , the least fixed point of F is obtained by iteration of F , i.e. we take in case of \mathbb{N} the union of

$$\emptyset, \underbrace{F(\emptyset)}_{0+S(\emptyset)=\{0\}}, \underbrace{F(F(\emptyset))}_{0+S(\{0\})}, \underbrace{F(F(F(\emptyset)))}_{0+S(\{0,S(0)\})}, F^4(\emptyset), F^5 \dots$$

and obtain $\mathbb{N} = \{0, S(0), S(S(0)), S(S(S(0))), \dots\}$. For strictly positive F the least fixed point of F is obtained by

$$F^* := \bigcup_{\alpha \in \text{Ord}} F^\alpha(\emptyset)$$

The closure principle expresses $F(F^*) \subseteq F^*$, i.e. $0 \in F^*$ and if $a \in F^*$ then $S(a) \in F^*$: $0 \in F^1 \subseteq F^*$ and if $a \in F^*$, then $a \in F^n$ for some n and therefore $S a \in F^{n+1} \subseteq F^*$. It can be seen that F^* is an F -algebra $(F^*, [0, S])$ such that $0 : F^*$ and $S : F^* \rightarrow F^*$

$$1 + F^* \xrightarrow{[0, S]} F^*$$

Furthermore, F^* is the least set closed under 0 and S, so if A is a set containing 0 and closed under S s.t. $F(A) = A$ then $F^* \subseteq A$.

Lemma 4.3.1. *If \mathbb{N} is an initial algebra in the category of sets then if for $A : \text{Set}$ and $0 + S(A) \subseteq A$ then $\mathbb{N} \subseteq A$.*

Proof. Let $0 + S(A) \subseteq A$. We define $g : 0 + S(A) \rightarrow A$ s.t. $g(x) := x$. By \mathbb{N} being an initial algebra there is a unique function $h : \mathbb{N} \rightarrow A$ such that

$$\begin{array}{ccc}
 0 + S(\mathbb{N}) & \xrightarrow{\text{intro}} & \mathbb{N} \\
 \downarrow 0 + S(h) & & \downarrow h \\
 0 + S(A) & \xrightarrow{g} & A
 \end{array}$$

As for the derivation of the iteration principle above, it follows

$$\begin{aligned}
 h(0) &= h(\text{intro}(0)) = g(0) = 0 \\
 h(n+1) &= h(\text{intro}(S(n))) = g(S(h(n))) = S(h(n)) = h(n) + 1
 \end{aligned}$$

and by induction of \mathbb{N} for all $n : \mathbb{N}$, $h(n) = n$. So h is inclusion. □

One can show as well that inductively defined sets are initial algebras. Dually, if we have a coinductive defined set A s.t. $A \subseteq F(A)$ then if $B \subseteq F(B)$ then $B \subseteq A$ such that

$$\begin{array}{ccc}
 B & \xrightarrow{\quad} & F(B) \\
 \text{incl} \downarrow & & \downarrow F(\text{incl}) \\
 A & \xrightarrow{g} & F(A)
 \end{array}$$

where $\text{incl} : B \rightarrow A$ is inclusion. Therefore, A is the greatest fixed point of F .

The construction of final coalgebras in set theory for strictly positive functors is more complicated, it requires the construction of inverse limits, see for instance

the article by Barr [Bar93]. \mathbb{N}^∞ above can be defined in set theory by $\mathbb{N}^\infty = \mathbb{N} \cup \{\omega\}$ together with

$$\begin{aligned} \text{elim} : \mathbb{N}^\infty &\rightarrow 1 + \mathbb{N}^\infty \\ \text{elim } 0 &= \text{inl } * \\ \text{elim } (n + 1) &= \text{inr } n \\ \text{elim } \omega &= \text{inr } \omega \end{aligned}$$

We have shown that the induction and coinduction principles can be derived from initial algebras and final coalgebra as the least and the greatest fixed points of F in the category of sets.

Let us now work on a smaller category namely the category $\mathcal{P}(U)$ of subsets of U . Note that the standard model of type theory interprets sets as subsets of the set of terms, so as an element of $\mathcal{P}(\text{Term})$. Let F be a monotone functor on $\mathcal{P}(U)$ i.e. $F : \mathcal{P}(U) \rightarrow \mathcal{P}(U)$. The least fixed point of F can be defined as follows (folklore, e.g. [Gor94]):

$$\mu.X.F(X) := \bigcap \{X \mid X \in \mathcal{P}(U), F(X) \subseteq X\}$$

$$\nu.X.F(X) := \bigcup \{X \mid X \in \mathcal{P}(U), X \subseteq F(X)\}$$

Then $\mu.X.F(X)$ is the least fixed point of F and $\nu.X.F(X)$ is the largest fixed point of F .

Chapter 5

Cauchy Reals in Agda

In this Chapter we will show how to formalise the real numbers in Agda using postulated axioms. Then we will investigate some properties of real numbers constructed by Cauchy sequences: we will prove that the Cauchy reals (which are the real numbers which are limits of Cauchy sequences of rational numbers) are closed under addition and multiplication and show that the Cauchy reals are Cauchy complete.

5.1 Axioms

We have already discussed (in Chapter 3) how to construct real numbers by Cauchy sequences and we called them Cauchy reals. For carrying out our investigation on some properties of Cauchy reals and computation purposes, we introduce two groups of axioms. In the first group we introduce the set of real numbers \mathbb{R} by axioms and in the second group we introduce the data type of the subset of \mathbb{R} of the rational numbers and what it means for real numbers $r : \mathbb{R}$ to be Cauchy complete.

The basic notions of axiomatizations for real numbers are defined in the language Agda as follows:

- a sort of real numbers \mathbb{R} .
- constants $0, 1$.

- relations : equal ($==$), less ($<$), less or equal (\leq), and apartness ($\#$).
- functions : addition ($+$), multiplication ($*$), negation ($-$), absolute value ($| _ |$), reciprocal (recip).

In the following we formalize our axiomatization in the language of Agda (see below) which can be found in the Agda file `Axioms.agda`. Since we need to use them as axioms and cannot prove them, we postulate them in Agda.

```

ℝ      : Set
r0     : ℝ
r1     : ℝ
_==_   : ℝ → ℝ → Set      {- Equal -}
_<_    : ℝ → ℝ → Set      {- Less -}
_≤_    : ℝ → ℝ → Set      {- Less or equal -}
-      : ℝ → ℝ              {- Negation -}
_#_    : ℝ → ℝ → Set      {- Apartness -}
_+_    : ℝ → ℝ → ℝ         {- Addition -}
*_     : ℝ → ℝ → ℝ         {- Multiplication -}
|_|    : ℝ → ℝ              {- Absolute value -}
recip  : (r : ℝ) → r0 # r → ℝ  {- Reciprocal -}

```

Since we have addition and negation we define subtraction by $r - s = r + (-s)$. Real numbers 2, 3, 4, 5, 6 are defined inductively e.g. $r2 = r1 + r1$, $r3 = r2 + r1$, $r4 = r3 + r1$, \dots , $r6 = r5 + r1$. The negative real numbers $-1, -2, -3, -5, -6$ are defined as negated positive real numbers e.g. $-r1, -r2, -r3, \dots$ etc.

```

{- Axioms for 0, 1 -}
0<1      : r0 < r1

{- Axioms for Negation -}
-0       : - r0 == r0
--x=x    : (r : ℝ) → -(-r) == r
-R<0     : (r : ℝ) → r0 < r → -r < r0

```

{- Axioms for == -}

$\text{refl} ==$: $(r : \mathbb{R}) \rightarrow r == r$
 $\text{symm} ==$: $(r s : \mathbb{R}) \rightarrow r == s \rightarrow s == r$
 $\text{trans} ==$: $(r s t : \mathbb{R}) \rightarrow r == s \rightarrow s == t \rightarrow r == t$

{- Axioms for |·| -}

abs0 : $|^r 0| ==^r 0$
 $0 \leq |x|$: $(r : \mathbb{R}) \rightarrow {}^r 0 \leq |r|$
 $|-x| = |x|$: $(r : \mathbb{R}) \rightarrow |-r| == |r|$
 $|xy| = |x||y|$: $(r s : \mathbb{R}) \rightarrow |r * s| == |r| * |s|$
 $r \in [-n, n] \rightarrow |r| \leq n$: $(r s : \mathbb{R}) \rightarrow -s \leq r \wedge r \leq s \rightarrow |r| \leq s$

{- Axioms for Apartness -}

$\# \text{less}$: $(r s : \mathbb{R}) \rightarrow r < s \rightarrow r \# s$
 $\# \text{symm}$: $(r s : \mathbb{R}) \rightarrow r \# s \rightarrow s \# r$
 $0 \# sr$: $(r s : \mathbb{R}) \rightarrow {}^r 0 \# r \rightarrow {}^r 0 \# s \rightarrow {}^r 0 \# r * s$

{- Axioms for <, ≤ -}

$\text{trans} \leq$: $(r s t : \mathbb{R}) \rightarrow r \leq s \rightarrow s \leq t \rightarrow r \leq t$
 $a \leq b <$: $(r s : \mathbb{R}) \rightarrow r < s \rightarrow r \leq s$
 $a \leq b =$: $(r s : \mathbb{R}) \rightarrow r == s \rightarrow r \leq s$
 $a \leq b \rightarrow b < c \rightarrow a < c$: $(r s t : \mathbb{R}) \rightarrow r \leq s \rightarrow s < t \rightarrow r < t$
 $a < b \rightarrow b \leq c \rightarrow a < c$: $(r s t : \mathbb{R}) \rightarrow r < s \rightarrow s \leq t \rightarrow r < t$

{- Axioms for + -}

$\text{axiom} + 0$: $(r : \mathbb{R}) \rightarrow r + {}^r 0 == r$
 $\text{symm} +$: $(r s : \mathbb{R}) \rightarrow r + s == s + r$
 $\text{assoc} +$: $(r s t : \mathbb{R}) \rightarrow r + (s + t) == (r + s) + t$
 $\text{minus} +$: $(r s : \mathbb{R}) \rightarrow -(r + s) == (-r) + (-s)$
 $\text{axiom} x - x$: $(r : \mathbb{R}) \rightarrow r + (-r) == {}^r 0$

{- Axioms for * -}

axiom*0	: (r : ℝ) → r * r0 == r0
axiom*1	: (r : ℝ) → r * r1 == r
a*-b=-ab	: (r s : ℝ) → r * (-s) == -(r * s)
symm*	: (r s : ℝ) → r * s == s * r
assoc*	: (r s t : ℝ) → r * (s * t) == (r * s) * t
distri*	: (r s t : ℝ) → r * (s + t) == r * s + r * t

{- Axioms for recip -}

0<recip	: (r : ℝ) → r0 < r → (p : r0 # r) → r0 < recip r p
recip<0	: (r : ℝ) → r < r0 → (p : r0 # r) → recip r p < r0
aa ⁻¹ =1	: (r : ℝ) → (p : r0 # r) → (r * (recip r p)) == r1
x<y→1/y<1/x	: (r s : ℝ) → (p : r0 # r) → (p' : r0 # s) → r < s → (recip s p') < (recip r p)
recipxy=recipx*recipy	: (r s : ℝ) → (prs : r0 # r * s) → (p : r0 # r) → (p' : r0 # s) → recip (r * s) prs == recip r p * recip s p'
recip==	: (r s : ℝ) → (p : r0 # r) → (p' : r0 # s) → r == s → recip r p == recip s p'

Note that: if we had $\forall x \in \mathbb{R}. x > 0 \vee x = 0 \vee x < 0$ then the axiom $aa^{-1}=1$ would imply $0<\text{recip}$, $\text{recip}<0$, $x<y\rightarrow 1/y<1/x$. Axiom $\text{recip}==$ is provable, as follows: if $a = r$, $a' = s$, $r = s$, $b = \text{recip } r p$, $b' = \text{recip } s p'$, we get by $aa^{-1}=1$, $a * b = 1$, $a' * b' = 1$, therefore

$$b = b * 1 = b * (a' * b') = b * (a * b') = (b * a) * b' = 1 * b' = b'$$

Similarly axiom $\text{recip}xy=\text{recip}x*\text{recip}y$ is provable.

The reciprocal is only defined for real numbers which are not 0. In order to define the reciprocal of real numbers we need the apartness relation. The apartness relation denotes that two real numbers are strictly not equal which

means of two real numbers either one is larger or less than another. So the intended meaning of $r \# s$ is $(r < s) \vee (s < r)$ although we only axiomatize $(r < s \vee s < r) \rightarrow r \# s$ since the other direction would be an axiom which has as conclusion an algebraic data type (see Chapter 8). Axiom $\neg(r \# r)$ is missing (it is not provable since we could model \mathbb{R} by the one element set: constants, functions return the only element of \mathbb{R} and $==, <, \leq$ as always true). We don't add it since it has the form $r \# r \rightarrow \perp$ which has as conclusion an algebraic data type (see below). So $r \# s$ doesn't imply $\neg(r == s)$. $\neg(r == s)$ doesn't imply $r \# s$, we might know $\neg(r == s)$ but neither $r < s$ nor $s < r$. With axiom $\neg(r \# r)$ we had that $r \# s \rightarrow \neg(r == s)$ (since if $r \# s$ and $r == s$ we get $r \# r$, a contradiction). So with this axiom, $r \# s$ is stronger than $\neg(r == s)$.

We need an axiom or lemma stating that $\text{recip } r \ p == \text{recip } r \ p'$ for $(p \ p' : \text{r}0 \# r)$: there could be different proofs $(p \ p' : \text{r}0 \# r)$ and therefore $\text{recip } r \ p$ and $\text{recip } r \ p'$ would not be definitionally equal but we expect $\text{recip } r \ p == \text{recip } r \ p'$ to be equal (see axiom `recip ==`).

{- Axioms for $<, \leq$ with $+$ -}

`axiom<+` : $(r \ s \ t : \mathbb{R}) \rightarrow r < s \rightarrow r + t < s + t$

`axiom $\leq+$` : $(r \ s \ t : \mathbb{R}) \rightarrow r \leq s \rightarrow r + t \leq s + t$

{- Axioms for $<, \leq$ with $*$ -}

`axiom<*` : $(r \ s \ t : \mathbb{R}) \rightarrow r < s \rightarrow \text{r}0 < t \rightarrow r * t < s * t$

`c \leq 0 \rightarrow a \leq b \rightarrow ac \leq bc`: $(r \ s \ t : \mathbb{R}) \rightarrow \text{r}0 \leq t \rightarrow r \leq s \rightarrow r * t \leq s * t$

{- Transfer -}

`transfer==` : $(P : \mathbb{R} \rightarrow \text{Set}) \rightarrow (r \ s : \mathbb{R}) \rightarrow r == s \rightarrow P \ r \rightarrow P \ s$

{- Triangle inequality -}

`triangle` : $(r \ s \ t : \mathbb{R}) \rightarrow |r - s| \leq |r - t| + |t - s|$

Note that `transfer==` violates our conditions (see Chapter 8). We will use it

only if $(P\ s)$ is an equality on postulate type. Then it fulfils our condition. More precisely we could define for each allowed instance used one constant which has as result type equality or postulate data type. Since this can easily be done, it is okay to introduce and use this constant axiom `transfer==` using these restrictions.

Another group of axioms and definitions consists of limitpoint and convergence axioms and the introduction of the data type $\mathbb{Q} \subseteq \mathbb{R}$ to be the set of real numbers which are rational numbers. Note that we have defined \mathbb{Q} before. \mathbb{Q} was introduced for computational reasons. We could define an isomorphism between \mathbb{Q} and $\{r : \mathbb{R} \mid \mathbb{Q}(r)\}$, provided we have the axiom $\neg (r0 \# r0)$

```

{- data type Q -}
data Q      : ℝ → Set  where
  close0    : Q r0
  close1    : Q r1
  close-    : (r : ℝ) → Q r → Q (-r)
  close+    : (r s : ℝ) → Q r → Q s → Q (r + s)
  close*    : (r s : ℝ) → Q r → Q s → Q (r * s)
  closerecip : (r : ℝ) → (p : r0 # r) → Q r
              → Q (recip r p)

```

Note that we don't extract programs in this part. Therefore, it is not a problem that \mathbb{Q} is not a restricted indexed inductive definition, see Chapter 8. If one would like to work with program extraction one could use the following alternative definition of \mathbb{Q}

```

embedN→ℝ : ℕ → ℝ
embedN→ℝ n = ...

```

Instead of showing full code we write ... which can be found in our Agda code.

```
embedN+→ℝ : ℕ+ → ℝ
embedN+→ℝ n = ...
```

```
embedℚ→ℝ : ℚ → ℝ
embedℚ→ℝ (x %' y) = embedℤ→ℝ x * recip (embedN+→ℝ y) (#less ...)
```

and define

$$\mathbb{Q}r := (q : \mathbb{Q}) \times (\text{embed}\mathbb{Q}\rightarrow\mathbb{R} q == r)$$

and work with this data type. We discovered this problem at a late stage and leave it for future work.

The limitpoint function states that each Cauchy sequence has a limit in \mathbb{R} . The convergence function axiomatises that all Cauchy sequences converge in \mathbb{R} . Both functions together can be regarded as the completeness axiom which states that \mathbb{R} is Cauchy complete. They are defined as follows

```
{- Exponential functions of ℝ -}
exp : (r : ℝ) → (r0 # r) → ℤ → ℝ
exp x p (pos (0 +1)) = x
exp x p (pos (suc n +1)) = exp x p ((pos (n +1)) * x
exp x p (neg (0 +1)) = recip x p
exp x p (neg (suc n +1)) = exp x p ((neg (n +1)) *
recip x p
exp x p zero = r1
```

```
2^ : ℤ → ℝ    {- 2^ n = 2^n -}
2^ n = exp r2 p n - - where p : r0#r2
```

```

limitpoint  : (f : ℕ+ → ℝ)
              → (p : (n m N : ℕ+)
                  → n ≥+ N
                  → m ≥+ N
                  → | f n - f m | < 2^ (neg N))
              → ℝ

```

```

convergence : (f : ℕ+ → ℝ)
              → (p : (n m N : ℕ+)
                  → n ≥+ N
                  → m ≥+ N
                  → | f n - f m | < 2^ (neg N))
              → (k : ℕ+)
              → | f k - limitpoint f p | ≤ 2^ (neg k)

```

Function `limitpoint` determines the real number which is the limit of a Cauchy sequence. Function `convergence` states that the Cauchy sequence converges to this limit point.

We introduce two more data types: the set of real numbers Q' , which are limits of elements in Q (these are the Cauchy reals), and the set of real numbers Q'' which are limits of elements in Q' . They are defined as follows:

```

data Q' (r : ℝ) : Set where
  Q'intro  : (f : ℕ+ → ℝ)
              → (Pf : (n : ℕ+) → Q (f n))
              → (p : (n : ℕ+)
                  → (| x - f n |) < 2^ (neg n))
              → Q' x

```

```

data Q'' (r : ℝ) : Set where
  Q''intro : (f : ℕ+ → ℝ)
            → (Pf : (n : ℕ+) → Q' (f n))
            → (p : (n : ℕ+)
              → (| x - f n |) < 2^- (neg n))
            → Q'' x

```

In the following, we will show that Q' is closed under $+$, $*$ and Q'' is also a subset of Q' , which means that Q' is Cauchy complete since Q'' is the set of limits of Cauchy sequences in Q' which are in \mathbb{R} , and \mathbb{R} is Cauchy complete. So every Cauchy sequence in Q' has a limit in Q' .

5.2 Q' Closure Under Addition

Let $(a_n)_{n \in \mathbb{N}}$, $(b_n)_{n \in \mathbb{N}}$ be two sequences of rational numbers converging to r , s . Then $(a_n + b_n)_{n \in \mathbb{N}}$ converges to $r + s$: This is shown as follows: Since $(a_n)_{n \in \mathbb{N}}$ converges to r we have $\forall \epsilon > 0. \exists N_1 \in \mathbb{N}. \forall n \in \mathbb{N}. n \geq N_1 \rightarrow |a_n - r| < \epsilon/2$. Since $(b_n)_{n \in \mathbb{N}}$ converges to s , we have $\forall \epsilon > 0. \exists N_2 \in \mathbb{N}. \forall n \in \mathbb{N}. n \geq N_2 \rightarrow |b_n - s| < \epsilon/2$. Let $\epsilon > 0$, N_1, N_2 are chosen as before, $N = \max\{N_1, N_2\}$. Then for $n \geq N$ we have $|(a_n + b_n) - (r + s)| \leq |a_n - r| + |b_n - s| < \epsilon/2 + \epsilon/2 = \epsilon$. So $((a_n)_{n \in \mathbb{N}}) + ((b_n)_{n \in \mathbb{N}})$ converges to $r + s$. In our formulation of Q' converging sequences have a fixed convergence rate, so instead of

$$\forall \epsilon > 0. \exists N : \mathbb{N}. \forall n, m \geq N. |a_n - a_m| < \epsilon$$

we have

$$\forall n, m \geq N. |a_n - a_m| < 2^{-N}$$

Therefore, we need to switch from $(a_n + b_n)_{n \in \mathbb{N}}$ to $(a_{n+1} + b_{n+1})_{n \in \mathbb{N}}$ in order to obtain a sequence which is in accordance with our fixed convergence rate. The proof in Agda is as follows (see the Agda file `CauchyRealsAddition.agda`):

$$\begin{aligned}
 |a + b| < 2^{-n} & : (r \ s \ r' \ s' : \mathbb{R}) \rightarrow (n : \mathbb{N}^+) \\
 & \rightarrow |r - r'| < 2^{\wedge(\text{neg } (n + 1))} \\
 & \rightarrow |s - s'| < 2^{\wedge(\text{neg } (n + 1))} \\
 & \rightarrow |(r + s) - (r' + s')| < 2^{\wedge(\text{neg } n)} \\
 |a + b| < 2^{-n} \ r \ s \ r' \ s' \ n \ p \ p' & = \dots
 \end{aligned}$$

$$\begin{aligned}
 \text{Q'close+} : (r \ s : \mathbb{R}) & \rightarrow \text{Q' } r \rightarrow \text{Q' } s \rightarrow \text{Q' } (r + s) \\
 \text{Q'close+ } r \ s \ (\text{Q'intro } f \ P f \ p) \ (\text{Q'intro } f' \ P f' \ p') & = \\
 \text{Q'intro } \{r + s\} & \\
 (\backslash n \rightarrow f \ (n \ +^+ \ +1) + f' \ (n \ +^+ \ +1)) & \\
 (\backslash n \rightarrow \text{close+ } (f \ (n \ +^+ \ +1)) \ (f' \ (n \ +^+ \ +1))) & \\
 (P f \ (n \ +^+ \ +1)) \ (P f' \ (n \ +^+ \ +1)) & \\
 (\backslash n \rightarrow |a + b| < 2^{-n} \ r \ s \ (f \ (n \ +^+ \ +1)) \ (f' \ (n \ +^+ \ +1)) \ n & \\
 (p \ (n \ +^+ \ +1)) \ (p' \ (n \ +^+ \ +1))) &
 \end{aligned}$$

The proof shows us that the sequence $(f \ (n \ +^+ \ +1) + f' \ (n \ +^+ \ +1))$ converges to $(r + s)$. `close+` ... proves that the sequence is in \mathbb{Q} and `|a + b| < 2-n` ... shows that $| (f \ (r + s)) - (f \ (n \ +^+ \ +1) + f' \ (n \ +^+ \ +1)) | < 2^{-n}$. So it converges to $(r + s)$.

5.3 Q' Closure Under Multiplication

Now we show that \mathbb{Q}' is closed under multiplication. If $((a_n)_{n \in \mathbb{N}})$ converges to a real number r and $((b_n)_{n \in \mathbb{N}})$ converges to a real number s then $((a_n)_{n \in \mathbb{N}}) * ((b_n)_{n \in \mathbb{N}})$ converges to $(r * s)$ for two Cauchy sequences of rational numbers $(a_n)_{n \in \mathbb{N}}$ and $(b_n)_{n \in \mathbb{N}}$. First we use the fact that \mathbb{Q} is Archimedean therefore all Cauchy sequences in \mathbb{Q} are bounded, so there are $K_1, K_2 \in \mathbb{N} > 1$ such that for all $n \in \mathbb{N}$, $|a_n| \leq K_1$ and $|s| \leq K_2$. The existence of K_1 follows in case of a fixed convergence rate because $|a_n - a_1| < 2^{-1}$ for $n \geq 1$ and by the Archimedean axiom $|a_1| \leq b$ for some $b \in \mathbb{N}$ therefore for $K_1 := b + 1$ we have

$$|a_n| \leq |a_n - a_1| + |a_1| \leq 2^{-1} + b \leq b + 1 = K_1$$

Without a fixed convergence rate we have an N s.t. $\forall n \geq N. |a_n - a_N| < 1$. We have by the Archimedean axiom $|a_N| \leq L$ for some $L \in \mathbb{N}$ and $|a_i| \leq L_i$ for $i < N$. Then for all $n \in \mathbb{N} |a_n| \leq K_1 := \max\{L_1, \dots, L_{n+1}, L + 1\}$; for $i < N |a_i| \leq L_i \leq K_1$, for $i \geq N |a_i| \leq |a_i - a_N| + |a_N| < 1 + L \leq K_1$.

Let $(a_n)_{n \in \mathbb{N}}$ converge to r , therefore $\forall \epsilon > 0. \exists N_1 \in \mathbb{N}. \forall n \in \mathbb{N}. n \geq N_1 \rightarrow |a_n - r| < \epsilon / (2K_2)$. $(b_n)_{n \in \mathbb{N}}$ convergest to s , therefore $\forall \epsilon > 0. \exists N_2 \in \mathbb{N}. \forall n \in \mathbb{N}. n \geq N_2 \rightarrow |b_n - s| < \epsilon / (2K_1)$. Let $N = \max\{N_1, N_2\}$. Then for $n \geq N$ we have $|a_n b_n - r s| = |a_n(b_n - s) + s(a_n - r)| \leq |a_n| |b_n - s| + |s| |a_n - r| < K_1(\epsilon / (2K_1)) + K_2(\epsilon / 2K_2) = \epsilon$.

The proof in Agda is as follows (again the proof needs to be adapted since we are dealing with sequences with fixed convergence rate; see the Agda file `CauchyRealsMultiplication.agda`)

The proof will show us that the sequence $(f (n \text{ } +^+ \text{ } m) * f' (n \text{ } +^+ \text{ } m))$ converges to $(r * s)$. The function `Q'close*` ... will prove that the sequence is in \mathbb{Q} and `Q'r*s<2-n` ... show that $|r * s - f (n \text{ } +^+ \text{ } m) * f' (n \text{ } +^+ \text{ } m)| < 2^{-n}$. Therefore, it will converge to $(r * s)$.

```

module closeQ* (r s : ℝ)
  (f : (n : ℕ+) → ℝ)
  (Pf : (n : ℕ+) → ℚ (f n))
  (p : (n : ℕ+) → (| (r - f n) |) < 2^(neg n))
  (f' : (n : ℕ+) → ℝ)
  (Pf' : (n : ℕ+) → ℚ (f' n))
  (p' : (n : ℕ+) → (| (s - f' n) |) < 2^(neg n))
  (k l : ℕ)
  (f1≤k : | f +1 | ≤ embedℕ→ℝ k)
  (f'1≤l : | f' +1 | ≤ embedℕ→ℝ l)
  (m : ℕ+)
  (1+l+2+k≤2^m : 1 + (embedℕ→ℝ l)
    +r2 + (embedℕ→ℝ k) ≤ 2^ (pos m))

```

where

$$\begin{aligned} Q'r*s < 2^{-n} & : (n : \mathbb{N}^+) \\ & \rightarrow |r * s - f(n +^+ m) * f'(n +^+ m)| < 2^{\wedge} (\text{neg } n) \end{aligned}$$

$$Q'r*s < 2^{-n} \ n = \dots$$

$$\begin{aligned} Q'r*s & : Q'(r * s) \\ Q'r*s & = (Q'\text{intro } \{r * s\}) \\ & (\backslash n \rightarrow f(n +^+ m) * f'(n +^+ m)) \\ & (\backslash n \rightarrow \text{close} * (f(n +^+ m)) (f'(n +^+ m))) \\ & (pf(n +^+ m)) (pf'(n +^+ m))) \\ & (\backslash n \rightarrow Q'r*s < 2^{-n} \ n) \end{aligned}$$

where the proof $Q'r*s < 2^{-n} \ n = \dots$ is obtained from

$$\begin{aligned} & |r * s - f(n +^+ m) * f'(n +^+ m)| = \\ & |s * (r - f(n +^+ m)) + f(n +^+ m) * (s - f'(n +^+ m))| \leq \\ & |s * (r - f(n +^+ m))| + |f(n +^+ m) * (s - f'(n +^+ m))| = \\ & \underbrace{|s|}_{\leq 1=l} * \underbrace{|(r - f(n +^+ m))|}_{< 2^{-(n+m)}} + \underbrace{|f(n +^+ m)|}_{< 2+k} * \underbrace{|(s - f'(n +^+ m))|}_{< 2^{-(n+m)}} < 2^{\wedge} (\text{neg } n) \\ & \underbrace{ * + * }_{< 2^{-(n+m)} * \underbrace{(1 + l + 2 + k)}_{\leq 2^m} = 2^{-n}} \end{aligned}$$

where $|s| \leq 1 = l$ follows from

$$|s| \leq |s - f' 1| + |f' 1| \leq 2^0 + 1 = 1 + l$$

$|f(n + m)| < 2 + k$ follows from

$$|f(n + m)| \leq |f(n + m) - r| + |r| \leq 2^{-(n+m)} + k \leq 1 + k < 2 + k$$

$$Q \rightarrow Q : (r : \mathbb{R}) \rightarrow Q \ r \rightarrow Q$$

$Q \rightarrow Q \ r \ p = \{ \}$ — left for future work, see below

$$|\mathbb{Z}| \rightarrow \mathbb{N} : \mathbb{Z} \rightarrow \mathbb{N}$$

$$|\mathbb{Z}| \rightarrow \mathbb{N} ((\text{pos } (y + 1))) = \text{suc } y$$

$$|\mathbb{Z}| \rightarrow \mathbb{N} ((\text{neg } (y + 1))) = \text{suc } y$$

$|\mathbb{Z}| \rightarrow \mathbb{N} \hat{\text{zero}} = 0$

$\text{archimedesaux} : (q : \mathbb{Q}) \rightarrow \mathbb{N}$
 $\text{archimedesaux } (y \%' y') = |\mathbb{Z}| \rightarrow \mathbb{N} y$

$\text{archimedes1} : (r : \mathbb{R}) \rightarrow \mathbb{Q} r \rightarrow \mathbb{N}$
 $\text{archimedes1 } r \text{ } Qr = \text{archimedesaux } (\mathbb{Q} \rightarrow \mathbb{Q} r \text{ } Qr)$

$\text{archimedes2} : (r : \mathbb{R}) \rightarrow (p : \mathbb{Q} r)$
 $\rightarrow | \text{embed } \mathbb{Q} \rightarrow \mathbb{R} (\mathbb{Q} \rightarrow \mathbb{Q} r \text{ } p) | \leq \text{embed } \mathbb{N} \rightarrow \mathbb{R} (\text{archimedes1 } r \text{ } p)$
 $\text{archimedes2 } r \text{ } p = \dots$

where the $\text{embed } \mathbb{Q} \rightarrow \mathbb{R}$ function is left for future work. We could give an $\text{embed } \mathbb{Q} \rightarrow \mathbb{R}$ function if we added the axiom $\neg(r0 \# r0)$ as we have mentioned in page:63.

Now we give the proof that \mathbb{Q}' is closed under $*$. It mainly uses the proof $\text{Q}'r*s$ of closeQ* which shows that $(f(n+1) * g(n+1))_{n \in \mathbb{N}^+}$ converge to $r + s$, if $(f n)_{n \in \mathbb{N}^+}$ converges to r and $(g n)_{n \in \mathbb{N}^+}$ converges to s .

open closeQ*

$\text{Q}'\text{close*} : (r s : \mathbb{R}) \rightarrow \mathbb{Q}' r \rightarrow \mathbb{Q}' s \rightarrow \mathbb{Q}' (r * s)$
 $\text{Q}'\text{close* } r \text{ } s (\text{Q}'\text{intro } f \text{ } Pf \text{ } p) (\text{Q}'\text{intro } f' \text{ } Pf' \text{ } p') = \text{closeQ*}.\text{Q}'r*$
 $r \text{ } s \text{ } f \text{ } Pf \text{ } p \text{ } r \text{ } f' \text{ } Pf' \text{ } p' \text{ } k \text{ } l \text{ } f1 \leq k \text{ } f'1 \leq l \text{ } m \text{ } 1+l+2+k \leq 2^{\wedge} m$

where

$k : \mathbb{N}$
 $k = \text{archimedes1 } (f \text{ } +1) (Pf \text{ } +1)$

$l : \mathbb{N}$
 $l = \text{archimedes1 } (f' \text{ } +1) (Pf' \text{ } +1)$

$f1 \leq k : (| f \text{ } +1 |$

$$\leq \text{embed } \mathbb{N} \rightarrow \mathbb{R} \text{ (archimedesaux (Q} \rightarrow \mathbb{Q} \text{ (f } +1) \text{ (Pf } +1)))}$$

$$f'1 \leq k = \dots$$

$$f'1 \leq l : (| f' +1 |$$

$$\leq \text{embed } \mathbb{N} \rightarrow \mathbb{R} \text{ (archimedesaux (Q} \rightarrow \mathbb{Q} \text{ (f' } +1) \text{ (Pf' } +1)))}$$

$$f'1 \leq l = \dots$$

$$2^{\wedge} \mathbb{N} \rightarrow \mathbb{N}^+ : \mathbb{N} \rightarrow \mathbb{N}^+$$

$$2^{\wedge} \mathbb{N} \rightarrow \mathbb{N}^+ n = \dots$$

$$m : \mathbb{N}^+$$

$$m = 2^{\wedge} \mathbb{N} \rightarrow \mathbb{N}^+ l +^{+} +1 +^{+} 2^{\wedge} \mathbb{N} \rightarrow \mathbb{N}^+ k +^{+} +2$$

$$1+l+2+k \leq 2^{\wedge} m : \text{ }^r 1 + (\text{embed } \mathbb{N} \rightarrow \mathbb{R} l)$$

$$+^r 2 + (\text{embed } \mathbb{N} \rightarrow \mathbb{R} k) \leq 2^{\wedge} (\text{pos } m)$$

$$1+l+2+k \leq 2^{\wedge} m = \dots$$

5.4 Q' is Cauchy-complete

We have defined $\{r : \mathbb{R} \mid \mathbb{Q} r\}$ to be the real numbers which are rational numbers and $\{r : \mathbb{R} \mid \mathbb{Q}' r\}$ to be the real numbers which are limits of Cauchy sequences of rational numbers. So we get $\mathbb{Q} \subseteq \mathbb{Q}'$. Then \mathbb{Q}'' are the real numbers which are limits of Cauchy sequences of elements in \mathbb{Q}' . A limit of a Cauchy sequence in \mathbb{Q} is a limit of a Cauchy sequence in \mathbb{Q}' , therefore $\mathbb{Q} \subseteq \mathbb{Q}' \subseteq \mathbb{Q}''$. We show $\mathbb{Q}'' \subseteq \mathbb{Q}'$.

First we prove that $\mathbb{Q} \subseteq \mathbb{Q}'$ and $\mathbb{Q}' \subset \mathbb{Q}''$. This is essentially done by taking as sequence to converge to a the sequence $(a_n)_{n \in \mathbb{N}^+}$. The proof that these sequences are in \mathbb{Q} , \mathbb{Q}' respectively is trivial (a is in \mathbb{Q} or \mathbb{Q}') and that they converge is trivial since the sequence elements are identical to the limit.

The proof in Agda is as follows (see the Agda file `CauchyRealsCompleteness.agda`)

$$|x-x| < 2^{\wedge} : (r : \mathbb{R}) \rightarrow (n : \mathbb{N}^+) \rightarrow |r - r| < 2^{\wedge} (\text{neg } n)$$

$$|x-x| < 2^{\wedge} x n = \dots$$

$$\begin{aligned} Q \rightarrow Q' &: (r : \mathbb{R}) \rightarrow Q \ r \rightarrow Q' \ r \\ Q \rightarrow Q' \ x \ qx &= (Q' \text{intro } \{x\}) (\backslash_ \rightarrow x) (\backslash_ \rightarrow qx) (\backslash n \rightarrow |x-x| < 2^{-n} \ x \ n) \end{aligned}$$

$$\begin{aligned} Q' \rightarrow Q'' &: (r : \mathbb{R}) \rightarrow Q' \ r \rightarrow Q'' \ r \\ Q' \rightarrow Q'' \ x \ q'x &= Q'' \text{intro } (\backslash n \rightarrow x) (\backslash n \rightarrow q'x) (\backslash n \rightarrow |x-x| < 2^{-n} \ x \ n) \end{aligned}$$

$$\begin{aligned} Q \rightarrow Q'' &: (r : \mathbb{R}) \rightarrow Q \ r \rightarrow Q'' \ r \\ Q' \rightarrow Q'' \ x \ qx &= Q' \rightarrow Q'' \ x \ (Q \rightarrow Q' \ x \ qx) \end{aligned}$$

where $Q \rightarrow Q'$ shows that $Q \subseteq Q'$. $Q' \rightarrow Q''$ has shown us that $Q' \subseteq Q''$. Therefore, $Q \subseteq Q' \subseteq Q''$ by $Q \rightarrow Q''$.

In order to prove $Q'' \subseteq Q$, we first extract from proof of $p : Q' \ r$ its Cauchy sequence $(Q' \text{tof } r \ p)$, a proof $(Q' \text{toPf } r \ p)$ that it is in Q and a proof $(Q' \text{toPf } r \ p)$ that the Cauchy sequence converges .

$$\begin{aligned} Q' \text{tof} &: (r : \mathbb{R}) \rightarrow Q' \ r \rightarrow \mathbb{N}^+ \rightarrow \mathbb{R} \\ Q' \text{tof } x \ (Q' \text{intro } f \ Pf \ p) \ n &= f \ n \end{aligned}$$

$$\begin{aligned} Q' \text{toPf} &: (r : \mathbb{R}) \rightarrow (qr : Q' \ r) \rightarrow (n : \mathbb{N}^+) \rightarrow Q \ (Q' \text{tof } qr \ n) \\ Q' \text{toPf } x \ (Q' \text{intro } f \ Pf \ p) \ n &= Pf \ n \end{aligned}$$

$$\begin{aligned} Q' \text{top} &: (r : \mathbb{R}) \rightarrow (qr : Q' \ r) \rightarrow (n : \mathbb{N}^+) \\ &\rightarrow (| (r - Q' \text{tof } r \ qr \ n) |) < (2^{-n} \ (\text{neg } n)) \\ Q' \text{top } x \ (Q' \text{intro } f \ Pf \ p) \ n &= p \ n \end{aligned}$$

Now we show $Q'' \subseteq Q$. Let $r : \mathbb{R}$, $p : Q'' \ r$ and the Cauchy sequence contained in p be $(g \ n)_{n \in \mathbb{N}^+}$, $(g \ n = Q' \text{tof } r \ p \ n)$. $g \ n \in Q'$ so there is a Cauchy sequence $(Q' \text{tof } (g \ n) \ \dots)_{n \in \mathbb{N}^+}$ which converges to r . Let the n th element of the Cauchy sequence be $f \ n = Q' \text{tof } (g \ n) \ n$. We show $(f \ (n + 1))_{n \in \mathbb{N}^+}$ converges to r :

$$| f \ (n + 1) - r | \leq | f \ (n + 1) - g \ (n + 1) | + | g \ (n + 1) - r | < 2^{-(n+1)} + 2^{-(n+1)} = 2^{-n}$$

(This argument will be given in $Q'' \rightarrow Q' \text{aux5}$ below). The formal proof in Agda is as follows:

– $Q'' \rightarrow Q'$ aux1 $r p$ is the sequence $(f (n + 1))_{n \in \mathbb{N}^+}$ just given

$Q'' \rightarrow Q'$ aux1 : $(r : \mathbb{R}) \rightarrow Q'' r \rightarrow \mathbb{N}^+ \rightarrow \mathbb{R}$

$Q'' \rightarrow Q'$ aux1 $x (Q'' \text{intro } f Pf p) n = Q' \text{tof } (f (n + + + 1))$
 $(Pf (n + + + 1)) (n + + + 1)$

– –proof that the sequence is in Q

$Q'' \rightarrow Q'$ aux2 : $(r : \mathbb{R}) \rightarrow (qr : Q'' r) \rightarrow (n : \mathbb{N}^+) \rightarrow Q (Q'' \rightarrow Q' \text{aux1 } r qr n)$

$Q'' \rightarrow Q'$ aux2 $x (Q'' \text{intro } f Pf p) n = Q' \text{toPf } (f (n + + + 1))$
 $(Pf (n + + + 1)) (n + + + 1)$

– –proof that the sequence converges

$Q'' \rightarrow Q'$ aux5 : $(r : \mathbb{R}) \rightarrow (qr : Q'' r) \rightarrow (n : \mathbb{N}^+)$

$\rightarrow (| (r - Q'' \rightarrow Q' \text{aux1 } r qr n) |) < (2^{\wedge} (\text{neg } n))$

$Q'' \rightarrow Q'$ aux5 $x qx n = \dots$

– –Proof of $Q'' \rightarrow Q'$

$Q'' \rightarrow Q' : (r : \mathbb{R}) \rightarrow Q'' r \rightarrow Q' r$

$Q'' \rightarrow Q' x qx = Q' \text{intro } \{x\} (Q'' \rightarrow Q' \text{aux1 } x qx) (Q'' \rightarrow Q' \text{aux2 } x qx)$
 $(Q'' \rightarrow Q' \text{aux5 } x qx)$

$Q \rightarrow Q'$ shows that $Q'' \subseteq Q'$. By $Q' \subseteq Q''$ and $Q'' \subseteq Q'$ we get that Q'' is equal to Q' . Since the limits of Cauchy sequences of elements of Q' are in Q' itself, Q' is Cauchy complete. Therefore, every Cauchy sequences in Q' has a limit in Q' , Q' is Cauchy complete.

Chapter 6

Signed Digit Representation of Real Numbers in Classical Mathematics

In this chapter we work in classical set theory. In the previous Chapter 2.2 we mentioned that not all real numbers have a decimal representation. We cannot prove constructively that every Cauchy sequence converges to a number which has a decimal representation. Here is an example, considering a Cauchy real $(a_n)_{n \in \mathbb{N}}$ starting with :

$$a_0 = 0$$

$$a_1 = 0.9$$

$$a_2 = 0.99$$

$$a_3 = 0.999$$

$$a_4 = 0.9999$$

$$a_5 = 0.99999$$

$$a_6 = 0.999998$$

and assume $|a_n - a_k| < 2 * 10^{-n}$ for $k \geq n$. If $a_6 = 0.999998$, then we know the first two digits should have been 0.9. However, if $a_6 = 1.000001$ then the first two digits would have been 1.0. What does this tell us? As long as we have a sequence containing $0.99 \cdots 9$ above, we cannot decide the first digit of the limit.

The underlying reason for this is that for $r, s \in \mathbb{R}$ s.t. $r < s \vee r \geq s$ we cannot

decide whether $r < s$ or $r \geq s$. This is because if r is approximated by something which is equal to s then we cannot decide whether eventually $r < s$ or eventually $s \leq r$. r might eventually become equal to s or larger than s or less than s . It is undecidable where r will be (this applies to $r \leq s \vee r \geq s$ as well). Nevertheless, for any $\epsilon > 0$ we can decide $r < s + \epsilon \vee r \geq s - \epsilon$ (just approximate r and s up to $\epsilon/3$).

For a real number r we know if its decimal representation starts with 1.0 then $r \geq 1.0$. If it starts with $0.9 \cdots 9d$ where $d \in \{0, 1, \dots, 8\}$ then $r < 1.0$. Since we cannot decide $r \geq 1.0$ or $r < 1.0$, for a real number we cannot determinate its decimal representation. The same applies to binary representation.

This chapter is based on the work of Berger, Seisenberger and Hou [BS10b, BS10a, BH08] but the representation has changed.

6.1 Signed Digit Representation

The idea to overcome this is by introducing binary representation of real numbers $0, 1$ with one extra digit -1 .

A sequence of such digits is called a signed digit stream i.e.

$$s = \{(d_0, d_1, d_2, \dots) \mid \forall n \in \mathbb{N}. d_n \in \text{Digit}\} \in \text{Stream}$$

We write (d_0, d_1, d_2, \dots) for the stream of digits $d_0 : d_1 : d_2 : \dots$ and if $s = (d_0, d_1, d_2, \dots)$ then $d : s = (d, d_0, d_1, d_2, \dots)$.

With the signed digits $1, 0, -1$ for $r \in \mathbb{R}$ and $r \in [-1, 1]$, if we know that

$$\begin{cases} 0 \leq r \leq 1, & \text{then we can set the first digit to } 1, \\ -1/2 \leq r \leq 1/2, & \text{then we can set the first digit to } 0, \\ -1 \leq r \leq 0, & \text{then we can set the first digit to } -1. \end{cases}$$

So intuitively this case distinction can be decided eventually. A real number

$0.d_0d_1d_2\cdots$ stands for the number

$$\sum_{i=0}^{\infty} d_i * 2^{-(i+1)}$$

If the first digit is 1 the maximum number we can represent is $0.1111111\cdots$ which is

$$2^{-1} + \sum_{i=1}^{\infty} 1 * 2^{-i} = 1$$

and the minimum number we can represent is $0.1 - 1 - 1 - 1 - 1\cdots$ which is

$$2^{-1} + \sum_{i=2}^{\infty} (-1) * 2^{-i} = 0$$

In general intuitively all numbers in the interval $[0, 1]$ can be represented by a signed digit number starting with digit 1. A similar calculation gives that if the first digit is 0 we get numbers in the interval

$$[0 * 2^{-1} + \sum_{i=2}^{\infty} (-1) * 2^{-i}, 0 * 2^{-1} + \sum_{i=2}^{\infty} 1 * 2^{-i}] = [-1/2, 1/2]$$

If the first digit is -1 we get numbers in the interval $[-1, 0]$.

A real number r has signed digit representations $0.a_0a_1a_2a_3\cdots$ where $a_i \in \text{Digit}$ if and only if

$$r = \sum_{i=0}^{\infty} d_i * 2^{-(i+1)}$$

It follows that if r has a signed digit representation then $r \in [-1, 1]$ because

$$-1 = \sum_{i=0}^{\infty} ((-1) * 2^{-(i+1)}) \leq \sum_{i=0}^{\infty} d_i * 2^{-(i+1)} = r \leq \sum_{i=0}^{\infty} (-1 * 2^{-(i+1)}) = 1$$

We see that r has signed digit representations $0.a_0a_1a_2a_3\cdots$ if and only if $r \in [-1, 1]$ and $r = (a_0 + y)/2$ for some y s.t. y has signed digit representations $0.a_1a_2a_3\cdots$ (choose $y = \sum_{i=0}^{\infty} d_{i+1} * 2^{-(i+1)}$). Let $\sim\mathbb{R}^+$ be the set $\sim\mathbb{R}^+$ of pairs (r, s) s.t. r has signed digit representation s . So (r, s) are in the relation if and only if $r \in [-1, 1]$

and $(2r - \text{head}(s), \text{tail}(s))$ are in this relation, or:

So $(r, s) \in \sim\mathbb{R}^+$ if and only if $r \in [-1, 1]$ and $(2r - \text{head}(s), \text{tail}(s)) \in \sim\mathbb{R}^+$.

This definition reads as the fixed point equation

$$\sim\mathbb{R}^+ = \{(r, s) \in [-1, 1] \times \text{Stream}(\text{Digit}) \mid (2r - \text{head}(s), \text{tail}(s)) \in \sim\mathbb{R}^+\}$$

There are many solutions for this equation. The largest fixed solution for this equation is $\sim\mathbb{R}^+$ as defined in the following:

Definition 6.1.1. (a) *The set of pairs (r, s) of a real number r with an signed digit stream s is defined as*

$$\begin{aligned} \sim\mathbb{R}^+ = \cup \{ & A \subseteq [-1, 1] \times \text{Stream}(\text{Digit}) \mid \\ & \forall (r, s) \in A. (2r - \text{head}(s), \text{tail}(s)) \in A \} \end{aligned}$$

(b) *We introduce a notion \sim as follows*

$$r \sim 0.a_0a_1a_2a_3 \cdots \Leftrightarrow (r, (a_0, a_1, a_2, a_3, \dots)) \in \sim\mathbb{R}^+$$

The problem is that to one real number correspond many streams. We see now that $\sim\mathbb{R}$ is usually the set of real numbers with signed digit representations:

Theorem 6.1.2.

$$(r, (a_0, a_1, a_2, a_3, \dots)) \in \sim\mathbb{R}^+ \Leftrightarrow r = \sum_{i=0}^{\infty} a_i * 2^{-(i+1)}$$

Proof. of “ \Rightarrow ”: Let $(r, s) \in A$, where A as in the definition of $\sim\mathbb{R}^+$ and $s = (a_0, a_1, a_2, a_3, \dots)$. Define $(r_n, s_n) \in A$ inductively by $(r_0, s_0) = (r, s)$, $(r_{n+1}, s_{n+1}) = (2r_n - \text{head}(s_n), \text{tail}(s_n))$. Then we have $s_n = (a_n, a_{n+1}, a_{n+2}, \dots)$, $(r_n, s_n) \in A$, $r_n \in [-1, 1]$, $r_{n+1} = 2r_n - \text{head}(s_n) = 2r_n - a_n$ and $r_n = (r_{n+1} + a_n)/2$. We show for all n

$$r = \sum_{i=0}^{n-1} a_i * 2^{-(i+1)} + 2^{-n} * r_n$$

by induction on n : Case $n = 0$: is trivial ($r = r_0$). Induction step $n \rightarrow n + 1$: by

IH

$$\begin{aligned}
 r &= \sum_{i=0}^{n-1} a_i * 2^{-(i+1)} + 2^{-n} * r_n \\
 &= \sum_{i=0}^{n-1} a_i * 2^{-(i+1)} + 2^{-n} * (r_{n+1}/2 + a_n/2) \\
 &= \sum_{i=0}^n a_i * 2^{-(i+1)} + 2^{-(n+1)} * r_{n+1}
 \end{aligned}$$

We have

$$\left| r - \sum_{i=0}^{n-1} a_i * 2^{-(i+1)} \right| = \left| 2^{-n} * r_n \right|$$

and therefore by $|r_n| \leq 1$

$$\sum_{i=0}^{n-1} a_i * 2^{-(i+1)} \rightarrow r \quad (n \rightarrow \infty)$$

Therefore,

$$r = \sum_{i=0}^{\infty} a_i * 2^{-(i+1)}$$

Proof of “ \Leftarrow ”: Let

$$r = \sum_{i=0}^{\infty} a_i * 2^{-(i+1)}$$

Let $r_n \in \mathbb{R}$ be defined by $r_0 = r$, $r_{n+1} = 2r_n - a_n$. We show

$$r_n = \sum_{i=0}^{\infty} a_{i+n} * 2^{-(i+1)} \in [-1, 1]$$

by induction on n : Case $n = 0$ is trivial. Induction step $n \rightarrow n + 1$:

$$\begin{aligned}
 r_{n+1} &= 2 * \sum_{i=0}^{\infty} a_{i+n} * 2^{-(i+1)} - a_n \\
 &= \sum_{i=0}^{\infty} a_{i+n} * 2^{-i} - a_n \\
 &= \sum_{i=1}^{\infty} a_{i+n} * 2^{-i} \\
 &= \sum_{i=0}^{\infty} a_{i+n+1} * 2^{-(i+1)}
 \end{aligned}$$

Let $s_n = (a_n, a_{n+1}, a_{n+2}, \dots)$. Then $(r_{n+1}, s_{n+1}) = (2r_n - \text{head}(s_n), \text{tail}(s_n))$, $A = \{(r_n, s_n) \mid n \in \mathbb{N}\} \subseteq \sim\mathbb{R}^+$, $(r, s) \in \sim\mathbb{R}^+$. \square

Definition 6.1.3. *We define*

$$\sim\mathbb{R} = \{r \in \mathbb{R} \mid \exists s \in \text{Stream}(\text{Digit}).(r, s) \in \sim\mathbb{R}^+\}$$

In next Chapter we will introduce $\sim\mathbb{R}$ in Agda as a codata type.

We prove the principle of guarded recursion for $\sim\mathbb{R}^+$:

Theorem 6.1.4. (Guarded recursion for $\sim\mathbb{R}^+$) *Assume A is a Set, $\text{next} : A \rightarrow A$ and*

$$f : A \rightarrow [-1, 1] \times \text{Stream}(\text{Digit})$$

s.t. for $a \in A$, if $f(a) = (r, s)$ then $f(\text{next}(a)) = (2r - \text{head}(s), \text{tail}(s))$. Then $\forall a \in A. f(a) \in \sim\mathbb{R}^+$.

Proof. Let $B = \{f(a) \mid a \in A\}$. We have $B \subseteq [-1, 1] \times \text{Stream}(\text{Digit})$. For all $(r, s) \in B$, $(r, s) = f(a)$ for some a therefore $(2r - \text{head}(s), \text{tail}(s)) = f(\text{next}(a)) \in B$. Therefore, $B \subseteq \sim\mathbb{R}^+$. So for $a \in A$ we have $f(a) \in B \subseteq \sim\mathbb{R}^+$. \square

Theorem 6.1.5. (Special case of Theorem 6.1.4) *Assume*

$$A \subseteq [-1, 1] \times \text{Stream}(\text{Digit})$$

s.t. $\forall (r, s) \in A. (2r - \text{head}(s), \text{tail}(s)) \in A$. Then $A \subseteq \sim\mathbb{R}^+$.

Proof. Take in Theorem 6.1.4 $f(a) = a$ and $\text{next}(r, s) = (2r - \text{head}(s), \text{tail}(s))$. \square

Theorem 6.1.6. (Special case of Theorem 6.1.4) *Assume I is a set, and assume for $i \in I$ that $r_i \in [-1, 1]$ and $s_i \in \text{Stream}(\text{Digit})$ s.t.*

$$\forall i \in I. \exists j \in I. (r_j = 2r_i - \text{head}(s_i) \wedge s_j = \text{tail}(s_i))$$

Then $\forall i \in I. (r_i, s_i) \in \sim\mathbb{R}^+$.

Proof. Let in Theorem 6.1.4 $A = I$ and define $\text{next} : A \rightarrow A$, as follows: if $f(i) = (r_i, s_i)$, choose j s.t. $r_j = 2r_i - \text{head}(s_i)$ and $s_j = \text{tail}(s_i)$. Then $\text{next}(i) = j$. (Here we use axiom of choice.) \square

Theorem 6.1.7. (Guarded recursion for $\sim\mathbb{R}$) *If A is a Set, $\text{next} : A \rightarrow A$, $f : A \rightarrow [-1, 1]$ and $d : A \rightarrow \text{Digit}$ s.t. for $a \in A$ we have $f(\text{next}(a)) = 2 * f(a) - d(a)$.*

Let

$$\begin{aligned} s &: A \rightarrow \text{Stream}(\text{Digit}) \\ s(a) &= (d(a), d(\text{next}(a)), d(\text{next}^2(a)), \dots) \end{aligned}$$

Then $\forall a \in A. (f(a), s(a)) \in \sim\mathbb{R}^+$ and therefore $\forall a \in A. f(a) \in \sim\mathbb{R}$.

Proof. Let $f' : A \rightarrow [-1, 1] \times \text{Stream}(\text{Digit})$, $f'(a) = (f(a), s(a))$. Then if $f'(a) = (r, s)$ then $f'(\text{next}(a)) = (2r - \text{head}(s), \text{tail}(s))$. So by Theorem 6.1.4, $f'(a) \in \sim\mathbb{R}^+$, therefore $(f(a), s(a)) \in \sim\mathbb{R}^+$, $f(a) \in \sim\mathbb{R}$. \square

Theorem 6.1.8. (Special case of Theorem 6.1.7) *Let $A \subseteq [-1, 1]$, $d : A \rightarrow \text{Digit}$ s.t. $\forall r \in A. 2r - d(r) \in A$. Then $A \subseteq \sim\mathbb{R}$.*

Proof. Let in Theorem 6.1.7 $f(r) = r$ and $\text{next}(r) = 2 * r - d(r)$. \square

Theorem 6.1.9. (Special case of Theorem 6.1.8) *Let $A \subseteq [-1, 1]$ s.t. $\forall r \in A. \exists d \in \text{Digit}. 2r - d \in A$. Then $A \subseteq \sim\mathbb{R}$.*

Proof. Using Axiom of choice define $d : A \rightarrow \text{Digit}$ s.t. $\forall r \in A. 2r - d(r) \in A$. Then the conclusion follows by Theorem 6.1.8. \square

Lemma 6.1.10. *If $(r, s) \in \sim\mathbb{R}^+$ then $(2r - \text{head}(s), \text{tail}(s)) \in \sim\mathbb{R}^+$ and $2r - \text{head}(s) \in \sim\mathbb{R}$.*

Proof. $(r, s) \in A$ for some A as in the definition 6.1.1 (a) of $\sim\mathbb{R}^+$. Therefore $(2 * r - \text{head}(s), \text{tail}(s)) \in A \subseteq \sim\mathbb{R}^+$. \square

The following Theorem 6.1.11 shows that under certain conditions a function f defined by guarded recursion has a computational meaning i.e. we can compute from input data the stream for the output of f .

By computational data types we mean finitely representable types such as finite types, $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ for which there is a notion of computation defined on them.

Theorem 6.1.11. (Computational guarded recursion) *Assume $k \in \mathbb{N}$, B_1, \dots, B_k are computational data types. Let*

$$f : [-1, 1]^n \times B_1 \times \dots \times B_k \rightarrow [-1, 1]$$

Assume we can compute from

$$\begin{aligned} s_1, \dots, s_n &\in \text{Stream}(\text{Digit}), \\ b_1 \in B_1, \dots, b_k &\in B_k \end{aligned}$$

a digit

$$\hat{d}(s_1, \dots, s_n, b_1, \dots, b_k) \in \text{Digit}$$

streams

$$\hat{s}_i(s_1, \dots, s_n, b_1, \dots, b_k) \in \text{Stream}(\text{Digit})$$

and

$$\hat{b}_i(s_1, \dots, s_n, b_1, \dots, b_k) \in B_i$$

such that

$$\begin{aligned} \forall r_1, \dots, r_n &\in [-1, 1]. \\ \forall s_1, \dots, s_n &\in \text{Stream}(\text{Digit}). \\ \forall b_1 \in B_1, \dots, b_k \in B_k. &(r_1, s_1), \dots, (r_n, s_n) \in \sim\mathbb{R}^+ \\ \rightarrow \exists r'_1, \dots, r'_n &\in \mathbb{R}. \\ &2 * f(r_1, \dots, r_n, b_1, \dots, b_k) - \hat{d}(s_1, \dots, s_n, b_1, \dots, b_k) \\ &= f(r'_1, \dots, r'_n, \\ &\quad \hat{b}_1(s_1, \dots, s_n, b_1, \dots, b_k), \dots, \hat{b}_k(s_1, \dots, s_n, b_1, \dots, b_k)) \\ &\wedge (r'_1, \hat{s}_1(s_1, \dots, s_n, b_1, \dots, b_k)), \dots, (r'_n, \hat{s}_n(s_1, \dots, s_n, b_1, \dots, b_k)) \in \sim\mathbb{R}^+ \end{aligned}$$

Then we can compute from $(s_1, \dots, s_n, b_1, \dots, b_k)$ a stream

$$\hat{s}(s_1, \dots, s_n, b_1, \dots, b_k) \in \text{Stream}(\text{Digit})$$

s.t.

$$\begin{aligned} \forall r_1, \dots, r_n &\in [-1, 1]. \\ s_1, \dots, s_n &\in \text{Stream}(\text{Digit}). \\ b_1 \in B_1, \dots, b_k &\in B_k. \\ (r_1, s_1), \dots, (r_n, s_n) &\in \sim\mathbb{R}^+ \\ \rightarrow (f(r_1, \dots, r_n, b_1, \dots, b_k), \hat{s}(s_1, \dots, s_n, b_1, \dots, b_k)) &\in \sim\mathbb{R}^+ \end{aligned}$$

Proof. Compute

$$\begin{aligned}
 \hat{s}(s_1, \dots, s_n, b_1, \dots, b_k) = & \hat{d}(s_1, \dots, s_n, b_1, \dots, b_k) \\
 & : \hat{s}(\hat{s}_1(s_1, \dots, s_n, b_1, \dots, b_k), \\
 & \quad \hat{s}_2(s_1, \dots, s_n, b_1, \dots, b_k), \\
 & \quad \quad \vdots, \\
 & \quad \hat{s}_n(s_1, \dots, s_n, b_1, \dots, b_k), \\
 & \quad \hat{b}_1(s_1, \dots, s_n, b_1, \dots, b_k), \\
 & \quad \quad \vdots, \\
 & \quad \hat{b}_k(s_1, \dots, s_n, b_1, \dots, b_k))
 \end{aligned}$$

This obviously computes a stream. We show the assertion using \hat{s} in Theorem 6.1.4. Let

$$\begin{aligned}
 A = \{ & (r_1, \dots, r_n, s_1, \dots, s_n, b_1, \dots, b_k) \mid (r_1, s_1), \dots, (r_n, s_n) \in \sim\mathbb{R}^+ \\
 & \wedge b_1 \in B_1 \wedge \dots \wedge b_k \in B_k \}
 \end{aligned}$$

We define

$$\begin{aligned}
 g : A & \rightarrow [-1, 1] \times \text{Stream}(\text{Digit}) \\
 g(r_1, \dots, r_n, s_1, \dots, s_n, b_1, \dots, b_k) = & \\
 & (f(r_1, \dots, r_n, b_1, \dots, b_k), \hat{s}(s_1, \dots, s_n, b_1, \dots, b_k))
 \end{aligned}$$

next : $A \rightarrow A$

$$\begin{aligned}
 \text{next}(r_1, \dots, r_n, s_1, \dots, s_n, b_1, \dots, b_k) = & \\
 & (r'_1, \dots, r'_n, \hat{s}_1(s_1, \dots, s_n, b_1, \dots, b_k), \\
 & \quad \quad \quad \vdots \\
 & \quad \quad \hat{s}_n(s_1, \dots, s_n, b_1, \dots, b_k), \\
 & \quad \hat{b}_1(s_1, \dots, s_n, b_1, \dots, b_k), \\
 & \quad \quad \quad \vdots \\
 & \quad \hat{b}_k(s_1, \dots, s_n, b_1, \dots, b_k))
 \end{aligned}$$

where r'_1, \dots, r'_n are s.t.

$$\begin{aligned} 2 * f(r_1, \dots, r_n, b_1, \dots, b_k) - \hat{d}(s_1, \dots, s_n, b_1, \dots, b_k) \\ = f(r'_1, \dots, r'_n, \hat{b}_1(s_1, \dots, s_n, b_1, \dots, b_k), \\ \vdots \\ \hat{b}_k(s_1, \dots, s_n, b_1, \dots, b_k)) \end{aligned}$$

and $(r'_i, \hat{s}_i(s_1, \dots, s_n, b_1, \dots, b_k)) \in \sim\mathbb{R}^+$.

If $g(r_1, \dots, r_n, s_1, \dots, s_n, b_1, \dots, b_k) = (r, s)$

then $\text{next}(r_1, \dots, r_n, s_1, \dots, s_n, b_1, \dots, b_k) = (r'_1, \dots, r'_n, s'_1, \dots, s'_n, b'_1, \dots, b'_k)$

where

$$\begin{aligned} s'_i &= \hat{s}_i(s_1, \dots, s_n, b_1, \dots, b_k) \\ b'_i &= \hat{b}_i(s_1, \dots, s_n, b_1, \dots, b_k) \\ r &= f(r_1, \dots, r_n, b_1, \dots, b_k) \\ \text{head}(s) &= \hat{d}(s_1, \dots, s_n, b_1, \dots, b_k) \\ 2r - \text{head}(s) &= f(r'_1, \dots, r'_n, b'_1, \dots, b'_k) \\ \text{tail}(s) &= \hat{s}(s'_1, \dots, s'_n, b'_1, \dots, b'_k) \end{aligned}$$

So

$$(2r - \text{head}(s), \text{tail}(s)) = g(\text{next}(r_1, \dots, r_n, s_1, \dots, s_n, b_1, \dots, b_k))$$

Therefore, by Theorem 6.1.4 $g(r_1, \dots, r_n, s_1, \dots, s_n, b_1, \dots, b_k) \in \sim\mathbb{R}^+$. \square

6.2 The signed Digit Representations of Real Numbers -1, 0 and 1

We determine the signed digit representation of real number 1. If the first digit is

$$\begin{cases} 0, & \text{then } 2 * 1 - 0 = 2 \notin [-1, 1], \\ 1, & \text{then } 2 * 1 - 1 = 1 \in [-1, 1], \\ -1, & \text{then } 2 * 1 - (-1) = 3 \notin [-1, 1]. \end{cases}$$

So the only choice for the first digit is 1. We note that $2 * 1 - 1 = 1$. We show $1 \sim 0.11111\dots$ by using Theorem 6.1.6. Take $I = \{0\}$, $r_0 = 1$ and

$s_0 = (1, 1, 1, 1, \dots)$. We have $2r_0 - \text{head}(s_0) = r_0$, $\text{tail}(s_0) = s_0$. Therefore $(1, (1, 1, 1, \dots)) \in \sim\mathbb{R}^+$, $1 \sim 0.11111\dots$.

Similarly for $r = -1$ if the first digit is

$$\begin{cases} 0, & \text{then } 2 * (-1) - 0 = -2 \notin [-1, 1], \\ 1, & \text{then } 2 * (-1) - 1 = -3 \notin [-1, 1], \\ -1, & \text{then } 2 * (-1) - (-1) = -1 \in [-1, 1]. \end{cases}$$

So the only choice for the first digit is -1 . We note that $2 * (-1) - (-1) = -1$. By the same argument as before we get $-1 \sim 0.(-1)(-1)(-1)(-1)(-1)\dots$.

Now for $r = 0$ if the first digit is

$$\begin{cases} 0, & \text{then } 2 * 0 - 0 = 0 \in [-1, 1], \\ 1, & \text{then } 2 * 0 - 1 = -1 \in [-1, 1], \\ -1, & \text{then } 2 * 0 - (-1) = 1 \in [-1, 1]. \end{cases}$$

That shows that we can choose the first digit starting with 0, 1 and -1 respectively. Starting with first digit 0 we get $2 * 0 - 0 = 0$. As before $0 \sim 0.000\dots$. Next, we choose the first digit to be 1. We show $0 \sim 0.1(-1)(-1)(-1)\dots$ by using Theorem 6.1.6: let $I = \{0, 1\}$,

$$\begin{aligned} r_0 &= 0, & s_0 &= (1, -1, -1, -1, \dots) \\ r_1 &= -1, & s_1 &= (-1, -1, -1, \dots) \end{aligned}$$

Then $2r_0 - \text{head}(s_0) = r_1$, $\text{tail}(s_0) = s_1$, $2r_1 - \text{head}(s_1) = r_1$ and $\text{tail}(s_1) = s_1$. Therefore $(r_0, s_0) = (0, (1, -1, -1, \dots)) \in \sim\mathbb{R}^+$, $0 \sim 0.1(-1)(-1)\dots$. Finally we choose as first digit to be -1 . We show $0 \sim 0.(-1)111\dots$ by Theorem 6.1.6 $I = \{0, 1\}$,

$$\begin{aligned} r_0 &= 0, & s_0 &= (-1, 1, 1, 1, \dots) \\ r_1 &= 1, & s_1 &= (1, 1, 1, \dots) \end{aligned}$$

As before we get $(0, (-1, 1, 1, 1, \dots)) \in \sim\mathbb{R}^+$, $0 \sim 0.(-1)111\dots$.

6.3 The signed Digit Representations of Rational Numbers

We embed the rational numbers in $[-1, 1]$ into $\sim\mathbb{R}$, i.e. we show $\forall q \in \mathbb{Q} \cap [-1, 1]. q \in \sim\mathbb{R}$. For $q \in \mathbb{Q} \cap [-1, 1]$ we can choose the first digit to be

$$\begin{cases} -1, & \text{if } q \in [-1, 0] & -2 * q - (-1) \in [-1, 1], \\ 0, & \text{if } q \in [-1/2, 1/2] & -2 * q - 0 \in [-1, 1], \\ 1, & \text{if } q \in [0, 1] & -2 * q - 1 \in [-1, 1]. \end{cases}$$

Then by guarded recursion we can determine the second digit and so on.

So as choice for the first digit of q we can choose any $d : \mathbb{Q} \cap [-1, 1] \rightarrow \text{Digit}$ s.t.

$$\begin{cases} d(q) = -1, & \text{if } q \in [-1, -1/2[\\ d(q) \in \{-1, 0\}, & \text{if } q \in [-1/2, 0[\\ d(q) \in \{-1, 0, 1\}, & \text{if } q = 0 \\ d(q) \in \{0, 1\}, & \text{if } q \in]0, 1/2] \\ d(q) = 1, & \text{if } q \in]1/2, 1] \end{cases}$$

Then $\forall q \in \mathbb{Q} \cap [-1, 1]. 2q - d(q) \in [-1, 1]$. One choice is

$$d(q) = \begin{cases} -1, & \text{if } q \in [-1, 0[\\ 1, & \text{if } q \in [0, 1]. \end{cases}$$

Then by Theorem 6.1.8 we get $A := \mathbb{Q} \cap [-1, 1] \subseteq \sim\mathbb{R}$. By Theorem 6.1.11 with $n = 0, k = 1, B_1 = \mathbb{Q} \cap [-1, 1], \hat{d}(q) = d(q), \hat{b}_1(q) = 2q - d(q)$, we can compute for $q \in \mathbb{Q} \cap [-1, 1]$ a stream s s.t. $(q, s) \in \sim\mathbb{R}^+$.

6.4 Average

We are going to define an average function $\text{av} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ and show $\text{av}(\sim\mathbb{R} \times \sim\mathbb{R}) \subseteq \sim\mathbb{R}$. We cannot apply guarded recursion directly in order to show $\text{av}(\sim\mathbb{R} \times \sim\mathbb{R}) \subseteq \sim\mathbb{R}$. Instead we introduce a function $\text{avaux} : \mathbb{R} \times \mathbb{R} \times \text{Digit2} \rightarrow \mathbb{R}$ where $\text{Digit2} = \{-2, -1, 0, 1, 2\}$ which can be defined by guarded recursion and show

$\text{avaux}(\sim\mathbb{R} \times \sim\mathbb{R} \times \text{Digit2}) \subseteq \sim\mathbb{R}$. In Section 6.4.2 we will use function avaux in order to show $\text{av}(\sim\mathbb{R} \times \sim\mathbb{R}) \subseteq \sim\mathbb{R}$.

6.4.1 Function avaux

The closure of $\sim\mathbb{R}$ under av is shown using avaux

$$\begin{aligned} \text{avaux} &: \mathbb{R} \times \mathbb{R} \times \text{Digit2} \rightarrow \mathbb{R} \\ \text{avaux}(a, b, i) &= (a + b + i)/4 \end{aligned}$$

where $\text{Digit2} = \{-2, -1, 0, 1, 2\}$, $\text{avaux}([-1, 1] \times [-1, 1] \times \text{Digit2}) \subseteq [-1, 1]$. We show $\sim\mathbb{R}^+$ is closed under avaux :

Lemma 6.4.1. *For any $a, b \in \sim\mathbb{R}$, $i \in \text{Digit2}$ we have $\text{avaux}(a, b, i) \in \sim\mathbb{R}$. Furthermore, if $(a, s^a), (b, s^b) \in \sim\mathbb{R}^+$ we can compute from s^a, b, s^b, i a stream s s.t. $(\text{avaux}(a, b, i), s) \in \sim\mathbb{R}^+$.*

Proof. We use Theorem 6.1.7 (guarded recursion for $\sim\mathbb{R}$). Let

$$A = \{(a, s^a, b, s^b, i) \mid (a, s^a), (b, s^b) \in \sim\mathbb{R}^+, i \in \text{Digit2}\}$$

$f : A \rightarrow [-1, 1]$ s.t. $f(a, s^a, b, s^b, i) = \text{avaux}(a, b, i)$.

We need to determine functions $d : A \rightarrow \text{Digit}$ and $\text{next} : A \rightarrow A$ s.t. for $x \in A$ we have $f(\text{next}(x)) = 2 * f(x) - d(x)$. Then for $(a, s^a), (b, s^b) \in \sim\mathbb{R}^+$, $i \in \text{Digit2}$ $\text{avaux}(a, b, i) = f(a, s^a, b, s^b, i) \in \sim\mathbb{R}$.

Let for $(a, s^a, b, s^b, i) \in A$,

$$\begin{aligned} a_0 &= \text{head}(s^a), & a' &= 2a - a_0 \\ b_0 &= \text{head}(s^b), & b' &= 2b - b_0 \end{aligned}$$

Then by

$$(a, s^a) \in \sim\mathbb{R}^+, \text{ we get } (a', \text{tail}(s_a)) \in \sim\mathbb{R}^+$$

Similarly $(b', \text{tail}(s_b)) \in \sim\mathbb{R}^+$. Assume $d' \in \text{Digit}$. We have

$$\begin{aligned}
 2 * \text{avaux}(a, b, i) - d' &= 2 * \frac{\frac{a'+a_0}{2} + \frac{b'+b_0}{2} + i}{4} - d' \\
 &= \frac{a'+b'+(a_0+b_0+2i-4d')}{4} \\
 &= \frac{a'+b'+(j-4d')}{4} \\
 &= \frac{a'+b'+i'}{4} \\
 &= \text{avaux}(a', b', i')
 \end{aligned}$$

where $j = a_0 + b_0 + 2i$ and $i' = j - 4d'$. If we have chosen d' s.t. $i' \in \text{Digit2}$ then we can choose $d(a, s^a, b, s^b, i) = d'$, $\text{next}(a, s^a, b, s^b, i) = (a', \text{tail}(s^a), b', \text{tail}(s^b), i')$ since

$$\begin{aligned}
 f(\text{next}(a, s^a, b, s^b, i)) &= f(a', \text{tail}(s^a), b', \text{tail}(s^b), i') \\
 &= \text{avaux}(a', b', i') \\
 &= 2 * \text{avaux}(a, b, i) - d' \\
 &= 2 * f(a, s^a, b, s^b, i) - d(a, s^a, b, s^b, i)
 \end{aligned}$$

The calculation of d' is as follows: we have $j \in [-6, 6]$ since $a_0, b_0 \in \{-1, 0, 1\}$ and $i \in \{-2, -1, 0, 1, 2\}$. Let d' be chosen as

$$d' = \begin{cases} 0, & \text{if } -2 \leq j \leq 2; \\ 1, & \text{if } j > 2; \\ -1, & \text{if } j < -2 \end{cases}$$

Then $i' = j - 4d' \in [-2, 2]$.

From s^a, s^b, i we can compute $d', \text{tail}(s^a), \text{tail}(s^b), i'$ and have $(a', \text{tail}(s^a)), (b', \text{tail}(s^b)) \in \sim\mathbb{R}^+$. Therefore by Theorem 6.1.11 we can compute the stream s for $\text{avaux}(a, b, i)$ from s^a, s^b, i .

□

6.4.2 Function av

Using avaux we will show $\text{av}(\sim\mathbb{R} \times \sim\mathbb{R}) \subseteq \sim\mathbb{R}$: let $a, b \in \sim\mathbb{R}$ and $(a, a_0 : s^a), (b, b_0 : s^b) \in \sim\mathbb{R}^+$. Let $a' = 2a - a_0, b' = 2b - b_0$. Then $(a', s^a), (b', s^b) \in \sim\mathbb{R}^+, a', b' \in \sim\mathbb{R}$.

We have

$$a = \frac{a' + a_0}{2} \quad b = \frac{b' + b_0}{2}$$

Let $i = a_0 + b_0$. Then, by Lemma 6.4.1

$$\begin{aligned} \text{av}(a, b) &= \frac{a+b}{2} = \frac{\frac{a'+a_0}{2} + \frac{b'+b_0}{2}}{2} = \frac{a'+a_0+b'+b_0}{4} = \frac{a'+b'+(a_0+b_0)}{4} \\ &= \text{avaux}(a', b', i) \in \sim\mathbb{R} \end{aligned}$$

This gives as well a way to compute from s^a, s^b, i a stream for $\text{avaux}(a', b', i)$.

Theorem 6.4.2. *For any $a, b \in \sim\mathbb{R}$ we have $\text{av}(a, b) \in \sim\mathbb{R}$. Furthermore if $(a, s^a), (b, s^b) \in \sim\mathbb{R}^+$ we can compute from s^a, s^b a stream s s.t. $(\text{av}(a, b), s) \in \sim\mathbb{R}^+$.*

Proof. Let $a, b \in \sim\mathbb{R}$, so $(a, s^a), (b, s^b) \in \sim\mathbb{R}^+$ for some s^a and s^b

$$\begin{aligned} a_0 &= \text{head}(s^a), & a' &= 2a - a_0, & \text{then } (a', \text{tail}(s^a)) &\in \sim\mathbb{R}^+ \\ b_0 &= \text{head}(s^b), & b' &= 2b - b_0, & \text{then } (b', \text{tail}(s^b)) &\in \sim\mathbb{R}^+ \end{aligned}$$

Let $i = a_0 + b_0$. Then

$$\begin{aligned} \text{av}(a, b) &= \frac{a+b}{2} = \frac{\frac{a'+a_0}{2} + \frac{b'+b_0}{2}}{2} = \frac{a'+a_0+b'+b_0}{4} = \frac{a'+b'+(a_0+b_0)}{4} \\ &= \text{avaux}(a', b', i) \end{aligned}$$

Therefore, $\text{av}(a, b) = \text{avaux}(a', b', i) \in \sim\mathbb{R}$ by Lemma 6.4.1.

From s^a, s^b we can compute $a_0, a', b_0, b', \text{tail}(s^a), \text{tail}(s^b)$ and have $(a', \text{tail}(s^a)), (b', \text{tail}(s^b)) \in \sim\mathbb{R}^+$. Therefore, we can by Theorem 6.1.11 compute the stream s for $\text{av}(a, b)$. \square

6.5 Multiplication

We define a multiplication function $\text{mp} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, $\text{mp}(a, b) = a * b$. We show $\text{mp}(\sim\mathbb{R} \times \sim\mathbb{R}) \subseteq \sim\mathbb{R}$. In order to show this we introduce function

$$\begin{aligned} \text{mpaux} &: \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \text{Digit2} \rightarrow \mathbb{R} \\ \text{mpaux}(a, b, c, i) &= (a * b + c + i)/4 \end{aligned}$$

We show

$$\text{mpaux}(\sim\mathbb{R} \times \sim\mathbb{R} \times \sim\mathbb{R} \times \text{Digit2}) \subseteq \sim\mathbb{R}$$

Therefore for $a, b, c \in \sim\mathbb{R}$, $i \in \text{Digit2}$, $r \in [-1, 1]$

$$r = \text{mpaux}(a, b, c, i) = \frac{a * b + c + i}{4} \in \sim\mathbb{R}$$

Let $a, b \in \sim\mathbb{R}$. Then $\text{mp}(a, b) = 4 * \text{mpaux}(a, b, 0, 0)$. Let $r = \text{mpaux}(a, b, 0, 0)$. If we have

$$\forall r \in \sim\mathbb{R}. 4r \in [-1, 1] \rightarrow 4r \in \sim\mathbb{R}$$

then we get $\text{mp}(\sim\mathbb{R} \times \sim\mathbb{R}) \subseteq \sim\mathbb{R}$. So in order to show $\text{mp}(\sim\mathbb{R} \times \sim\mathbb{R}) \subseteq \sim\mathbb{R}$, we need to show $\text{mpaux}(\sim\mathbb{R} \times \sim\mathbb{R} \times \sim\mathbb{R} \times \text{Digit2}) \subseteq \sim\mathbb{R}$ (Section 6.5.1) and $\forall r \in \sim\mathbb{R}. 4r \in [-1, 1] \rightarrow 4r \in \sim\mathbb{R}$ (Section 6.5.3).

6.5.1 Function mpaux

Let

$$\begin{aligned} \text{mpaux} &: \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \text{Digit2} \rightarrow \mathbb{R} \\ \text{mpaux}(a, b, c, i) &= (a * b + c + i)/4 \end{aligned}$$

Note $\text{mpaux}([-1, 1]^3 \times \text{Digit2}) \subseteq [-1, 1]$.

Lemma 6.5.1. *For any $a, b, c \in \sim\mathbb{R}$, $i \in \text{Digit2}$ we have $\text{mpaux}(a, b, c, i) \in \sim\mathbb{R}$. Furthermore, if $(a, s^a), (b, s^b), (c, s^c) \in \sim\mathbb{R}^+$ we can compute from s^a, s^b, s^c, i a stream s s.t. $(\text{mpaux}(a, b, c, i), s) \in \sim\mathbb{R}^+$.*

Proof. We use Theorem 6.1.7 (guarded recursion for $\sim\mathbb{R}$). Let

$$A = \{(a, s^a, b, s^b, c, s^c, i) \mid (a, s^a), (b, s^b), (c, s^c) \in \sim\mathbb{R}^+, i \in \text{Digit2}\}$$

$f : [-1, 1] \rightarrow \mathbb{R}$ s.t. $f(a, s^a, b, s^b, c, s^c, i) = \text{mpaux}(a, b, c, i)$.

We need to determine functions $d : A \rightarrow \text{Digit}$ and $\text{next} : A \rightarrow A$ s.t. for $x \in A$ we have $f(\text{next}(x)) = 2 * f(x) - d(x)$. Then for $(a, s^a), (b, s^b), (c, s^c) \in \sim\mathbb{R}^+$, $i \in \text{Digit2}$, $\text{mpaux}(a, b, c, i) = f(a, s^a, b, s^b, c, s^c, i) \in \sim\mathbb{R}$.

Let $(a, s^a, b, s^b, c, s^c, i) \in A$,

$$\begin{aligned} a_0 &= \text{head}(s^a), & a' &= 2a - a_0 \\ c_0 &= \text{head}(s^c), & c' &= 2c - c_0 \end{aligned}$$

Then by

$$(a, s^a) \in \sim\mathbb{R}^+, \text{ we get } (a', \text{tail}(s^a)) \in \sim\mathbb{R}^+$$

Similarly $(c', \text{tail}(s_c)) \in \sim\mathbb{R}^+$. Assume $d' \in \text{Digit}$. Then

$$\begin{aligned} 2 * \text{mpaux}(a, b, c, i) - d' &= 2 * \frac{\frac{a'+a_0}{2} * b + \frac{c'+c_0}{2} + i}{4} - d' \\ &= \frac{(a'+a_0) * b + c' + c_0 + 2i - 4d'}{4} \\ &= \frac{a' * b + (a_0 * b + c' + i) + c_0 + i - 4d'}{4} \\ &= \frac{a' * b + 4e + c_0 + i - 4d'}{4} \end{aligned}$$

where

$$e = \frac{a_0 * b + c' + i}{4}$$

Below we will show $a_0 * b \in \sim\mathbb{R}$ (Lemma 6.5.2). Let

$$e = \frac{a_0 * b + c' + i}{4} = \text{avaux}(a_0 * b, c', i) \in \sim\mathbb{R}$$

Therefore, by Lemma 6.4.1 $(e, s^e) \in \sim\mathbb{R}^+$ for some s^e . Note we can compute s^e from s^a, s^b, s^c, i . Let

$$\begin{aligned} e_0 &= \text{head}(s^e), & e' &= 2e - e_0, & \text{then } (e', \text{tail}(s^e)) &\in \sim\mathbb{R}^+ \\ e_1 &= \text{head}(\text{tail}(s^e)), & e'' &= 2e' - e_1, & \text{then } (e'', \text{tail}^2(s^e)) &\in \sim\mathbb{R}^+ \end{aligned}$$

So we have

$$\begin{aligned} e &= \frac{e' + e_0}{2} = \frac{\frac{e'' + e_1}{2} + e_0}{2} = \frac{2e_0 + e_1 + e''}{4} \\ a_0 * b + c' + i &= 4e = 2e_0 + e_1 + e'' \end{aligned}$$

Then

$$\begin{aligned}
 2 * \text{mpaux}(a, b, c, i) - d' &= \frac{a' * b + 4e + c_0 + i - 4d'}{4} \\
 &= \frac{a' * b + 2e_0 + e_1 + e'' + c_0 + i - 4d'}{4} \\
 &= \frac{a' * b + e'' + 2e_0 + e_1 + c_0 + i - 4d'}{4} \\
 &= \frac{a' * b + e'' + j - 4d'}{4} \\
 &= \frac{a' * b + e'' + i'}{4} \\
 &= \text{mpaux}(a', b, e'', i')
 \end{aligned}$$

where $j = 2e_0 + e_1 + c_0 + i$ and $i' = j - 4d'$. If we have chosen d' s.t. $i' \in \text{Digit}2$ then we can choose above, $d(a, s^a, b, s^b, c, s^c, i) = d'$, $\text{next}(a, s^a, b, s^b, c, s^c, i) = (a', \text{tail}(s^a), b, s^b, e'', \text{tail}^2(s^e), i')$ since

$$\begin{aligned}
 f(\text{next}(a, s^a, b, s^c, c, s^c, i)) &= f(a', \text{tail}(s^a), b, s^b, e'', \text{tail}^2(s^e), i') \\
 &= \text{mpaux}(a', b, e'', i') \\
 &= 2 * \text{mpaux}(a, b, c, i) - d' \\
 &= 2 * f(a, s^a, b, s^b, c, s^c, i) - d(a, s^a, b, s^b, c, s^c, i)
 \end{aligned}$$

The calculation of d' is as follows: we have $j \in [-6, 6]$ since $e_0, e_1, c_0 \in \{-1, 0, 1\}$ and $i \in \{-2, -1, 0, 1, 2\}$. Therefore, let d' be chosen as

$$d' = \begin{cases} 0, & \text{if } -2 \leq j \leq 2; \\ 1, & \text{if } j > 2; \\ -1, & \text{if } j < -2 \end{cases}$$

Then $i' = j - 4d' \in [-2, 2]$.

From s^a, s^b, s^c, s^c, i we can compute the first digit $d', a', e, e'', \text{tail}(s^a), \text{tail}^2(s^e), i'$, once we have shown Lemma 6.5.2 (see below), and have $(a', \text{tail}(s^a)), (e, s^e), (e'', \text{tail}^2(s^e)) \in \sim\mathbb{R}^+$. Therefore, we can by Theorem 6.1.11 compute the stream s for $\text{mpaux}(a, b, c, i)$.

□

Lemma 6.5.2. *For $d \in \text{Digit}$, $b \in \sim\mathbb{R}$ we have $d * b \in \sim\mathbb{R}$. Furthermore, if $(b, s^b) \in \sim\mathbb{R}^+$ we can compute from d and s^b a stream for $d * b$.*

Proof. Let $b \in \text{Digit}$, $A = \{d * b \mid b \in \sim\mathbb{R}\}$, $A \subseteq [-1, 1]$ since $\sim\mathbb{R} \subseteq [-1, 1]$. If

$b \in \sim\mathbb{R}$, $(b, s^b) \in \sim\mathbb{R}^+$ then

$$2 * (d * b) - d * \text{head}(s^b) = d * (2b - \text{head}(s^b)) \in A$$

since $(2b - \text{head}(s^b)) \in \sim\mathbb{R}$. By Lemma 6.1.9 $A \subseteq \sim\mathbb{R}$ and by Theorem 6.1.11 we can compute a stream s for $d * b$ from d and a stream for b . \square

As indicated at the beginning of Section 6.5 the second step for showing closure of $\sim\mathbb{R}$ under multiplication is to show that for $r \in \sim\mathbb{R} \cap [-1/4, 1/4]$ we have $4r \in \sim\mathbb{R}$. This will be shown using the function `addR`.

6.5.2 Function `addR`

Let

$$\begin{aligned} \text{addR} &: \mathbb{Q} \times \mathbb{R} \rightarrow \mathbb{R} \\ \text{addR}(u, a) &= u + a \end{aligned}$$

Lemma 6.5.3. *For any $u \in \mathbb{Q}$, $a \in \sim\mathbb{R}$ such that $u + a \in [-1, 1]$ we have $\text{addR}(u, a) \in \sim\mathbb{R}$. Furthermore, if $(a, s^a) \in \sim\mathbb{R}^+$ we can compute from u, s^a a stream s s.t. $(\text{addR}(u, a), s) \in \sim\mathbb{R}^+$.*

Proof. We use Theorem 6.1.7 (guarded recursion for $\sim\mathbb{R}$). Let

$$A = \{(u, a, s^a) \mid u \in \mathbb{Q}, (a, s^a) \in \sim\mathbb{R}^+, u + a \in [-1, 1]\}$$

$f : A \rightarrow \mathbb{R}$ s.t. $f(u, a, s^a) = \text{mpaux}(u, a)$.

We need to determine functions $d : A \rightarrow \text{Digit}$ and $\text{next} : A \rightarrow A$ s.t. for $x \in A$ we have $f(\text{next}(x)) = 2 * f(x) - d(x)$. Then for $u \in \mathbb{Q}$, $(a, s^a) \in \sim\mathbb{R}^+$ s.t. $u + a \in [-1, 1]$ we have $\text{addR}(u, a) = f(u, a, s^a) \in \sim\mathbb{R}$.

Let $(u, a, s^a) \in A$,

$$\begin{aligned} a_0 &= \text{head}(s^a), & a' &= 2a - a_0, & \text{then } (a', \text{tail}(s^a)) &\in \sim\mathbb{R}^+ \\ a_1 &= \text{head}(\text{tail}(s^a)), & a'' &= 2a' - a_1, & \text{then } (a'', \text{tail}^2(s^a)) &\in \sim\mathbb{R}^+ \end{aligned}$$

Then

$$a = \frac{a_0 + a'}{2} \quad a' = \frac{a_1 + a''}{2}$$

Assume $d' \in \text{Digit}$. We have

$$\begin{aligned}
 2 * \text{addR}(u, a) - d' &= 2 * (u + \frac{a_0 + a'}{2}) - d' \\
 &= 2u + a_0 + a' - d' \\
 &= 2u + a_0 - d' + a' \\
 &= \text{addR}(2u + a_0 - d', a')
 \end{aligned}$$

With $\text{next}(u, a, s^a) = (2u + a_0 - d', a', \text{tail}(s^a))$ this is an instance of guarded recursion (Theorem 6.1.7) provided

$$2(u + a) - d' \in [-1, 1]$$

So we need to compute d' such that this relation holds.

Note that $a \in \mathbb{R}$ so we cannot directly compute d' . We need to carry out some calculation first. We have

$$2(u + a) - d' = 2(u + a_0/2 + a_1/4) + 2(a - a_0/2 - a_1/4) - d' \stackrel{!}{\in} [-1, 1]$$

Let $q = (u + a_0/2 + a_1/4)$, $b = (a - a_0/2 - a_1/4)$. So we need to determine d' s.t. $2q + 2b - d' \in [-1, 1]$. We know

$$4b = 4 * (a - a_0/2 - a_1/4) = a'' \in [-1, 1]$$

so $b \in [-1/4, 1/4]$. Furthermore, $2(q+b) = 2(u+a) \in [-2, 2]$ since $u+a \in [-1, 1]$ we have $q \in \mathbb{Q}$, so we can compare computationally q with other elements of \mathbb{Q} . If $q \geq 1/4$ and $d' = 1$ then

$$\underbrace{\underbrace{2q}_{\geq 1/2} + \underbrace{2b}_{\geq -1/2}}_{\geq 0} - \underbrace{d'}_1 \geq -1$$

and

$$\underbrace{2(q+b)}_{\leq 2} - \underbrace{d'}_1 \leq 1$$

If $q \in [-1/4, 1/4]$ and $d' = 0$ then

$$\underbrace{2q}_{\in[-1/2, 1/2]} + \underbrace{2b}_{\in[-1/2, 1/2]} - \underbrace{d'}_0 \geq \in[-1, 1]$$

If $q \leq -1/4$ and $d' = -1$ then

$$\underbrace{\underbrace{2q}_{\leq -1/2} + \underbrace{2b}_{\leq 1/2}}_{\leq 0} - \underbrace{d'}_{-1} \leq 1$$

and

$$\underbrace{2(q+b)}_{\geq -2} - \underbrace{d'}_{-1} \geq -1$$

So if we choose

$$d' = \begin{cases} -1, & \text{if } q < -1/4; \\ 0, & \text{if } -1/4 \leq q \leq 1/4; \\ 1, & \text{if } 1/4 < q \end{cases}$$

then $2(u+a) - d' \in [-1, 1]$.

If we choose $d(u, a, s^a) = d'$, $\text{next}(u, a, s^a) = (2u + a_0 - d', a', \text{tail}(s^a))$ then we get $\text{next}(u, a, s^a) \in A$,

$$\begin{aligned} f(\text{next}(u, a, s^a)) &= f(2u + a_0 - d', a', \text{tail}(s^a)) \\ &= \text{addR}(2u + a_0 - d', a') \\ &= 2 * \text{addR}(u, a) - d' \\ &= 2 * f(u, a, s^a) - d(u, a, s^a) \end{aligned}$$

and by the Theorem 6.1.7 the assertion follows.

From s^a we can compute d' , $2u + a_0 - d'$ and have $(a', \text{tail}(s^a)) \in \sim\mathbb{R}^+$. Therefore, we can by Theorem 6.1.11 compute the stream s for $\text{addR}(u, a, s^a)$ from u and a stream for s^a . \square

6.5.3 Function mp

Lemma 6.5.4. *For any $r \in \sim\mathbb{R}$ s.t. $4r \in [-1, 1]$ we have $4r \in \sim\mathbb{R}$. Furthermore, if $(r, s^r) \in \sim\mathbb{R}^+$ we can compute from s^r a stream s s.t. $(4r, s) \in \sim\mathbb{R}^+$.*

Proof. Let $r \in \sim\mathbb{R}$, so $(r, s^r) \in \sim\mathbb{R}^+$ for some s^r

$$\begin{aligned} r_0 &= \text{head}(s^r), & r' &= 2r - r_0, & \text{then } (r', \text{tail}(s^r)) &\in \sim\mathbb{R}^+ \\ r_1 &= \text{head}(\text{tail}(s^r)), & r'' &= 2r' - r_1, & \text{then } (r'', \text{tail}^2(s^r)) &\in \sim\mathbb{R}^+ \end{aligned}$$

Therefore,

$$r = \frac{r_0 + \frac{r_1 + r''}{2}}{2} = \frac{2r_0 + r_1 + r''}{4}$$

$$2r_0 + r_1 + r'' = 4r \in [-1, 1]$$

Therefore, $4r = \text{addR}(2r_0 + r_1, r'') \in \sim\mathbb{R}$ by Lemma 6.5.3.

From s^r we can compute $r_0, r_1, \text{tail}^2(s^r)$ and have $(r'', \text{tail}^2(s^r)) \in \sim\mathbb{R}^+$. Therefore, we can by Theorem 6.1.11 compute computationally the stream s for $4r$. \square

Theorem 6.5.5. *For $a, b \in \sim\mathbb{R}$ we have $a * b \in \sim\mathbb{R}$. Furthermore, if $(a, s^a), (b, s^b) \in \sim\mathbb{R}^+$ we can compute from s^a, s^b a stream s s.t. $(\text{mp}(a, b), s) \in \sim\mathbb{R}^+$.*

Proof.

$$\text{mp}(a, b) = a * b = 4 * \frac{a * b + 0 + 0}{4} = 4 * \text{mpaux}(a, b, 0, 0) \in \sim\mathbb{R}$$

by Lemma 6.5.1 and 6.5.4 since $a * b \in [-1, 1]$. Furthermore, by the same lammata we can compute a stream for $a * b$. \square

Chapter 7

Signed Digit Representation of Real Numbers in Agda

7.1 SDR

The signed digits consist of 0, 1, -1 and are defined in Agda as follows

```
data Digit : Set where
  (d)0 : Digit
  (d)1 : Digit
  (d)-1 : Digit
```

The infinite sequences, i.e. streams of digits are described as follows

$$a = \{a_0 : a_1 : a_2 : \dots \mid \forall n \in \mathbb{N}. a_n \in \text{Digit}\} : \text{Stream}$$

We can define Stream in Agda by the codata type

```
codata Stream (A : Set) : Set where
  _ :: _ : A → Stream A → Stream A
```

We want to extract functions

$$\text{average} : \text{Stream Digit} \rightarrow \text{Stream Digit} \rightarrow \text{Stream Digit}$$

$$\text{multi} : \text{Stream Digit} \rightarrow \text{Stream Digit} \rightarrow \text{Stream Digit}$$

such that if $r : \mathbb{R}$ has a signed digit representation $0.a_0a_1a_2\cdots$ given by a stream a and $s : \mathbb{R}$ has a signed digit representation $0.b_0b_1b_2\cdots$ given by a stream b , then $r * s : \mathbb{R}$ or $((r + s)/2) : \mathbb{R}$ respectively have a signed digit representation $0.c_0c_1c_2\cdots$ given by a stream $c = \text{multi } a \ b$ or $c = \text{average } a \ b$, respectively. For this purpose we introduce in Agda the set of $\sim\mathbb{R}$ of real numbers having signed digit representation, which corresponds to the mathematical definition of $\sim\mathbb{R}$ in Section 6.1; see also Subsection 7.2 below for the definition in Agda. We prove in Agda the Theorems corresponding to Theorem 6.4.3 and 6.5.5

$$\sim\mathbb{R}\text{average} : (r \ s : \mathbb{R}) \rightarrow \sim\mathbb{R} \ r \rightarrow \sim\mathbb{R} \ s \rightarrow \sim\mathbb{R} \ ((r * s)/2)$$

$$\sim\mathbb{R}\text{multi} : (r \ s : \mathbb{R}) \rightarrow \sim\mathbb{R} \ r \rightarrow \sim\mathbb{R} \ s \rightarrow \sim\mathbb{R} \ (r * s)$$

and we define functions $\sim\mathbb{R}\text{toStream}$, which define for r s.t. $\sim\mathbb{R} \ r$ holds the stream contained in it (StreamtoReal , $\text{Streamto}\sim\mathbb{R}$ is the inverse). We define

$$\sim\mathbb{R}\text{toStream} : (r : \mathbb{R}) \rightarrow \sim\mathbb{R} \ r \rightarrow \text{Stream Digit}$$

$$\text{StreamtoReal} : \text{Stream Digit} \rightarrow \mathbb{R}$$

$$\text{Streamto}\sim\mathbb{R} : (s : \text{Stream Digit}) \rightarrow \sim\mathbb{R} \ (\text{StreamtoReal } s)$$

such that for every $s : \text{Stream Digit}$, and real number r such that there exists $p : \sim\mathbb{R} \ r$ we have $\text{StreamtoReal} \ (\sim\mathbb{R}\text{toStream } r \ p) == r$ (using postulate) and we could prove (left for future work)

$$\sim\mathbb{R}\text{toStream} \ (\text{StreamtoReal } s) \ (\text{Streamto}\sim\mathbb{R} \ s) \approx s$$

where \approx is bisimilarity. We don't define the notion of bisimilarity since it is not relevant for the rest of this thesis. We say s represents r if for some $p : \sim\mathbb{R} \ r$ we

have $\sim\mathbb{R}\text{toStream } r \ p = s$. Then we define

$$\begin{aligned} \text{multi} &: \text{Stream Digit} \rightarrow \text{Stream Digit} \rightarrow \text{Stream Digit} \\ \text{multi } s \ s' &= \sim\mathbb{R}\text{toStream } (\text{StreamtoReal } s * \text{StreamtoReal } s') \\ &(\sim\mathbb{R}\text{multi} \\ &\quad (\text{StreamtoReal } s) \\ &\quad (\text{StreamtoReal } s') \\ &\quad (\text{Streamto}\sim\mathbb{R} \ s) \\ &\quad (\text{Streamto}\sim\mathbb{R} \ s')) \end{aligned}$$

We know that `multi` is correct, namely if s represents r and s' represents r' then $(\text{multi } s \ s')$ represents $r * r'$. The same can be done for average $s \ s'$. Furthermore, for some $r, r' : \mathbb{R} \ p : \sim\mathbb{R} \ r$ and $p' : \sim\mathbb{R} \ r'$ we can compute the signed digits representing $r * r'$, $(r + r')/2$, respectively as

$$\sim\mathbb{R}\text{toStream } (\sim\mathbb{R}\text{multi } r \ r' \ p \ p')$$

or

$$\sim\mathbb{R}\text{toStream } (\sim\mathbb{R}\text{average } r \ r' \ p \ p')$$

$\sim\mathbb{R}$ stream is not printable (because it is infinite) so we finitize it and compute list representations of the first n digits. We define

$$\text{StreamToListn} : \{A : \text{Set}\} \rightarrow \text{Stream } A \rightarrow (n : \mathbb{N}) \rightarrow \text{List } A$$

such that $\text{StreamToListn } l \ n$ are the first n elements of l . For instance,

$$\text{StreamToListn}(\sim\mathbb{R}\text{toStream } (\text{embed}\mathbb{Q}\rightarrow\mathbb{R} \ q29/39) \ \sim\mathbb{R}q29/39) \ (5 \ +1)$$

computes the list of the first 6 digits of $29/39$ where $q29/39$ is the rational number $29/39$, $\sim\mathbb{R}q29/39$ is the proof of signed digits representing $29/39$ (see next section) and $(5 \ +1)$ is natural number 6 ($_ \ +1 : \mathbb{N} \rightarrow \mathbb{N}^+$ is the constructor of the nonzero natural numbers \mathbb{N}^+). Since $29/39 \sim 0.101111\dots$ it returns, as expected it

returns the list

$${}_{(d)}1 :: ({}_{(d)}0 :: ({}_{(d)}1 :: ({}_{(d)}1 :: ({}_{(d)}1 :: ({}_{(d)}1 :: ({}_{(d)}1 :: []))))))$$

We can define the above in Agda as follows

`headS : {A : Set} → Stream A → A`

`headS (y :: y') = y`

`tailS : {A : Set} → Stream A → Stream A`

`tailS (y :: y') = y'`

We use \mathbb{N}^+ instead of \mathbb{N} . `toList s n` gives the list of first n digits of s .

`toList : {A : Set} → Stream A → \mathbb{N}^+ → List A`

`toList (y :: l) (zero +1) = y :: []`

`toList (y :: l) (suc y1 +1) = y :: toList l (y1 +1)`

We convert now streams of digits into a real number. This is done by taking for a stream $(0.a_0 \cdots a_{n-1})_{n \in \mathbb{N}^+}$ which is a Cauchy sequence. By completeness of the real number it has a limit point r . Let $g [a_0, \cdots, a_{n-1}]$ be the rational number $0.a_0 \cdots a_{n-1}$. Then $g (y :: y') = \frac{y + g y'}{2} : \mathbb{Q}$.

`g : List Digit → \mathbb{Q}`

`g [] = 0% ' +1`

`g (y :: y') = ((embedD→ \mathbb{Z} y $\%$ ' +1) + \mathbb{Q} g y') / \mathbb{Q} ($i+2 \%$ ' +1)`

`h s` is the Cauchy sequence $(a_n)_{n \in \mathbb{N}^+}$ where $a_n = 0.s_0 \cdots s_{n-1}$ if $s = s_0 :: s_1 :: s_2 :: \cdots$. We don't prove that $(h l n)_{n \in \mathbb{N}^+}$ is Cauchy but postulate it. Note: we axiomatize formulas and equations with no computational content even though they would require real proofs.

`h : Stream Digit → \mathbb{N}^+ → \mathbb{R}`

`h l = \n n → embed \mathbb{Q} → \mathbb{R} (g (toList l n))`

postulate

`axiomS : (l : Stream Digit)`

`→ (n m N : \mathbb{N}^+)`

$$\begin{aligned} &\rightarrow n \geq^+ N \\ &\rightarrow m \geq^+ N \\ &\rightarrow | h l n - h l m | < 2^{\wedge} (\text{neg } N) \end{aligned}$$

We obtain the real number represented by a stream $s_0 :: s_1 :: s_2 :: \dots$ as the limit of the Cauchy sequence $(a_n)_{n \in \mathbb{N}}$ as above, which we obtain by the completeness of \mathbb{R} . Note the completeness of \mathbb{R} is represented by the axioms `limitpoint` and `convergence`.

`StreamtoReal : Stream Digit \rightarrow \mathbb{R}`
`StreamtoReal = \l \rightarrow limitpoint (h l) (axiomS l)`

`converge` function shows that $(h l m)_{m \in \mathbb{N}^+}$ converges to its limit point

`converg : (l : Stream Digit)`
 $\rightarrow (m k : \mathbb{N}^+)$
 $\rightarrow m \geq^+ k$
 $\rightarrow | h l m - \text{limitpoint } (h l) \text{ (axiomS } l) | < 2^{\wedge} (\text{neg } k)$

`converg l = convergence (h l) (axiomS l)`

`Streamto \sim \mathbb{R} : (l : Stream Digit) \rightarrow (r : \mathbb{R}) \rightarrow r == StreamtoReal l \rightarrow \sim \mathbb{R} r`

`Streamto \sim \mathbb{R} l r p = cons (headS l) r w q`

where

`postulate`

`w1 : r2 * r - embedD (headS l) == StreamtoReal (tailS l)`

Note that `w1` above would really need a proof, since it has no computational content we postulate it.

`pl : - r1 \leq r — easy`

`pr : r \leq r1 — easy`

`q : r \in [-1, 1]`

`q = and pl pr`

```
w : ~ℝ (r2 * r - embedD (headS l))
w = Streamto~ℝ (tailS l) (r2 * r - embedD (headS l)) w1
```

Here for $w1$, pl and pr we use $p : r == \text{StreamtoReal } l$ therefore $r \in [-1, 1]$ and $r2 * r - \text{embedD } (\text{headS } l) == \text{StreamtoReal } (\text{tailS } l)$.

```
~ℝtoStream : (r : ℝ) → ~ℝ r → Stream Digit
~ℝtoStream r (cons d .r y y') = d :: ~ℝtoStream (r2 * r - embedD d) y
```

We give the operation which from two streams computes the stream representing its product. It uses $\sim\text{mp}$ which is given in Section 7.4.4

```
multi : Stream Digit → Stream Digit → Stream Digit
multi l s = ~ℝtoStream (StreamtoReal l * StreamtoReal s)
  (~mp (StreamtoReal l)
    (StreamtoReal s)
    (StreamtoReal l * StreamtoReal s)
    (Streamto~ℝ l (StreamtoReal l) (refl== (StreamtoReal l)))
    (Streamto~ℝ s (StreamtoReal s) (refl== (StreamtoReal s)))
    (refl== (StreamtoReal l * StreamtoReal s)))
```

Similarily average computes from two streams computes the stream representing its average. It uses $\sim\text{av}$ which is given in Section 7.3

```
average : Stream Digit → Stream Digit → Stream Digit
average l s = ~ℝtoStream ((StreamtoReal l + StreamtoReal s) * r1/2)
  (~av (StreamtoReal l)
    (StreamtoReal s)
    ((StreamtoReal l + StreamtoReal s) * r1/2)
    (Streamto~ℝ l (StreamtoReal l) (refl== (StreamtoReal l)))
    (Streamto~ℝ s (StreamtoReal s) (refl== (StreamtoReal s)))
    (refl== (StreamtoReal l + StreamtoReal s) * r1/2))
```

postulate

```
axiomS= : (r : ℝ) → (p : ~ℝ r) → StreamtoReal (~ℝtoStream r p) == r
```

We postulate that if we form from $r : \mathbb{R}$ s.t. $\sim\mathbb{R}$ holds a stream and convert it back into a real number, we obtain r . Note that: we add an axiom `axiomS=`

which has no computational content. This axiom would require a complicated proof but is as an axiom acceptable, i.e. fulfils our conditions. This was not done because of lack of time. It's easy to prove informally that this axiom holds. With this axiom we get

$$\text{StreamtoReal} (\text{multi } s, s') = \text{StreamtoReal } s * \text{StreamtoReal } s'$$

7.2 SDR as Codata Type $\sim\mathbb{R}$

We have seen in Section 6.1 that a real number r has a signed digit representation (SDR). In this chapter $\sim\mathbb{R}$ will denote the Agda representation of real number with SDR. $\sim\mathbb{R}$ is defined in Agda as follows

`embedD : Digit → ℝ`

`embedD (d)0 = r0`

`embedD (d)1 = r1`

`embedD (d)-1 = - r1`

`_-∈[-1, 1] : ℝ → Set`

`_-∈[-1, 1] = -r1 ≤ r ∧ r ≤ r1`

`codata ~ℝ : ℝ → Set` where

`cons : (d : Digit)(r : ℝ)`

`→ ~ℝ (r2 * r - embedD d)`

`→ r ∈[-1, 1]`

`→ ~ℝ r`

$\sim\mathbb{R}$ is the largest set such that for all $r \in \sim\mathbb{R}$ we have $r \in [-1, 1]$ and there exists a (first) digit a_0 s.t. $2r - a_0 \in \sim\mathbb{R}$. So $\sim\mathbb{R}$ is just the representation of $\sim\mathbb{R}_{math}$ (in this Chapter we write $\sim\mathbb{R}_{math}$ for $\sim\mathbb{R}$ as in the Definition 6.1.3) in Agda.

$\sim\mathbb{R}$ contains a stream if $p : \sim\mathbb{R} r$ then p has the form

$$\begin{aligned} p &= \text{cons } d_0 r_0 p_0 q_0 \\ &= \text{cons } (d_0 r_0 (\text{cons } d_1 r_1 (\text{cons } d_2 r_2 (\text{cons } \dots) p_2 q_2) p_1 q_1) p_0 q_0) \end{aligned}$$

where

$$\begin{aligned} &d_i \text{ are digits} \\ &r = r_0 \\ &r1 = 2r_0 - d_0 \\ &r2 = 2r_1 - d_1 \\ &\quad \vdots \\ &q_i : r_i \in [-1, 1] \end{aligned}$$

It contains the stream $d_0 d_1 d_2 d_3 \dots$ which we obtain by defining

$$\sim\mathbb{R}\text{toStream } r (\text{cons } d .r p q) = d :: \sim\mathbb{R}\text{toStream } (r2 * r - \text{embedD } d) p$$

which is an element of Stream by coinduction. $p : \sim\mathbb{R} r$ is a proof that r has a signed digit representation and we call it a proof that p has an SDR. We introduce a notation \sim as follows

$$r \sim 0.a_0 a_1 a_2 a_3 \dots$$

if and only if there exists $p : \sim\mathbb{R} r$ such that the digit stream contained in p is

$$a_0 :: a_1 :: a_2 :: a_3 :: \dots$$

7.2.1 Proof of $\sim\mathbb{R} (-^r 1)$, $\sim\mathbb{R} ^r 0$, $\sim\mathbb{R} ^r 1$

We will define $\sim\mathbb{R}0 : \sim\mathbb{R} ^r 0$, $\sim\mathbb{R}1 : ^r 1$ and $\sim\mathbb{R}-1 : \sim\mathbb{R} (-^r 1)$. We start proving $\sim\mathbb{R} ^r 1$ and $\sim\mathbb{R} (-^r 1)$ first. We have seen in Section 6.2 that $1 \sim 0.11111\dots$. We define $\sim\mathbb{R}1$ in Agda as follows

$$\begin{aligned} \sim\mathbb{R}1 &: \sim\mathbb{R} ^r 1 \\ \sim\mathbb{R}1 &= \sim\mathbb{R}1\text{aux } ^r 1 (\text{refl} == ^r 1) \end{aligned}$$

where

$$\begin{aligned} p1 : (r : \mathbb{R}) &\rightarrow r == ^r 1 \rightarrow r2 * r - \text{embedD } (d) 1 == ^r 1 \\ p1 r p &= \dots \end{aligned}$$

$$p2 : (r : \mathbb{R}) \rightarrow r == \text{r1} \rightarrow r \in [-1, 1]$$

$$p2 \ r \ p = \dots$$

$$\sim\mathbb{R}1\text{aux} : (r : \mathbb{R}) \rightarrow r == \text{r1} \rightarrow \sim\mathbb{R} \ r$$

$$\sim\mathbb{R}1\text{aux} \ r \ p = \text{cons}_{(d)} 1 \ r \ (\sim\mathbb{R}1\text{aux} \ (\text{r2} * r - \text{embedD}_{(d)} 1) \ (p1 \ r \ p))$$

$$(p2 \ r \ p)$$

(again, instead of showing full code here we use \dots to denote the rest of code) which is a proof of signed digit stream of real number 1. It shows that $\sim\mathbb{R} \ \text{r1}$ holds. For technical reasons we define $\sim\mathbb{R}1\text{aux}$ as a transfer $\sim\mathbb{R}$ lemma which states that $\forall r : \mathbb{R}. r == \text{r1} \rightarrow \sim\mathbb{R} \ \text{r1}$ holds, since otherwise guarded recursion doesn't hold. The reason is that we define $\sim\mathbb{R}1\text{aux}$ corecursively and refer to a proof that $\sim\mathbb{R}1\text{aux} \ (\text{r2} * r - \text{embedD}_{(d)} 1) \ p$ holds which we get from the fact that $p : \text{r2} * r - \text{embedD}_{(d)} 1 == \text{r1}$. A naive definition of $\sim\mathbb{R} \ \text{r1}$ would be

$$\sim\mathbb{R}1' : \sim\mathbb{R} \ \text{r1}$$

$$\sim\mathbb{R}1' = \text{cons}_{(d)} 1 \ \text{r1} \ p \ p2$$

where

$$p1 : \text{r1} == \text{r2} * \text{r1} - \text{embedD}_{(d)} 1$$

$$p1 = \dots$$

$$p2 : \text{r1} \in [-1, 1]$$

$$p2 = \dots$$

$$p : \sim\mathbb{R} \ (\text{r2} * \text{r1} - \text{embedD}_{(d)} 1)$$

$$p = \text{transfer} == (\lambda z \rightarrow \sim\mathbb{R} \ z) \ \text{r1} \ (\text{r2} * \text{r1} - \text{embedD}_{(d)} 1) \ p1 \ \sim\mathbb{R}1'$$

Here, the use of transfer $==$ shows that $p : \sim\mathbb{R} \ \text{r1}$ and $q : (\text{r2} * \text{r1} - \text{embedD}_{(d)} 1) == \text{r1}$ implies $p : \sim\mathbb{R} \ (\text{r2} * \text{r1} - \text{embedD}_{(d)} 1)$. This is not an example of guarded recursion since we apply an elimination rule (namely transfer) to the corecursive call. Instead, we define a lemma which has as result the result of the transfer $==$ function applied to a proof of $\sim\mathbb{R} \ 1'$, i.e.

$$\sim\mathbb{R}1\text{aux} : (r : \mathbb{R}) \rightarrow r == \text{r1} \rightarrow \sim\mathbb{R} \ r$$

Now we can refer on the recursive call to

$${}^r2 * r - \text{embedD}_{(d)} 1$$

and a proof that, if $r == {}^r1$, then ${}^r2 * r - \text{embedD}_{(d)} 1 == {}^r1$. Then we get $\sim\mathbb{R}1 = \sim\mathbb{R}1\text{aux } {}^r1$ ($\text{refl} == {}^r1$).

In Section 6.2 we have seen $-1 \sim 0.(-1)(-1)(-1)(-1)(-1)\dots$. We define $\sim\mathbb{R}-1$ in Agda as follows

$$\sim\mathbb{R}-1 : \sim\mathbb{R} (- {}^r1)$$

$$\sim\mathbb{R}1 = \sim\mathbb{R}-1\text{aux } (- {}^r1) (\text{refl} == (- {}^r1))$$

where

$$p1 : (r : \mathbb{R}) \rightarrow r == - {}^r1 \rightarrow {}^r2 * r - \text{embedD}_{(d)} -1 == - {}^r1$$

$$p1 \ r \ p = \dots$$

$$p2 : (r : \mathbb{R}) \rightarrow r == - {}^r1 \rightarrow r \in [-1, 1]$$

$$p2 \ r \ p = \dots$$

$$\sim\mathbb{R}-1\text{aux} : (r : \mathbb{R}) \rightarrow r == - {}^r1 \rightarrow \sim\mathbb{R} r$$

$$\sim\mathbb{R}-1\text{aux } r \ p = \text{cons}_{(d)} -1 \ r \ (\sim\mathbb{R}1\text{aux } ({}^r2 * r - \text{embedD}_{(d)} -1) \ (p1 \ r \ p)) \\ (p2 \ r \ p)$$

Here, $\sim\mathbb{R}-1\text{aux}$ is defined as the transferred $\sim\mathbb{R}(-1)$ lemma again.

We have seen in Section 6.2 that there are at least 3 signed digit representations of 0 (in fact there are more), namely $0 \sim 0.00000\dots$, $0 \sim 0.1(-1)(-1)(-1)\dots$ and $0 \sim 0.(-1)111\dots$. We first define $\sim\mathbb{R}0 : \sim\mathbb{R} {}^r0$, which starts with first digit 0 (we get $0 \sim 0.00000\dots$) as follows

$$\sim\mathbb{R}0 : \sim\mathbb{R} {}^r0$$

$$\sim\mathbb{R}0 = \sim\mathbb{R}0\text{aux } {}^r0 (\text{refl} == {}^r0)$$

where

$$p1 : ((r : \mathbb{R})) \rightarrow r == {}^r0 \rightarrow {}^r2 * r - \text{embedD}_{(d)} 0 == {}^r0$$

$$p1 \ r \ p = \dots$$

$$p2 : (r : \mathbb{R}) \rightarrow r == {}^r0 \rightarrow r \in [-1, 1]$$

$$p2\ r\ p = \dots$$

$$\sim\mathbb{R}0_{\text{aux}} : (r : \mathbb{R}) \rightarrow r == {}^r0 \rightarrow \sim\mathbb{R}\ r$$

$$\sim\mathbb{R}0_{\text{aux}}\ r\ p = \text{cons}_{(d)}\ 0\ r\ (\sim\mathbb{R}1_{\text{aux}}\ ({}^r2 * x - \text{embedD}_{(d)}\ 0)\ (p1\ r\ p)) \\ (p2\ r\ p)$$

Next, we define $\sim\mathbb{R}0_1 : \sim\mathbb{R}\ {}^r0$ which starts with first digit 1, (note that $2 * 0 - 1 = -1$ so we get $0 \sim 0.1a_0a_1a_2 \dots$. In fact $a_i = -1$ so $0 \sim 0.1(-1)(-1) \dots$) as follows. We will use the proof $\sim\mathbb{R}-1 : \sim\mathbb{R}\ (-{}^r1)$ to show $\sim\mathbb{R}\ ({}^r2 * {}^r0 - {}^r1)$

$$\sim\mathbb{R}0_1 : \sim\mathbb{R}\ {}^r0$$

$$\sim\mathbb{R}0_1 = \text{cons}_{(d)}\ 1\ p\ (p2\ {}^r0\ (\text{refl} == {}^r0))$$

where

$$p1 : ((r : \mathbb{R})) \rightarrow r == {}^r0 \rightarrow {}^r2 * r - \text{embedD}_{(d)}\ 1 == -{}^r1$$

$$p1\ r\ p = \dots$$

$$p2 : (r : \mathbb{R}) \rightarrow r == {}^r0 \rightarrow r \in [-1, 1]$$

$$p2\ r\ p = \dots$$

$$p : \sim\mathbb{R}\ ({}^r2 * {}^r0 - {}^r1)$$

$$p = \text{transfer}\ \sim\mathbb{R}\ (-{}^r1)\ \sim\mathbb{R}-1\ ({}^r2 * {}^r0 - {}^r1)\ (p1\ {}^r0\ (\text{refl} == {}^r0))$$

where we use the lemma $\text{transfer}\ \sim\mathbb{R}$ which is defined in next section.

We define $\sim\mathbb{R}0_{-1} : \sim\mathbb{R}\ {}^r0$ which starts with first digit -1 (we get $0 \sim 0.(-1)a_0a_1a_2 \dots$ where $1 \sim 0.a_0a_1 \dots$, so $0 \sim 0.(-1)111 \dots$) as follows. We will use the proof $\sim\mathbb{R}-1 : \sim\mathbb{R}\ (-{}^r1)$ to show $\sim\mathbb{R}\ ({}^r2 * {}^r0 - (-{}^r1))$

$$\sim\mathbb{R}0_{-1} : \sim\mathbb{R}\ {}^r0$$

$$\sim\mathbb{R}0_{-1} = \text{cons}_{(d)}\ -1\ p\ (p2\ {}^r0\ (\text{refl} == {}^r0))$$

where

$$p1 : ((r : \mathbb{R})) \rightarrow r == {}^r0 \rightarrow {}^r2 * r - \text{embedD}_{(d)}\ -1 == {}^r1$$

$$p1\ r\ p = \dots$$

$$p2 : (r : \mathbb{R}) \rightarrow r == {}^r0 \rightarrow r \in [-1, 1]$$

$$p2\ r\ p = \dots$$

```

p : ~ℝ (r2 * r0 - (- r1))
p = transfer~ℝ r1 ~ℝ1 (r2 * r0 - (- r1)) (p1 r0 (refl==r0))

```

7.2.2 The Function transfer~ℝ

We need to define transfer~ℝ which proves that if ~ℝ r , $r == s$ then we get ~ℝ s , since we cannot just use the function transfer== by case distinction on $s == r$ since this would violate our restrictions on the Agda code (see Chapter 5 where one of the conditions is that conditions on equalities need to have as result types equalities or postulated types). In fact, if we simply use the function transfer== to get ~ℝ s , the extracted program doesn't normalise. First we need to define an extraction function ~ℝ'aux which states that if we have a proof of ~ℝ r , then we get its first digit d and a proof of ~ℝ ($r2 * r - \text{embedD } d$). We define first the product of a set A and a set B depending on A together with projections π^l and π^r . Now ~ℝ'aux is defined as follows

```

data _ ×(d) _ (A : Set)(B : A → Set) : Set where

```

```

  π : (a : A) → B a → A ×(d) B

```

```

πl : {A : Set}{B : A → Set} → A ×(d) B → A

```

```

πl (π(d) y) =(d)

```

```

πr : {A : Set}{B : A → Set} → (x : A ×(d) B) → B (πl x)

```

```

πr (π(d) y) = y

```

```

~ℝ'aux : (r : ℝ) → ~ℝ r → Digit ×(d) (\d → ~ℝ (r2 * r - embedD d))

```

```

~ℝ'aux r (cons d .r y y') = π d y

```

~ℝ'aux $r p = \pi d y$ where d is the head of the stream contained in p and y is a proof of ~ℝ ($r2 * r - \text{embedD } d$). Then, if we have ~ℝ r , we can retrieve head and tail of ~ℝ r by the functions head and tail which are defined as follows

head : (r : ℝ) → ~ℝ r → Digit

head r Rr = π^l (~ℝ'aux r Rr)

tail : (r : ℝ) → ~ℝ r → ℝ

tail r Rr = r2 * r - embedD (head r Rr)

~tail : (r : ℝ) → (Rr : ~ℝ r) → ~ℝ (tail r Rr)

~tail r Rr = π^r (~ℝ'aux r Rr)

Here, tail r p is the real number r2 * r - embedD (head r p) and ~tail r p is the proof of ~ℝ (tail r p), i.e. a proof of ~ℝ (r2 * r - embedD (head r p)).

Now we can define the function transfer~ℝ as follows by guarded recursion on ~ℝ r

transfer~ℝ : (r : ℝ) → ~ℝ r → (s : ℝ) → s == r → ~ℝ s

transfer~ℝ r Rr s p = cons δ s ~ℝs' pp

where

δ : Digit

δ = head r Rr

pp : s ∈ [-1, 1]

pp = ...

p2 : r2 * s - embedD δ == r2 * r - embedD δ

p2 = ...

~ℝs' : ~ℝ (r2 * s - embedD δ)

~ℝs' = transfer~ℝ (tail r Rr) (~tail r Rr) (r2 * s - embedD δ) p2

7.2.3 The Function ~ℝembedQ

(This section corresponds to Section 6.3.)

We have shown how to define ~ℝ r0, ~ℝ r1 and ~ℝ (- r1). Now we prove

$\sim\mathbb{R} r$ for all rational numbers in the interval $[-1,1]$. This is done by the function $\sim\mathbb{R}\text{embed}\mathbb{Q}$.

Before we introduce the $\sim\mathbb{R}\text{embed}\mathbb{Q}$ function we introduce a function $2q-d \in [-1,1] \text{--Prod}$ which determines for $q : \mathbb{Q}$ s.t. $q \in [-1,1]$ a pair $(\pi \delta p)$ s.t. $\delta : \text{Digit}$ and $p : 2q - \delta \in [-1,1]$. The function $2q-d \in [-1,1] \text{--Prod}$ determines for the embedding of the rational number q into \mathbb{R} which is in the interval $[-1,1]$ a digit d such that $2q - d \in [-1,1]$. In Section 6.3 we have seen that we can choose as first digit

$$\begin{cases} -1, & \text{if } q \in [-1, 0] \\ 0, & \text{if } q \in [-1/2, 1/2] \\ 1, & \text{if } q \in [0, 1] \end{cases}$$

and get then $2q - d \in [-1,1]$. The function $2q-d \in [-1,1] \text{--Prod}$ is defined in Agda as follows

```
embedD→ℤ : Digit → ℤ
embedD→ℤ (d)0 = i0
embedD→ℤ (d)1 = i+1
embedD→ℤ (d)-1 = i-1
```

$q \in [-1,0] \vee q \in [-1/2, 1/2] \vee q \in [0,1]$ function will check whether $q \geq 0$ or $q < 0$:
 if $q \geq 0$ it will return a proof of $\text{embed}\mathbb{Q} \rightarrow \mathbb{R} q \in [0,1]$,
 if $q < 0$ it will return a proof of $\text{embed}\mathbb{Q} \rightarrow \mathbb{R} q \in [-1,0]$.

```
q ∈ [-1,0] ∨ q ∈ [-1/2, 1/2] ∨ q ∈ [0,1] : (q : ℚ)
  → embedℚ→ℝ q ∈ [-1,1]
  → (embedℚ→ℝ q ∈ [-1,0] ∨
     embedℚ→ℝ q ∈ [-1/2, 1/2]) ∨
     embedℚ→ℝ q ∈ [0,1]
```

$q \in [-1,0] \vee q \in [-1/2, 1/2] \vee q \in [0,1]$ $q p = \dots$

Note that case $q \in [-1/2, 1/2]$ is not chosen. (In order to determine functions which might behave better this choice could be modified)

$2q-d \in [-1, 1]$ -Prod : $(q : \mathbb{Q})$
 $\rightarrow \text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q \in [-1, 1]$
 $\rightarrow \text{Digit} \times_{(d)} (\backslash d$
 $\rightarrow (\text{r2} * \text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q - \text{embedD} \ d) \in [-1, 1])$
 $2q-d \in [-1, 1]$ -Prod $q \ p = \text{aux} \ q \ (q \in [-1, 0] \vee q \in [-1/2, 1/2] \vee q \in [0, 1] \ q \ p)$

where

$\text{aux} : (q : \mathbb{Q})$
 $\rightarrow (\text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q \in [-1, 0] \vee$
 $\text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q \in [-1/2, 1/2]) \vee$
 $\text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q \in [0, 1]$
 $\rightarrow \text{Digit} \times_{(d)} (\backslash d$
 $\rightarrow (\text{r2} * \text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q - \text{embedD} \ d) \in [-1, 1])$

$\text{aux} \ q \ (\text{inl} \ (\text{inl} \ ll)) = \pi_{(d)} - 1$ (and $v10 \ v11$)

where

$v10 : -\text{r1} \leq (\text{r2} \text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q - \text{embedD} \ (d) - 1)$
 $v10 = \dots$

$v11 : (\text{r2} \text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q - \text{embedD} \ (d) - 1) \leq \text{r1}$
 $v11 = \dots$

$\text{aux} \ q \ (\text{inl} \ (\text{inr} \ lr)) = \pi_{(d)} 0$ (and $v12 \ v13$)

where

$v12 : -\text{r1} \leq (\text{r2} \text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q - \text{embedD} \ (d) 0)$
 $v12 = \dots$

$v13 : (\text{r2} \text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q - \text{embedD} \ (d) 0) \leq \text{r1}$
 $v13 = \dots$

$\text{aux} \ q \ (\text{inr} \ rr) = \pi_{(d)} 1$ (and $v14 \ v15$)

where

$v14 : -\text{r1} \leq (\text{r2} \text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q - \text{embedD} \ (d) 1)$
 $v14 = \dots$

$v15 : (\text{r2} \text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ q - \text{embedD} \ (d) 1) \leq \text{r1}$

$v15 = \dots$

Now we can define the embedding $\sim\mathbb{R}\text{embed}\mathbb{Q}$ of $\mathbb{Q} \cap [-1, 1]$ into $\sim\mathbb{R}$ in Agda as follows

```

 $\sim\mathbb{R}\text{embed}\mathbb{Q} : (q : \mathbb{Q})$ 
   $\rightarrow (r : \mathbb{R})$ 
   $\rightarrow \text{embed}\mathbb{Q}\rightarrow\mathbb{R} q \in [-1, 1]$ 
   $\rightarrow r == \text{embed}\mathbb{Q}\rightarrow\mathbb{R} q$ 
   $\rightarrow \sim\mathbb{R} r$ 

```

```

 $\sim\mathbb{R}\text{embed}\mathbb{Q} q r p p' = \text{cons } \delta r 1h pr$ 

```

where

```

 $\delta : \text{Digit}$ 
 $\delta = \pi^l (2q - d \in [-1, 1] - \text{Prod } q p)$ 

```

```

 $pr : r \in [-1, 1]$ 

```

```

 $pr = \dots$ 

```

```

 $p2 : \text{embed}\mathbb{Q}\rightarrow\mathbb{R} (\text{pos } +2 \% ' +1 * \mathbb{Q} q - \mathbb{Q} \text{embedD}\rightarrow\mathbb{Z} \delta \% ' +1)$ 
   $\in [-1, 1]$ 

```

```

 $p2 = \dots$ 

```

```

 $w3 : \text{!}2 * r - \text{embedD } \delta ==$ 
   $\text{embed}\mathbb{Q}\rightarrow\mathbb{R} (\text{pos } +2 \% ' +1 * \mathbb{Q} q - \mathbb{Q} \text{embedD}\rightarrow\mathbb{Z} \delta \% ' +1)$ 

```

```

 $w3 = \dots$ 

```

```

 $1h : \sim\mathbb{R} (\text{!}2 * r - \text{embedD } \delta)$ 

```

```

 $1h = \sim\mathbb{R}\text{embed}\mathbb{Q} (\text{pos } +2 \% ' +1 * \mathbb{Q} q - \mathbb{Q} \text{embedD}\rightarrow\mathbb{Z} \delta \% ' +1)$ 
   $(\text{!}2 * r - \text{embedD } \delta) p2 w3$ 

```

where pr is the proof r is in the interval $[-1, 1]$ which follows from $r == \text{embed}\mathbb{Q}\rightarrow\mathbb{R} q$ and $\text{embed}\mathbb{Q}\rightarrow\mathbb{R} q \in [-1, 1]$. We can get $p2$ by $\pi^r (2q - d \in [-1, 1] - \text{Prod } q p)$ which shows that $2q - \delta \in [-1, 1]$. $w3$ is just equational reasoning.

7.2.4 Examples of Function $\sim\mathbb{R}\text{embed}\mathbb{Q}$

With the embedding function $\sim\mathbb{R}\text{embed}\mathbb{Q}$ we can define

$$\sim\mathbb{R}q_{1/2} : \sim\mathbb{R} (\text{embed}\mathbb{Q} \rightarrow \mathbb{R} \text{ } q_{1/2})$$

where $q_{1/2}$ is the rational number $1/2$ and

$$\sim\mathbb{R}q_{2/3} : \sim\mathbb{R} (\text{embed}\mathbb{Q} \rightarrow \mathbb{R} \text{ } q_{2/3})$$

where $q_{2/3}$ is the rational number $2/3$ and

$$\sim\mathbb{R}q_{-2/3} : \sim\mathbb{R} (\text{embed}\mathbb{Q} \rightarrow \mathbb{R} \text{ } q_{-2/3})$$

where $q_{-2/3}$ is the rational number $-2/3$, such that

$$q_{1/2} : \mathbb{Q}$$

$$q_{1/2} = \text{1} \text{ } \text{2}^{-1}$$

$$\sim\mathbb{R}q_{1/2} : \sim\mathbb{R} (\text{embed}\mathbb{Q} \rightarrow \mathbb{R} \text{ } q_{1/2})$$

$$\sim\mathbb{R}q_{1/2} = \sim\mathbb{R}\text{embed}\mathbb{Q} (q_{1/2}) (\text{r1} * \text{r1}/2) w (\text{refl} == (\text{r1} * \text{r1}/2))$$

where

$$w : \text{r1} * \text{r1}/2 \in [-1, 1]$$

$$w = \dots$$

We have $\sim\mathbb{R} (\text{embed}\mathbb{Q} \rightarrow \mathbb{R} \text{ } q_{1/2})$ but for $\sim\mathbb{R} (\text{r1}/2)$ we need to adopt $\sim\mathbb{R}q_{1/2}$ and apply the transfer $\sim\mathbb{R}$ function in order to prove that $\text{r1} * \text{r1}/2 == \text{r1}/2$. Then we get $\sim\mathbb{R}1/2 : \sim\mathbb{R} (\text{r1}/2)$ as follows

$$\text{r1}/2 : \mathbb{R}$$

$$\text{r1}/2 = 2^{-1} (\text{neg} \text{ } +1)$$

$$\sim\mathbb{R}1/2 : \sim\mathbb{R} (\text{r1}/2)$$

$$\sim\mathbb{R}1/2 = \text{transfer}\sim\mathbb{R} (\text{r1} * \text{r1}/2) \sim\mathbb{R}q_{1/2} \text{r1}/2 p$$

where

$$p : \text{r1}/2 == \text{r1} * \text{r1}/2$$

$$p = \text{symm} == (\text{r1} * \text{r1}/2) \text{r1}/2$$

$$\begin{aligned} & (\text{trans} == (^r1 * ^r1/2) (^r1/2 * ^r1) ^r1/2 \\ & \quad (\text{symm} * ^r1 ^r1/2) (\text{axiom} * 1 ^r1/2)) \end{aligned}$$

Similarly we get $q2/3 : \mathbb{Q}$, $\sim\mathbb{R}q2/3 : \sim\mathbb{R}(\text{embed}\mathbb{Q} \rightarrow \mathbb{R} q2/3)$ and $q-2/3 : \mathbb{Q}$, $\sim\mathbb{R}q-2/3 : \sim\mathbb{R}(\text{embed}\mathbb{Q} \rightarrow \mathbb{R} q-2/3)$ and obtain $\sim\mathbb{R}1/3 : \sim\mathbb{R} (^r1/3)$ and $\sim\mathbb{R}-1/3 : \sim\mathbb{R} (- ^r1/3)$.

We have demonstrated how to define $\sim\mathbb{R} r$ for r being the embedding of $q : \mathbb{Q}$ into \mathbb{R} and $r \in [-1, 1]$. In the next section we are going to define operators $f : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$, where f is the average and multiplication functions, and give proofs that for $r s : \mathbb{R}$ s.t. $\sim\mathbb{R} r, \sim\mathbb{R} s$ we get $\sim\mathbb{R} (f r s)$.

7.3 $\sim\mathbb{R}$ Is Closed Under the Average Function

av

(This section corresponds to Section 6.4.)

Ideally we would like to show $\sim\mathbb{R}$ is closed under addition, i.e. if $\sim\mathbb{R} r$ and $\sim\mathbb{R} s$ then $\sim\mathbb{R} (r + s)$. However, this is only possible if $r + s$ is in the interval $[-1, 1]$. So instead, we define closure of $\sim\mathbb{R}$ under the average function

$$\begin{aligned} \text{av} & : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \\ \text{av } r \ s & = (r + s) * ^r1/2 \end{aligned}$$

and note that the interval $[-1, 1]$ is closed under av . So we prove for $p : \sim\mathbb{R} r$, $q : \sim\mathbb{R} s$

$$\sim\text{av } r \ s \ p \ q : \sim\mathbb{R} (\text{av } r \ s)$$

As usual in order to deal with the problem of equality result type we define:

$$\sim\text{av} : (r \ s \ t : \mathbb{R}) \rightarrow \sim\mathbb{R} r \rightarrow \sim\mathbb{R} s \rightarrow t == (r + s) * ^r1/2 \rightarrow \sim\mathbb{R} t$$

We will introduce the function $\text{avaux} : \mathbb{R} \rightarrow \mathbb{R} \rightarrow i \in \{-2, -1, 0, 1, 2\} \rightarrow \mathbb{R}$ such that

$$\text{avaux } r' \ s' \ i = \frac{r' + s' + i}{4}$$

We introduce a new data type `Digit2` for i such that $i : \text{Digit2}$ as follows

```
data Digit2 : Set where
  (d)2 0 : Digit2
  (d)2 1 : Digit2
  (d)2 2 : Digit2
  (d)2 -1 : Digit2
  (d)2 -2 : Digit2
```

So, if we have

$$\sim\text{avaux} : (r' s' : \mathbb{R}) \rightarrow \sim\mathbb{R} r' \rightarrow \sim\mathbb{R} s' \rightarrow (i : \text{Digit2}) \rightarrow \sim\mathbb{R} (\text{avaux } r' s' i)$$

for some r', s', p', q', i then, as in Section 6.4, we can define $\sim\text{av } r s p q = \sim\text{avaux } r' s' p' q' i$. $\sim\text{avaux}$ can be defined by applying the same technique (guarded recursion) as when defining $\text{transfer}\sim\mathbb{R}$.

7.3.1 $\sim\mathbb{R}$ Is Closed Under the Function `avaux`

(This section corresponds to Section 6.4.1)

We want to compute $\text{avaux } r s i = (r + s + i)/4$ and its digit stream for some $a, b : \mathbb{R}$, $p : \sim\mathbb{R} a$ and $q : \sim\mathbb{R} b$. We introduce the function `avaux` for computing $(a + b + i)/4$ as follows

```
embedD2 : Digit2 → ℝ
embedD2 (d)2 0 = r0
embedD2 (d)2 1 = r1
embedD2 (d)2 -1 = - r1
embedD2 (d)2 2 = r2
embedD2 (d)2 -2 = - r2
```

```
avaux : ℝ → ℝ → Digit2 → ℝ
avaux r s i = (r + s + (embedD2 i)) * r1/4
```

```
avaux ∈ [-1, 1] : (r s : ℝ)(i : Digit2)
```

$$\begin{aligned}
 &\rightarrow r \in [-1, 1] \\
 &\rightarrow s \in [-1, 1] \\
 &\rightarrow (\text{embedD2 } i) \in [-2, 2] \\
 &\rightarrow (\text{avaux } r \ s \ i) \in [-1, 1]
 \end{aligned}$$

$\text{avaux} \in [-1, 1] \ r \ s \ i$ (and $rl \ rr$) (and $sl \ sr$) (and $il \ ir$) = and $v \ v'$

where

$$v : - \text{r}1 \leq \text{avaux } r \ s \ i$$

$$v = \dots$$

$$v' : \text{avaux } r \ s \ i \leq \text{r}1$$

$$v' = \dots$$

Clearly i is in the interval $[-2, 2]$ and we know that a, b are in the interval $[-1, 1]$. Therefore, $(\text{avaux } a \ b \ i)$ is in the interval $[-1, 1]$ so we get for $r' = (\text{avaux } a \ b \ i)$ a proof $q : r' \in [-1, 1]$. Our next goal is to prove that $\sim\mathbb{R}$ is closed under the avaux function i.e. to find the first digit f and a proof $p : \sim\mathbb{R} (2r' - f)$ s.t. for $r' = \text{avaux } a \ b \ i$ we have

$$\text{cons } f \ r' \ p \ q : \sim\mathbb{R} \ r'$$

In Section 6.4.1 we denoted the first digit by d' . We use here f instead in order to be in accordance with our Agda code written. We give the following proof:

$$\begin{aligned}
 &\sim\text{avaux} : (r \ s \ r' : \mathbb{R})(i : \text{Digit2}) \rightarrow \sim\mathbb{R} \ r \rightarrow \sim\mathbb{R} \ s \rightarrow r' == \text{avaux } r \ s \ i \rightarrow \sim\mathbb{R} \ r' \\
 &\sim\text{avaux } r \ s \ r' \ i \ (\text{cons } d \ .r \ Rr \ pr) \ (\text{cons } e \ .s \ Rs \ ps) \ p = \text{cons } f \ r' \ 1h \ p'
 \end{aligned}$$

where $1h$ is a proof of $\sim\mathbb{R} (2r' - f)$ and p' is a proof of $r' \in [-1, 1]$. So we want to show that $\text{avaux } a \ b \ i = (f + c)/2$ s.t. c is of the form $\text{avaux } (\dots)$. Then we can define $\sim\text{avaux}$ by guarded recursion. In Section 6.4.1 we have seen the following: If a_0, b_0 , are the first digits of a, b

$$\begin{aligned}
 a_0 &= \text{head}(s^a), & a' &= 2a - a_0 \\
 b_0 &= \text{head}(s^b), & b' &= 2b - b_0
 \end{aligned}$$

and $j = a_0 + b_0 + 2i$,

$$f = \begin{cases} 0, & \text{if } -2 \leq j \leq 2; \\ 1, & \text{if } j > 2; \\ -1, & \text{if } j < -2 \end{cases}$$

Then $i' = j - 4f \in [-2, 2]$ and $2 * \text{avaux } (a, b, i) - f = \text{avaux}(a', b', i')$. This proof is represented in Agda as follows

`embedD2→ℤ : Digit2 → ℤ`

`embedD2→ℤ (d)2 0 = i0`

`embedD2→ℤ (d)2 1 = i+1`

`embedD2→ℤ (d)2 2 = i+2`

`embedD2→ℤ (d)2 -1 = i-1`

`embedD2→ℤ (d)2 -2 = i-2`

`embedN+→ℝ : N+ → ℝ`

`embedN+→ℝ (0 +1) = r1`

`embedN+→ℝ (suc y +1) = embedN+→ℝ (y +1) + r1`

`embedℤ→ℝ : ℤ → ℝ`

`embedℤ→ℝ (pos y) = embedN+→ℝ y`

`embedℤ→ℝ (neg y) = - (embedN+→ℝ y)`

`embedℤ→ℝ zero = r0`

`computej : (d e : Digit)(i : Digit2) → ℤ`

`computej d e i = (i+2) *i embedD2→ℤ i +i embedD→ℤ d +i embedD→ℤ e`

`correctnessj : (d e : Digit)`

`→ (i : Digit2)`

`→ (embedℤ→ℝ (computej d e i)) ∈ [-6, 6]`

`correctnessj d e i = and v v'`

where

`v : - r6 ≤ embedℤ→ℝ (computej d e i)`

$v = \dots$

$v' : \text{embed}\mathbb{Z}\rightarrow\mathbb{R} (\text{computej } d \ e \ i) \leq {}^r6$

$v' = \dots$

The function $(\text{computej } d \ e \ i)$ computes the integer $2i + d + e$ and $(\text{correctnessj } d \ e \ i)$ is a proof that $(\text{computej } d \ e \ i)$ is in the interval $[-6, 6]$. Note that: j needs to be an element of \mathbb{Z} instead of \mathbb{R} , otherwise we cannot prove its properties.

$\text{compute}f\text{-aux} : (j : \mathbb{Z})$
 $\rightarrow ((\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) < - {}^r2 \vee (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) \in [-2, 2])$
 $\vee {}^r2 < (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j)$
 $\rightarrow \text{Digit}$

$\text{compute}f\text{-aux } j \ (\text{inl } (\text{inl } ll)) = {}_{(d)}-1$

$\text{compute}f\text{-aux } j \ (\text{inl } (\text{inr } rr)) = {}_{(d)}0$

$\text{compute}f\text{-aux } j \ (\text{inr } r) = {}_{(d)}1$

$j < -n \vee j \in [-n, n] \vee n < j : (j \ n : \mathbb{Z})$
 $\rightarrow ((\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) < - (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ n)$
 $\vee (- (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ n) \leq (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j)$
 $\wedge (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) \leq (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ n)))$
 $\vee ((\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ n) < (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j))$

$j < -n \vee j \in [-n, n] \vee n < j \ j \ n = \dots$

$\text{compute}f : (d \ e : \text{Digit})(i : \text{Digit2}) \rightarrow \text{Digit}$

$\text{compute}f \ d \ e \ i = \text{compute}f\text{-aux}$

$(\text{computej } d \ e \ i)$

$(j < -n \vee j \in [-n, n] \vee n < j \ (\text{computej } d \ e \ i)^{i+2})$

$(\text{compute}f \ d \ e \ i)$ computes a digit which is the first digit f such that if $(\text{computej } d \ e \ i)$ is less than $-{}^r2$ then $f = 1$. If $(\text{computej } d \ e \ i)$ is in the interval $[-2, 2]$ then $f = 0$. Otherwise, $f = 1$. Since we have $i' = j - 4f \in [-2, 2]$ we obtain $i' : \text{Digit}$.

$z \in [-2, 2] \rightarrow D2 : (z : \mathbb{Z}) \rightarrow \text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ z \in [-2, 2] \rightarrow \text{Digit2}$

$$\begin{aligned}
 z \in [-2, 2] &\rightarrow D2 \text{ (pos ((0 +1))) } p = {}_{(d)_2}1 \\
 z \in [-2, 2] &\rightarrow D2 \text{ (pos ((1 +1))) } p = {}_{(d)_2}2 \\
 z \in [-2, 2] &\rightarrow D2 \hat{\text{zero}} p = {}_{(d)_2}0 \\
 z \in [-2, 2] &\rightarrow D2 \text{ (neg ((0 +1))) } p = {}_{(d)_2}-1 \\
 z \in [-2, 2] &\rightarrow D2 \text{ (neg ((1 +1))) } p = {}_{(d)_2}-2 \\
 z \in [-2, 2] &\rightarrow D2 _p = {}_{(d)_2}0
 \end{aligned}$$

$$\begin{aligned}
 \text{compute}'\text{-aux}' : (j : \mathbb{Z}) & \\
 &\rightarrow (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j \in [-6, 6]) \\
 &\rightarrow ((\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) < -\ ^r2 \ \vee \ (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) \in [-2, 2]) \\
 &\quad \vee \ ^r2 < (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) \\
 &\rightarrow \mathbb{Z}
 \end{aligned}$$

$$\text{compute}'\text{-aux}' \ j \ p \ p' = j \text{ -}^i \text{ }^i + 4 \ *^i \ \text{embed}\mathbb{D}\rightarrow\mathbb{Z} \ (\text{compute}\text{f}\text{-aux} \ j \ p')$$

$$\begin{aligned}
 \text{correctnesscompute}'\text{-aux}' : (j : \mathbb{Z}) & \\
 &\rightarrow (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j \in [-6, 6]) \\
 &\rightarrow (q : ((\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) < -\ ^r2 \ \vee \ (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) \in [-2, 2]) \\
 &\quad \vee \ ^r2 < (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j)) \\
 &\rightarrow (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ (\text{compute}'\text{-aux}' \ j \ p \ q)) \in [-2, 2]
 \end{aligned}$$

$$\text{correctnesscompute}'\text{-aux}' \ j \ (\text{and} \ pl \ pr) \ (\text{inl} \ (\text{inl} \ ll)) = \text{and} \ w1 \ v1$$

where

$$\begin{aligned}
 w1 : -\ ^r2 \leq \text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ (j + ^i \text{ }^i + 4) \\
 w1 = \dots
 \end{aligned}$$

$$\begin{aligned}
 v1 : \text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ (j + ^i \text{ }^i + 4) \leq \ ^r2 \\
 v1 = \dots
 \end{aligned}$$

$$\text{correctnesscompute}'\text{-aux}' \ j \ (\text{and} \ pl \ pr) \ (\text{inl} \ (\text{inr} \ (lrl \ lrr))) = \text{and} \ w \ v$$

where

$$\begin{aligned}
 w : -\ ^r2 \leq \text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ (j + ^i \text{ }^i 0) \\
 w = \dots
 \end{aligned}$$

$$\begin{aligned}
 v : \text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ (j + ^i \text{ }^i 0) \leq \ ^r2 \\
 v = \dots
 \end{aligned}$$

correctnesscompute*i'*-aux' *j* (and *pl pr*) (inl (inr *r*) = and *w v*

where

$$w : - \text{r}2 \leq \text{embed}\mathbb{Z}\rightarrow\mathbb{R} (j + \text{i} \text{i} - 4)$$

$$w = \dots$$

$$v : \text{embed}\mathbb{Z}\rightarrow\mathbb{R} (j + \text{i} \text{i} - 4) \leq \text{r}2$$

$$v = \dots$$

(compute*i'*-aux' *j p p'*) is defined as $j - 4f$ and (correctnesscompute*i'*-aux' *j p p'*) is a proof that (compute*i'*-aux' *j p p'*) is in the interval $[-2, 2]$.

compute*i'*-aux : (*j* : \mathbb{Z})

$$\rightarrow (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} j \in [-6, 6])$$

$$\rightarrow ((\text{embed}\mathbb{Z}\rightarrow\mathbb{R} j) < - \text{r}2 \vee (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} j) \in [-2, 2])$$

$$\vee \text{r}2 < (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} j)$$

$$\rightarrow \text{Digit2}$$

compute*i'*-aux *j p p'* = $z \in [-2, 2] \rightarrow D2$ (compute*i'*-aux' *j p p'*)

$$(\text{correctnesscompute}\text{i}'\text{-aux}' j p p')$$

compute*i'* : (*d e* : Digit)(*i* : Digit2) \rightarrow Digit2

compute*i'* *d e i* = compute*i'*-aux

$$(\text{compute}j d e i)$$

$$(\text{correctness}j d e i)$$

$$(j < -n \vee j \in [-n, n] \vee n < j (\text{compute}j d e i)^{\text{i}+2})$$

(compute*i'* *d e i*) computes *i'* which is a type of Digit2. It computes $j = 2i + d + e$ and finds a digit *f* s.t. $i' = j - 4f$ which is in the interval $[-2, 2]$.

Now we introduce the \sim avaux function. One can define it in Agda as follows

\sim avaux : (*r s r'* : \mathbb{R})(*i* : Digit2) $\rightarrow \sim\mathbb{R} r \rightarrow \sim\mathbb{R} s \rightarrow r' == \text{avaux } r s i \rightarrow \sim\mathbb{R} r'$

\sim avaux *r s r' i* (cons *d .r Rr pr*) (cons *e .s Rs ps*) *p* = cons *f r'* 1h *p'*

where

$$f : \text{Digit}$$

$$f = \text{compute}f d e i$$

$$i' : \text{Digit2}$$

$i' = \text{compute! } d \ e \ i$

$q : (\text{r2} * r' - \text{embedD } f) ==$
 $\text{avaux } (\text{r2} * r - \text{embedD } d) (\text{r2} * s - \text{embedD } e) \ i'$
 $q = \dots$

$p' : r' \in [-1, 1]$
 $p' = \dots$

$1h : \sim\mathbb{R} (\text{r2} * r' - \text{embedD } f)$
 $1h = \sim\text{avaux } (\text{r2} * r - \text{embedD } d) (\text{r2} * s - \text{embedD } e)$
 $(\text{r2} * r' - \text{embedD } f) \ i' \ Rr \ Rs \ q$

7.3.2 $\sim\mathbb{R}$ Closure Under av

(This section corresponds to Section 6.4.2)

Now we prove that $\sim\mathbb{R}$ is closed under the average function. If we have $\sim\mathbb{R} \ r$, $\sim\mathbb{R} \ s$ for r and $s : \mathbb{R}$, then we prove $\sim\mathbb{R} (\text{av } r \ s)$ such that

$\sim\text{av} : (r, s, r' : \mathbb{R}) \rightarrow \sim\mathbb{R} \ r \rightarrow \sim\mathbb{R} \ s \rightarrow r' == \text{av } r \ s \rightarrow \sim\mathbb{R} \ r'$
 $\sim\text{av } r \ s \ r' (\text{cons } d \ .r \ Rr' \ pr) (\text{cons } e \ .s \ Rs' \ ps) \ p = w$

where

$g : \mathbb{R}$
 $g = \text{avaux } (\text{r2} * r - \text{embedD } d) (\text{r2} * s - \text{embedD } e) (d +_{(d)} e)$

$w2 : r' == g$
 $w2 = \dots$

$w : \sim\mathbb{R} \ r'$
 $w = \sim\text{avaux } (\text{r2} * r - \text{embedD } d) (\text{r2} * s - \text{embedD } e) \ r' (d +_{(d)} e)$
 $Rr' \ Rs' \ w2$

where $w2$ is obtained by using equality reasoning. We know that $\sim\mathbb{R}$ is closed under avaux , i.e. $\sim\text{avaux } a \ b \ t \ i \ \dots : \sim\mathbb{R} (\text{avaux } a \ b \ i)$. The $\sim\text{av}$ function shows

us that for $a b r' : \mathbb{R}$, $p : r' = \text{av } a b$, and

$$w2 : r' == \text{avaux } (2a' - a_0) (2b' - b_0) (a_0 + b_0)$$

$$\sim \text{avaux } (2a' - a_0) (2b' - b_0) t (a_0 + b_0) \cdots : \sim \mathbb{R} \left(\frac{(2a' - a_0) + (2b' - b_0) + (a_0 + b_0)}{4} \right)$$

such that

$$\text{av } a b = r' = \text{avaux } (2a' - a_0) (2b' - b_0) (a_0 + b_0)$$

$$\begin{aligned} \sim \text{av } a b \cdots &= \sim \text{avaux } (2a' - a_0) (2b' - b_0) t (a_0 + b_0) \cdots : \sim \mathbb{R} (\text{av } a b) = \\ &\sim \mathbb{R} (\text{avaux } (2a' - a_0) (2b' - b_0) (a_0 + b_0)) \end{aligned}$$

7.3.3 Examples of the Average Function

Here are some examples of the average function:

$$\text{av}1+1 : \sim \mathbb{R} ((r1 + r1) * r1/2)$$

$$\text{av}1+1 = \sim \text{av } r1 r1 ((r1 + r1) * r1/2) \sim \mathbb{R}1 \sim \mathbb{R}1 (\text{refl} == ((r1 + r1) * r1/2))$$

$\text{av}1+1$ is a proof of $\sim \mathbb{R} ((r1+r1)*r1/2)$ which is the average of $r1$ and $r1$. Similarly, we obtain

$$\text{av}1-1 : \sim \mathbb{R} ((r1 - r1) * r1/2)$$

and

$$\text{av}q1/3+q1/3 : \sim \mathbb{R} ((\text{embed}\mathbb{Q}\rightarrow\mathbb{R} q1/3 + \text{embed}\mathbb{Q}\rightarrow\mathbb{R} q1/3) * r1/2)$$

7.4 $\sim \mathbb{R}$ Is Closed Under the Multiplication Function mp

(This section corresponds to Section 6.5)

We want to show that if $\sim \mathbb{R} a$ and $\sim \mathbb{R} b$ then $\sim \mathbb{R} (a * b)$. Let $\text{mp } a b = a * b$,

i.e.

$$\begin{aligned} \text{mp} &: \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \\ \text{mp } a \ b &= a * b \end{aligned}$$

As usual, we will show that $\sim\mathbb{R}$ is closed under mp. In order to deal with the problem of equality, we define:

$$\sim\text{mp} : (a \ b \ s : \mathbb{R}) \rightarrow (Ra : \sim\mathbb{R} \ a) \rightarrow (Rb : \sim\mathbb{R} \ b) \rightarrow s == \text{mp } a \ b \rightarrow \sim\mathbb{R} \ s$$

We will introduce the function $\text{mpaux} : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow \text{Digit2} \rightarrow \mathbb{R}$ such that

$$\text{mpaux } a \ b \ q \ i = \frac{a * b + q + i}{4}$$

and show

$$\sim\mathbb{R} \ a \rightarrow \sim\mathbb{R} \ b \rightarrow \sim\mathbb{R} \ q \rightarrow (i : \text{Digit2}) \rightarrow \sim\mathbb{R} \ (\text{mpaux } a \ b \ q \ i)$$

This can be done by guarded recursion (whereas we don't know how to define closure of $\sim\mathbb{R}$ under $*$ by direct guarded recursion). Then we get

$$\text{mp } a \ b = a * b = 4 * \frac{a * b + 0 + 0}{4} = 4 * (\text{mpaux } a \ b \ 0 \ 0)$$

Let $s = a * b$ and $r' = \text{mpaux } a \ b \ 0 \ 0$ then we get

$$s = \text{mp } a \ b = 4 * r'$$

Furthermore, $r' \in [-1/4, 1/4]$. Therefore we show

$$\sim\mathbb{R} \ r' \rightarrow r' \in [-1/4, 1/4] \rightarrow \sim\mathbb{R} \ (4 * r')$$

In order to show this we define two functions appr^2 , tail^2 and show that

$$4 * r' = \text{appr}^2 \ r' \ Rr' + \text{tail}^2 \ r' \ Rr'$$

where Rr' is of type $\sim\mathbb{R} \ r'$. Here $\text{appr}^2 \ r' \ Rr' : \mathbb{Q}$ is the number formed by the first two digits of r' , $\text{tail}^2 \ r' \ Rr'$ is the number formed by the remaining

digits. So if $r' \sim 0.e_0e_1e_2\dots$ (remember this means that the signed digit stream is $e_0 :: e_1 : e_2 :: \dots$) then $\text{appr}^2 r' Rr' = 2e_0 + e_1$, $\text{tail}^2 r' Rr' \sim 0.e_2e_3\dots$. Then $4 * r' == \text{appr}^2 r' Rr' + \text{tail}^2 r' Rr'$ and since we know already $Rr' : \sim\mathbb{R} r'$ we can show $\sim\text{tail}^2 r' Rr' : \sim\mathbb{R} (\text{tail}^2 r' Rr')$.

We show for $u \in \mathbb{Q}$, $a : \mathbb{R}$ s.t. $\sim\mathbb{R} a$ holds that if $u + a \in [-1, 1]$ then $\sim\mathbb{R} (u + a)$ holds and therefore by taking $u = \text{appr}^2 r' Rr'$, $a = \text{tail}^2 r' Rr'$ and $4 * r' = u + a$ we get $\sim\mathbb{R} (u + a)$. So if we have r' s.t. $\sim\mathbb{R} r'$ and $2^2 * r' \in [-1, 1]$ then we can get $\sim\mathbb{R} (2^2 * r')$.

More generally we introduce two functions: the first function $\text{addR } u a = u + a$ such that

$$\begin{aligned} \text{addR} &: \mathbb{Q} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \\ \text{addR } u a &= \text{embed}\mathbb{Q}\rightarrow\mathbb{R} u + a \end{aligned}$$

We show that if $u : \mathbb{Q}$, $r : \mathbb{R}$, $p : \sim\mathbb{R} r$ and $u + r \in [-1, 1]$ then $\sim\mathbb{R} (\text{addR } u r)$ holds, i.e. $\sim\text{addR } u r p q \dots : \sim\mathbb{R} (\text{addR } u r)$ where $q : u + r \in [-1, 1]$.

We will more generally instead of showing closure of $\sim\mathbb{R}$ under multiplication by 4 show closure under multiplication by 2^n , and define a function $\text{scale}^n n r' = 2^n * r'$ as follows

$$\begin{aligned} \text{embed}\mathbb{N}\rightarrow\mathbb{Z} &: \mathbb{N} \rightarrow \mathbb{Z} \\ \text{embed}\mathbb{N}\rightarrow\mathbb{Z} \text{ zero} &= \hat{\text{zero}} \\ \text{embed}\mathbb{N}\rightarrow\mathbb{Z} (\text{suc } y) &= \text{pos } (y + 1) \end{aligned}$$

$$\begin{aligned} \text{scale}^n &: \mathbb{N} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \\ \text{scale}^n n r &= 2^{\wedge} (\text{embed}\mathbb{N}\rightarrow\mathbb{Z} n) * r \end{aligned}$$

And we show that if $\sim\mathbb{R} r$, $r \in [-2^{-n}, 2^{-n}]$ then $\sim\mathbb{R} (\text{scale}^n n r)$ holds i.e. define $\sim\text{scale}^n n r \dots : \sim\mathbb{R} (\text{scale}^n n r)$. Once we have defined $\sim\text{scale}^n$ and $\sim\text{addR}$, we get for

$$r' = \text{mpaux } a b 0 0 = \frac{a * b + 0 + 0}{4}$$

that

$$\begin{aligned}
 a * b &= \text{mp } a \ b \\
 &= \text{scale}^2 \ r' \\
 &= \text{addR } (\text{appr}^2 \ r' \ Rr') \ (\text{tail}^2 \ r' \ Rr') \\
 \\
 \sim \text{mp } a \ b \ \dots &= \sim \text{scale}^2 \ r' \ \dots \\
 &= \sim \text{addR } (\text{appr}^2 \ r' \ Rr') \ (\text{tail}^2 \ Rr') \ p \ q \ \dots \\
 &: \sim \mathbb{R} \ (\text{mp } a \ b) \\
 &= \sim \mathbb{R} \ (\text{scale}^2 \ r') \\
 &= \sim \mathbb{R} \ (\text{addR } (\text{appr}^2 \ r' \ Rr') \ (\text{tail}^2 \ r' \ Rr')) \\
 &= \sim \mathbb{R} \ (a * b)
 \end{aligned}$$

have and proved $\sim \mathbb{R} \ (a * b)$.

In the following sections, we firstly show that $\sim \mathbb{R}$ is closed under the `mpaux` function. Secondly, we show $\sim \mathbb{R}$ is closed under the `scalen` function. Thirdly, we show $\sim \mathbb{R}$ is closed under `addR` function. Finally, we show $\sim \mathbb{R}$ is closed under the `mp` function followed by some examples of the multiplication function.

7.4.1 $\sim \mathbb{R}$ Is Closed Under the `mpaux` Function

(This section corresponds to Section 6.5.1.)

In Section 6.5.1 we have seen the following: Let a_0, c_0 be the first digits of a, c , $a' = 2a - a_0$, $c' = 2c - c_0$ and $e = \text{avaux}(a_0 * b, c', i)$. Let e_0, e_1 be the first two digits of e , $e' = 2e - e_0$, $e'' = 2e - e_1$. Then $2 * \text{mpaux}(a, b, c, i) - d' = \text{mpaux}(a', b, e'', i')$. Let $j = 2e_0 + e_1 + c_0 + i$ and $i' = j - 4d'$. If we choose

$$d' = \begin{cases} 0, & \text{if } -2 \leq j \leq 2; \\ 1, & \text{if } j > 2; \\ -1, & \text{if } j < -2 \end{cases}$$

then $i' \in [-2, 2]$. In order to be in accordance with our Agda code we write δ for d' . We first show closure of $\sim \mathbb{R}$ under multiplication by a digit (Lemma 6.5.2).

We introduce a function multiplying by a digit `midig` $r \ i = i * r$ together with

a proof

$$\sim\text{midig } r \ i \ \dots : \sim\mathbb{R} (\text{midig } r \ i)$$

holds, where $i : \text{Digit}$. One can define this in Agda as follows

$$\begin{aligned} \text{midig} &: \mathbb{R} \rightarrow \text{Digit} \rightarrow \mathbb{R} \\ \text{midig } r \ i &= \text{embedD } i * r \end{aligned}$$

$$\sim\text{midigaux} : (r \ s : \mathbb{R}) \rightarrow (i : \text{Digit}) \rightarrow \sim\mathbb{R} \ r \rightarrow s == \text{midig } r \ i \rightarrow \sim\mathbb{R} \ s$$

$$\sim\text{midigaux } r \ s \ i \ (\text{cons } d \ .r \ Rr' \ pr) \ p = \text{cons } (i *_{(d)} d) \ s \ \sim\mathbb{R} s' \ w$$

where

$$\begin{aligned} w &: s \in [-1, 1] \\ w &= \dots \end{aligned}$$

$$\begin{aligned} w1 &: \text{r}2 * s - \text{embedD}(i *_{(d)} d) == \text{embedD } i * (\text{r}2 * r - (\text{embedD } d)) \\ w1 &= \dots \end{aligned}$$

$$\begin{aligned} \sim\mathbb{R} s' &: \sim\mathbb{R} (\text{r}2 * s - \text{embedD}(i *_{(d)} d)) \\ \sim\mathbb{R} s' &= \sim\text{midigaux } (\text{r}2 * r - \text{embedD } d) \\ &\quad (\text{r}2 * s - \text{embedD}(i *_{(d)} d)) \ i \ Rr' \ w1 \end{aligned}$$

$$\sim\text{midig} : (r : \mathbb{R})(i : \text{Digit}) \rightarrow \sim\mathbb{R} \ r \rightarrow \sim\mathbb{R} (\text{midig } r \ i)$$

$$\sim\text{midig } r \ i \ Rr = \sim\text{midigaux } r \ (\text{embedD } i * r) \ i \ Rr \ (\text{refl} == (\text{embedD } i * r))$$

which allows us to compute a digit stream of a digit times a real number.

Now we show closure of $\sim\mathbb{R}$ under the `mpaux` function (Lemma 6.5.1). We define function in Agda as follows

$$\begin{aligned} \text{mpaux} &: (r \ p \ q : \mathbb{R}) \rightarrow (i : \text{Digit2}) \rightarrow \mathbb{R} \\ \text{mpaux } r \ p \ q \ i &= (r * p + q + \text{embedD2 } i) * \text{r}1/4 \end{aligned}$$

$$\begin{aligned} \text{mpaux} \in [-1, 1] &: (r \ p \ q : \mathbb{R})(i : \text{Digit2}) \\ &\rightarrow r \in [-1, 1] \\ &\rightarrow p \in [-1, 1] \\ &\rightarrow q \in [-1, 1] \end{aligned}$$

$$\begin{aligned} &\rightarrow (\text{embedD2 } i) \in [-2, 2] \\ &\rightarrow (\text{mpaux } r \ p \ q \ i) \in [-1, 1] \\ \text{mpaux} \in [-1, 1] \ r \ p \ q \ i \ r' \ p' \ q' \ i' = \text{and } v2 \ v1 \end{aligned}$$

where

$$\begin{aligned} v2 : & - \ ^r1 \leq \text{mpaux } r \ p \ q \ i \\ v2 = & \dots \end{aligned}$$

$$\begin{aligned} v2 : & \text{mpaux } r \ p \ q \ i \leq \ ^r1 \\ v1 = & \dots \end{aligned}$$

where the $\text{mpaux} \in [-1, 1]$ function guarantees that $\text{mpaux } r \ p \ q \ i$ is in the interval $[-1, 1]$ if $r, p, q \in [-1, 1]$ and $i \in [-2, 2]$.

$$\begin{aligned} \text{mpcomputej} : & (e_0 \ e_1 \ c_0 : \text{Digit})(i : \text{Digit2}) \rightarrow \mathbb{Z} \\ \text{mpcomputej } e_0 \ e_1 \ c_0 \ i = & (i+2) *^i \text{embedD} \rightarrow \mathbb{Z} \ e_0 \\ & +^i \text{embedD} \rightarrow \mathbb{Z} \ e_1 +^i \text{embedD} \rightarrow \mathbb{Z} \ c_0 +^i \text{embedD2} \rightarrow \mathbb{Z} \ i \end{aligned}$$

$$\begin{aligned} \text{mpcorrectnessj} : & (e_0 \ e_1 \ c_0 : \text{Digit}) \\ & \rightarrow (i : \text{Digit2}) \\ & \rightarrow (\text{embed}\mathbb{Z} \rightarrow \mathbb{R} (\text{mpcomputej } e_0 \ e_1 \ c_0 \ i)) \in [-6, 6] \end{aligned}$$

$$\text{mpcorrectnessj } e_0 \ e_1 \ c_0 \ i = \text{and } v \ v'$$

where

$$\begin{aligned} v : & - \ ^r6 \leq \text{embed}\mathbb{Z} \rightarrow \mathbb{R} (\text{mpcomputej } e_0 \ e_1 \ c_0 \ i) \\ v = & \dots \end{aligned}$$

$$\begin{aligned} v' : & \text{embed}\mathbb{Z} \rightarrow \mathbb{R} (\text{mpcomputej } e_0 \ e_1 \ c_0 \ i) \leq \ ^r6 \\ v' = & \dots \end{aligned}$$

We are able to compute $j = 2e_0 + e_1 + c_0 + i$ as $(\text{mpcorrectnessj } e_0 \ e_1 \ c_0 \ i)$ and shows that $(\text{mpcomputej } e_0 \ e_1 \ c_0 \ i)$ is in the interval $[-6, 6]$.

$$\begin{aligned} \text{computed}\delta\text{-aux} : & (j : \mathbb{Z}) \\ & \rightarrow ((\text{embed}\mathbb{Z} \rightarrow \mathbb{R} \ j) < - \ ^r2 \ \vee \ (\text{embed}\mathbb{Z} \rightarrow \mathbb{R} \ j) \in [-2, 2]) \\ & \quad \vee \ ^r2 < (\text{embed}\mathbb{Z} \rightarrow \mathbb{R} \ j) \\ & \rightarrow \text{Digit} \end{aligned}$$

$$\text{computed}\delta\text{-aux } j \ (\text{inl } (\text{inl } ll)) = \text{(d)} - 1$$

$\text{computed}\delta\text{-aux } j \text{ (inl (inr } rr)) = {}_{(d)}0$

$\text{computed}\delta\text{-aux } j \text{ (inr } r) = {}_{(d)}1$

$\text{computed}\delta : (e_0 e_1 c_0 : \text{Digit})(i : \text{Digit2}) \rightarrow \text{Digit}$

$\text{computed}\delta \ d \ e \ f \ i = \text{computed}\delta\text{-aux}$

$(\text{mpcomputej } d \ e \ f \ i)$

$(j < -n \vee j \in [-n, n] \vee n < j \ (\text{mpcomputej } d \ e \ f \ i)^{i+2})$

$(\text{computed}\delta \ e_0 \ e_1 \ c_0 \ i)$ computes a digit which is the first digit δ . If $(\text{mpcomputej } d \ e \ f \ i)$ is less than $-r2$ then $\delta = 1$. If $(\text{mpcomputej } d \ e \ f \ i)$ is in the interval $[-2, 2]$ then $\delta = 0$. Otherwise, $\delta = 1$. Since we have j and δ we can compute i' for $i' = j - 4\delta$.

$\text{mpcomputei}'\text{-aux}' : (j : \mathbb{Z})$

$\rightarrow (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j \in [-6, 6])$

$\rightarrow ((\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) < -r2 \ \vee \ (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) \in [-2, 2])$

$\vee r2 < (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j)$

$\rightarrow \mathbb{Z}$

$\text{mpcomputei}'\text{-aux}' \ j \ p \ p' = j - {}^{i}i + 4 * {}^i \text{embedD} \rightarrow \mathbb{Z} \ (\text{computed}\delta\text{-aux } j \ p')$

$\text{mpcorrectnesscomputei}'\text{-aux}' : (j : \mathbb{Z})$

$\rightarrow (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j \in [-6, 6])$

$\rightarrow (q : ((\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) < -r2 \ \vee \ (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j) \in [-2, 2])$

$\vee r2 < (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ j))$

$\rightarrow (\text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ (\text{mpcomputei}'\text{-aux}' \ j \ p \ q)) \in [-2, 2]$

$\text{mpcorrectnesscomputei}'\text{-aux}' \ j \ (\text{and } pl \ pr) \ (\text{inl } (\text{inl } ll)) = \text{and } w1 \ v1$

where

$w1 : -r2 \leq \text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ (j + {}^i i + 4)$

$w1 = \dots$

$v1 : \text{embed}\mathbb{Z}\rightarrow\mathbb{R} \ (j + {}^i i + 4) \leq r2$

$v1 = \dots$

$\text{mpcorrectnesscomputei}'\text{-aux}' \ j \ (\text{and } pl \ pr) \ (\text{inl } (\text{inr } (lrl \ lrr))) = \text{and } w \ v$

where

$$w : - \text{r}2 \leq \text{embed}\mathbb{Z} \rightarrow \mathbb{R} (j + i \cdot 0)$$

$$w = \dots$$

$$v : \text{embed}\mathbb{Z} \rightarrow \mathbb{R} (j + i \cdot 0) \leq \text{r}2$$

$$v = \dots$$

$\text{mpcorrectnesscompute}'\text{-aux}' j$ (and $pl\ pr$) (inl ($\text{inr } r$) = and $w\ v$

where

$$w : - \text{r}2 \leq \text{embed}\mathbb{Z} \rightarrow \mathbb{R} (j + i \cdot i - 4)$$

$$w = \dots$$

$$v : \text{embed}\mathbb{Z} \rightarrow \mathbb{R} (j + i \cdot i - 4) \leq \text{r}2$$

$$v = \dots$$

$(\text{mpcompute}'\text{-aux}' j\ p\ p')$ is defined as $j - 4f$ and $(\text{mpcorrectnesscompute}'\text{-aux}' j\ p\ p')$ shows that $(\text{mpcompute}'\text{-aux}' j\ p\ p')$ is in interval $[-2, 2]$.

$$\begin{aligned} \text{mpcompute}'\text{-aux} : (j : \mathbb{Z}) & \\ & \rightarrow (\text{embed}\mathbb{Z} \rightarrow \mathbb{R} j \in [-6, 6]) \\ & \rightarrow ((\text{embed}\mathbb{Z} \rightarrow \mathbb{R} j) < -\text{r}2 \vee (\text{embed}\mathbb{Z} \rightarrow \mathbb{R} j) \in [-2, 2]) \\ & \quad \vee \text{r}2 < (\text{embed}\mathbb{Z} \rightarrow \mathbb{R} j) \\ & \rightarrow \text{Digit2} \end{aligned}$$

$$\begin{aligned} \text{compute}'\text{-aux } j\ p\ p' = z \in [-2, 2] \rightarrow D2 (\text{mpcompute}'\text{-aux}' j\ p\ p') \\ (\text{mpcorrectnesscompute}'\text{-aux}' j\ p\ p') \end{aligned}$$

$$\text{mpcompute}' : (e_0\ e_1\ c_0 : \text{Digit})(i : \text{Digit2}) \rightarrow \text{Digit2}$$

$$\begin{aligned} \text{compute}'\ d\ e\ f\ i = \text{mpcompute}'\text{-aux} \\ (\text{mpcompute } j\ d\ e\ f\ i) \\ (\text{mpcorrectness } j\ d\ e\ f\ i) \\ (j < -n \vee j \in [-n, n] \vee n < j (\text{mpcompute } j\ d\ e\ f\ i)^{i+2}) \end{aligned}$$

$(\text{mpcompute}'\ d'\ e'\ i)$ computes i' which is a type of Digit2 . It computes $j = 2e_0 + e_1 + c_0 + i$ and finds a digit δ with a proof that $i' = j - 4\delta$ is in the interval $[-2, 2]$.

$$\sim\text{mpaux} : (r\ p\ q\ r' : \mathbb{R}) \rightarrow (i : \text{Digit2}) \rightarrow \sim\mathbb{R}\ r \rightarrow \sim\mathbb{R}\ p \rightarrow \sim\mathbb{R}\ q$$

$$\rightarrow r' == (\text{mpaux } r \ p \ q \ i)$$

$$\rightarrow \sim\mathbb{R} \ r'$$

$$\sim\text{mpaux } (\text{cons } a_0 \ .r \ Rr' \ pr) \ (\text{cons } b_0 \ .p \ Rp' \ pp) \ (\text{cons } c_0 \ .q \ Rq' \ pq) \ p= = \\ \text{cons } \delta \ r' \ 1H \ p='$$

where

$$e : \mathbb{R}$$

$$e = \text{avaux } (\text{embedD } a_0 \ * \ p) \ (\text{r2} \ * \ q - \text{embedD } c_0) \ i$$

$$\sim\mathbb{R}e : \sim\mathbb{R} \ e$$

$$\sim\mathbb{R}e = \sim\text{avaux } (\text{embedD } a_0 \ * \ p) \ (\text{r2} \ * \ q - \text{embedD } c_0) \ e \ i \\ (\sim\text{midig } p \ a_0 \ (\text{cons } b_0 \ p \ Rp' \ pp)) \ Rq' \ (\text{refl} == e)$$

$$e_0 : \text{Digit}$$

$$e_0 = \text{head } e \ \sim\mathbb{R}e$$

$$e' : \mathbb{R}$$

$$e' = \text{tail}^n \ 1 \ e \ \sim\mathbb{R}e$$

$$\sim\mathbb{R}e' : \sim\mathbb{R} \ e'$$

$$\sim\mathbb{R}e' = \sim\text{tail}^n \ 1 \ e \ \sim\mathbb{R}e$$

$$e_1 : \text{Digit}$$

$$e_1 = \text{head } e' \ \sim\mathbb{R}e'$$

$$e'' : \mathbb{R}$$

$$e'' = \text{tail}^n \ 2 \ e \ \sim\mathbb{R}e$$

$$\sim\mathbb{R}e'' : \sim\mathbb{R} \ e''$$

$$\sim\mathbb{R}e'' = \sim\text{tail}^n \ 2 \ e \ \sim\mathbb{R}e$$

$$\delta : \text{Digit}$$

$$\delta = \text{computed} \ \delta \ e_0 \ e_1 \ c_0 \ i$$

$i' : \text{Digit2}$

$i' = \text{mpcompute} i' e_0 e_1 c_0 i$

$p = ' : r' \in [-1, 1]$

$p = ' = \dots$

$qq : (r2 * r' - \text{embedD} \delta) == \text{mpaux}(r2 * r - \text{embedD} a_0) p e'' 'i'$

$qq = \dots$

$1H : \sim\mathbb{R} (r2 * r' - \text{embedD} \delta)$

$1H = \sim\text{mpaux} (r2 * r - \text{embedD} a_0) p e'' (r2 * r' - \text{embedD} \delta) 'i'$

$Rr' (\text{cons } b_0 p Rp' pp) \sim\mathbb{R} e'' qq$

So far we are able to compute $r' = (a * b + q + i)/4$ and its digit stream $\sim\mathbb{R} r'$ by functions mpaux , $\sim\text{mpaux}$. Next we are going to introduce functions $\sim\text{addR}$ and $\sim\text{scale}^n$.

7.4.2 $\sim\mathbb{R}$ Is Closed Under the addR Function

(This section corresponds to Section 6.5.2.)

Now we show that $\sim\mathbb{R}$ is closed under the function addR . If $u : \mathbb{Q}$, $a : \mathbb{R}$, $\sim\mathbb{R} a$ holds, $\text{addR } u a = u + a$ and $u + a \in [-1, 1]$ then $\sim\mathbb{R} (\text{addR } u a)$ holds. We introduce the function $\sim\text{addR } u a p p' \dots : \sim\mathbb{R} (\text{addR } u a)$ where $p : \sim\mathbb{R} a$, $p' : u + a \in [-1, 1]$ such that

$\sim\text{addR} : (u : \mathbb{Q}) \rightarrow (a : \mathbb{R}) \rightarrow \sim\mathbb{R} a \rightarrow (u + a) \in [-1, 1] \rightarrow \sim\mathbb{R} (\text{addR } u a)$

In Section 6.5.2 we have seen the following: Let $a \in \sim\mathbb{R}$, $u \in \mathbb{Q}$ and $u + a \in [-1, 1]$. Let a_0, a_1 be the first two digits of a , $a' = 2a - a_0$, $a'' = 2a' - a_1$. Then $2 * \text{addR}(u, a) - d' = \text{addR}(u', r)$ where $u' = 2u + a_0 - d'$, $r = a'$ and with

$q = (u + a_0/2 + a_1/4)$ we have

$$d' = \begin{cases} -1, & \text{if } q < -1/4; \\ 0, & \text{if } -1/4 \leq q \leq 1/4; \\ 1, & \text{if } 1/4 < q \end{cases}$$

Then we get that with this choice of d' that $u' + r \in [-1, 1]$. In order to be in accordance with our Agda code we write δ instead of d' .

We can now introduce the $\sim\text{addR}$ function in Agda as follows

```
cases3 : (q : ℚ) → (embedℚ→ℝ q) < r-1/4 ∨ r-1/4 ≤ (embedℚ→ℝ q)
          → (embedℚ→ℝ q) ≤ r1/4 ∨ r1/4 < (embedℚ→ℝ q) → Digit
cases3 q (inl y) p' = (d)-1
cases3 q (inr y) (inl y') = (d)0
cases3 q (inr y) (inr y') = (d)1
```

```
 $\sim\text{addR} : (u : ℚ) → (r s : ℝ) → \simℝ r → \text{embed}\mathbb{Q}\rightarrow\mathbb{R} u + r \in [-1, 1]$ 
          → s == addR u r → \simℝ s
 $\sim\text{addR} u r s Rr p' = \text{cons } \delta s \simℝs' v4$ 
```

where

```
a0 : Digit
a0 = head r Rr
```

```
a1 : Digit
a1 = head (tail r Rr) (\simtail r Rr)
```

```
q : ℚ
q = u + ℚ (embedD→ℤ a0) %' +2 + ℚ (embedD→ℤ a1) %' +4
```

```
δ : Digit
δ = cases3 q (q < -1/4 ∨ -1/4 ≤ q) (q ≤ 1/4 ∨ 1/4 < q)
```

```
r' : ℝ
```

$$r' = \text{tail } r \text{ } Rr$$

$$u' : \mathbb{Q}$$

$$u' = (\text{pos } +2 \%' +1) * \mathbb{Q} \ u + \mathbb{Q} \ (\text{embedD} \rightarrow \mathbb{Z} \ a_0 \%' +1) \\ - \mathbb{Q} \ (\text{embedD} \rightarrow \mathbb{Z} \ \delta \%' +1)$$

$$v1 : (\text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ u') + r' \in [-1, 1]$$

$$v1 = \dots$$

$$w1 : \text{r}2 * \text{semedD } \delta == (\text{embed} \mathbb{Q} \rightarrow \mathbb{R} \ u') + r'$$

$$w1 = \dots$$

$$\sim \mathbb{R} s' : \sim \mathbb{R} (\text{r}2 * \text{semedD } \delta)$$

$$\sim \mathbb{R} s' = \sim \text{addR } u' \ r' (\text{r}2 * \text{semedD } \delta) (\sim \text{tail } r \text{ } Rr) \ v1 \ w1$$

$$v4 : s \in [-1, 1]$$

$$v4 \dots$$

So in this section we have shown that $\sim \mathbb{R}$ is closed under the `addR` function, i.e. if $(q : \mathbb{Q})$, $(a : \mathbb{R})$, $\sim \mathbb{R} \ a$ and $(u + a) \in [-1, 1]$ then $\sim \mathbb{R} \ (\text{addR } u \ a)$.

7.4.3 $\sim \mathbb{R}$ Is Closed Under the Function `scalen`

We remember `scalen n r = 2n * r` and we want to show that

$$\sim \text{scale}^n \ n \ r \ \dots : \sim \mathbb{R} \ (\text{scale}^n \ n \ r)$$

Before we introduce `~scalen` function, we first introduce the functions `apprn` and `tailn`. Note that for $r : \mathbb{R}$, $r \sim 0.a_0a_1a_2\dots$ (in Agda we write `apprn n` for `apprn` and `tailn n` for `tailn`)

$$\text{scale}^n \ r = 2^n * r \sim \underbrace{a_0a_1 \cdots a_{n-1}}_{\text{appr}^n \ r} \cdot \underbrace{l}_{\text{tail}^n \ r}$$

Chapter 7. Signed Digit Representation of Real Numbers in Agda

where $\text{appr}^n r = 2^{n-1}a_0 + 2^{n-2}a_1 + \dots + a_{n-1}$, $l = \text{tail}^n (r)$. The functions appr^n and tail^n are defined as follows

mutual

$$\text{tail}^n : (n : \mathbb{N}) \rightarrow (r : \mathbb{R}) \rightarrow \sim\mathbb{R} r \rightarrow \mathbb{R}$$

$$\text{tail}^n 0 r Rr = r$$

$$\text{tail}^n (\text{suc } n) r Rr = \text{tail} (\text{tail}^n n r Rr) (\sim\text{tail}^n n r Rr)$$

$$\sim\text{tail}^n : (n : \mathbb{N}) \rightarrow (r : \mathbb{R}) \rightarrow \sim\mathbb{R} r \rightarrow \sim\mathbb{R} (\text{tail}^n n r Rr)$$

$$\sim\text{tail}^n 0 r Rr = Rr$$

$$\sim\text{tail}^n (\text{suc } n) r Rr = \sim\text{tail} (\text{tail}^n n r Rr) (\sim\text{tail}^n n r Rr)$$

$$\text{appr}^n : (n : \mathbb{N}) \rightarrow (r : \mathbb{R}) \rightarrow \sim\mathbb{R} r \rightarrow \mathbb{Z}$$

$$\text{appr}^n 0 r Rr = \hat{\text{zero}}$$

$$\text{appr}^n (\text{suc } n) r Rr = {}^{i+2} *^i \text{appr}^n n r Rr + {}^i \text{embedD} \rightarrow \mathbb{Z} (\text{head} (\text{tail}^n n r Rr) (\sim\text{tail}^n n r Rr))$$

Now we can introduce the $\sim\text{scale}^n$ function in Agda as follows

$$\sim\text{scale}^n : (n : \mathbb{N}) \rightarrow (r s : \mathbb{R}) \rightarrow (Rr : \sim\mathbb{R} r)$$

$$\rightarrow \text{scale}^n n r \in [-1, 1]$$

$$\rightarrow s == \text{scale}^n n r$$

$$\rightarrow \sim\mathbb{R} s$$

$$\sim\text{scale}^n n r s Rr p p' = \text{addR} (\text{appr}^n n r Rr \%' + 1) (\text{tail}^n n r Rr) s (\sim\text{tail}^n n r Rr) w p''$$

where

$$p'' : s == \text{embedQ} \rightarrow \mathbb{R} (\text{appr}^n n r Rr \%' + 1) + \text{tail}^n n r Rr$$

$$p'' = \dots$$

$$w : \text{embedQ} \rightarrow \mathbb{R} (\text{appr}^n n r Rr \%' + 1) + \text{tail}^n n r Rr \in [-1, 1]$$

$$w = \dots$$

So in this section we have shown that if $(r : \mathbb{R})$, $(\sim\mathbb{R} r)$ holds and $\text{scale}^n n r \in [-1, 1]$ then $\sim\text{scale}^n n r \dots : \sim\mathbb{R} (\text{scale}^n n r)$. Let $u = \text{appr}^2 r Rr$ and

$a = \text{tail}^2 r Rr$. Then we get

$$\text{scale}^2 r = \text{addR } u a = 2^2 * r$$

$$\sim \text{scale}^2 r \dots = \sim \text{addR } u a \dots : \sim \mathbb{R} (2^2 * r)$$

7.4.4 $\sim \mathbb{R}$ Closure Under mp

(This section corresponds to Section 6.5.3.)

The multiplication function mp is defined in Agda as follows

$$\sim \text{mp} : (a b s : \mathbb{R}) \rightarrow (Ra : \sim \mathbb{R} a) \rightarrow (Rb : \sim \mathbb{R} b) \rightarrow s == a * b \rightarrow \sim \mathbb{R} s$$

$$\sim \text{mp } a b s Ra Rb ps = \text{scale}^n 2 r' s Rr' v p3$$

where

$$r' : \mathbb{R}$$

$$r' = \text{mpaux } a b r' 0_{(d)_2} 0$$

$$Rr' : \sim \mathbb{R} r'$$

$$Rr' = \sim \text{mpaux } a b r' 0_{(d)_2} 0 Ra Rb \sim \mathbb{R} 0 (\text{refl} == r')$$

$$p3 : s == \text{scale}^n 2 r'$$

$$p3 = \dots$$

$$v : \text{scale}^n 2 r' \in [-1, 1]$$

$$v = \dots$$

where $p3$ is obtained using equality reasoning and v is a proof that $(\text{scale}^n 2 r')$ is in the interval $[-1,1]$. We know that $\sim \mathbb{R}$ is closed under mpaux , i.e.

$$\sim \text{mpaux } a b q r' i \dots : \sim \mathbb{R} (\text{mpaux } a b q i)$$

$\sim \mathbb{R}$ is closed under scale^n , i.e.

$$\sim \text{scale}^n n r' \dots : \sim \mathbb{R} (\text{scale}^n n r')$$

holds.

$\sim\text{mp}$ function shows us that if $a\ b\ s : \mathbb{R}$, $s = a * b$, $\sim\mathbb{R}\ a$, $\sim\mathbb{R}\ b$, then we get $\sim\mathbb{R}\ s$

$$\begin{aligned} a * b &= \text{mp } a\ b \\ &= \text{scale}^2 (\text{mpaux } a\ b\ 0\ 0) \\ &= s \end{aligned}$$

$$\begin{aligned} \sim\text{mp } a\ b \ \dots &= \sim\text{scale}^2\ r' \ \dots \\ &: \sim\mathbb{R}\ (a * b) \\ &= \sim\mathbb{R}\ (\text{mp } a\ b) \\ &= \sim\mathbb{R}\ (\text{scale}^2 (\text{mpaux } a\ b\ 0\ 0)) \\ &= \sim\mathbb{R}\ s \end{aligned}$$

7.4.5 Examples of the Multiplication Function

Here are some examples of the multiplication function:

$$\begin{aligned} \text{mp1*1} &: \sim\mathbb{R}\ (r1 * r1) \\ \text{mp1*1} &= \sim\text{mp } r1\ r1\ (r1 * r1)\ \sim\mathbb{R}1\ \sim\mathbb{R}1\ (\text{refl} == (r1 * r1)) \end{aligned}$$

mp1*1 is the proof that signed digits of the multiplication of real numbers 1 and 1. Similarly, we get

$$\text{mp0*0} : \sim\mathbb{R}\ (r0 * r0)$$

and

$$\text{mp}q1/4*1/3 : \sim\mathbb{R}\ (\text{embed}\mathbb{Q}\rightarrow\mathbb{R}\ q1/4 * \text{embed}\mathbb{Q}\rightarrow\mathbb{R}\ q1/3)$$

7.5 Defining $\sim\mathbb{R}$ Using the New Representation of $\sim\mathbb{R}$

Above we have introduced $\sim\mathbb{R}$ using a slightly older version of Agda, which had codata type. In the current version 2.2.6, the syntax has changed. There codata type are introduced using the keyword **data**, and coalgebraic arguments of the constructor are indicated by applying the construction ∞ to it. We give now the definition of $\sim\mathbb{R}$ using this new syntax.

Stream, $\sim\mathbb{R}$ and $\sim\mathbb{R}$ toStream in the current Agda version 2.2.6 with the standard library 0.3 using the new version of representing coalgebras in Agda discussed in Section 4.2 are defined as follows: we first import the library file Coinduction.agda.

```
open import Coinduction
```

```
data Stream (A : Set) : Set where
  _ :: _ : A → ∞ (Stream A) → Stream A
```

```
data  $\sim\mathbb{R}$  :  $\mathbb{R}$  → Set where
  cons : (d : Digit)(r :  $\mathbb{R}$ )
    → ∞ ( $\sim\mathbb{R}$  ( $\tau_2 * r - \text{embedD } d$ ))
    →  $r \in [-1, 1]$ 
    →  $\sim\mathbb{R} r$ 
```

```
 $\sim\mathbb{R}$ toStream : (r :  $\mathbb{R}$ ) →  $\sim\mathbb{R} r$  → Stream Digit
 $\sim\mathbb{R}$ toStream r (cons d .r p q) = d :: (# ( $\sim\mathbb{R}$ toStream ( $\tau_2 * r - \text{embedD } d$ ) (b p)))
```

7.6 Computing the Extracted Program

Instead of defining the function $\sim\mathbb{R}$ toStream : (r : \mathbb{R}) → $\sim\mathbb{R} r$ → Stream Digit we define a "find digit" function fd with the result type String which allows us to show the first n digits in the form $0.d_0d_1d_2\cdots$, e.g. "0.01(-1)01". In Agda one can define String as follows

```

postulate String : Set

{-# COMPILED_TYPE String String #-}
{-# BUILTIN STRING String #-}

primitive
  primStringAppend : String → String → String

- - function appending two strings
_+_ : String → String → String
_+_ = primStringAppend

```

Unfortunately in Agda `String` is not defined as a `BUILTIN` but a primitive type. That's why it looks like a postulate type but it could be replaced by `String = List Char` and Agda behaves as if we had `String = List Char`, and as if `primStringAppend` and `_+_` were defined in the standard way. Furthermore, the Haskell `String` type is imported for the compiled version of Agda. We also define an append function `_+_` which allows us to join two strings by adopting Agda native function `primStringAppend`.

The `fd` function is defined in Agda as follows

```

fdaux : (n : ℕ) → (r : ℝ) → ~ℝ r → String
fdaux 0 r Rr = ""
fdaux (suc n') r (cons (d)0 .r y y') = "0" ++ fdaux n' (r2 * r - embedD (d)0) y
fdaux (suc n') r (cons (d)1 .r y y') = "1" ++ fdaux n' (r2 * r - embedD (d)1) y
fdaux (suc n') r (cons (d)-1 .r y y') = "(-1)" ++
  fdaux n' (r2 * r - embedD (d)-1) y

fd : (n : ℕ) → (r : ℝ) → ~ℝ r → String
fd n r Rr = "0." ++ fdaux n r Rr

```

With the `fd` function we are able to show the signed digit streams of a real number r s.t. $\sim\mathbb{R} r$. For instance, to show 100 digits of

Chapter 7. Signed Digit Representation of Real Numbers in Agda

1. $\sim\mathbb{R}(-^r 1)$, we can write `fd 100 (-^r 1) (~\mathbb{R}-1)` for the real number -1.
2. $\sim\mathbb{R}(\text{embed}\mathbb{Q}\rightarrow\mathbb{R} q2/3)$, we can write `fd 100 (embed\mathbb{Q}\rightarrow\mathbb{R} q2/3) (~\mathbb{R}q2/3)` for real number 2/3.
3. $\sim\mathbb{R}((^r 1 + ^r 1) * ^r 1/2)$, we can write `fd 100 ((^r 1 + ^r 1) * ^r 1/2) av1+1` for the average of two real numbers 1 and 1.
4. $\sim\mathbb{R}((^r 1 * ^r 1))$, we can write `fd 100 (^r 1 * ^r 1) mp1*1` for the multiplication of two real numbers 1 and 1.

In order to show digits on the screen, we use the compiled version of Agda. It requires to import Haskell data types `Unit`, `List`, `IO`, and the Haskell function `putStrLn`. One can write them in Agda as follows

```
data Unit : Set where
```

```
  unit : Unit
```

```
{-# COMPILED_TYPE Unit () () #-}
```

```
data List (A : Set) : Set where
```

```
  [] : List A
```

```
  _::_ : A → List A → List A
```

```
{-# COMPILED_TYPE List [] [] (:) #-}
```

```
{-# BUILTIN LIST List #-}
```

```
{-# BUILTIN NIL [] #-}
```

```
{-# BUILTIN CONS _::_ #-}
```

```
postulate
```

```
  IO : Set → Set
```

```
  putStrLn : String → IO Unit
```

```
{-# COMPILED_TYPE IO IO #-}
```

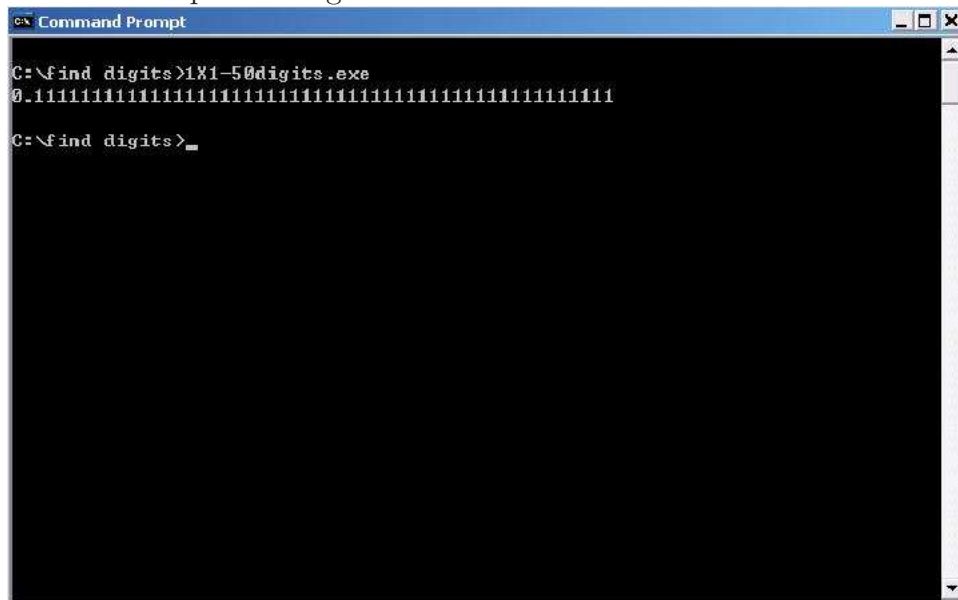
```
{-# COMPILED putStrLn putStrLn #-}
```

To Show 50 digits of real number $1 * 1$ on the screen, one can write it in Agda as follows

```
fd_mp1*1 : String
fd_mp1*1 = fd 100 (r1 * r1) mp1*1

main : IO Unit
main = putStrLn fd_mp1*1
```

Then we compile the Agda file and execute it. We obtain



It took 0.85 sec to compute the digits.

Another example shows 1000 digits of the real number

$$\frac{\frac{2}{3} - 1}{2} * \frac{1}{3}$$

on the screen, first we define the average of $2/3$ and -1

```
avq2/3-1 : ~ℝ ((embedQ → ℝ q2/3 - r1) * r1/2)
avq2/3-1 = ~av (embedQ → ℝ q2/3) (-r1) ((embedQ → ℝ q2/3 - r1) * r1/2)
~ℝ q2/3 (refl == ((embedQ → ℝ q2/3 - r1) * r1/2))
```

Then we define the multiplication of $av(2/3, -1)$ and $1/3$

```
mp_avq2/3-1*q1/3 : ~ℝ ((embedQ → ℝ q2/3 - r1) * r1/2) * embedQ → ℝ q1/3
```

```

avq2/3-1 = ~mp ((embedQ→ℝ q2/3 - r1) * r1/2)
            (embedQ→ℝ q1/3)
            ((embedQ→ℝ q2/3 - r1) * r1/2) * embedQ→ℝ q1/3)
avq2/3-1 ~ℝq1/3
(refl== ((embedQ→ℝ q2/3 - r1) * r1/2) * embedQ→ℝ q1/3)

```

```

fd_mp_avq2/3-1*q1/3 : String
fd_mp_avq2/3-1*q1/3 = fd 1000
                        ((embedQ→ℝ q2/3 - r1) * r1/2) * embedQ→ℝ q1/3)
                        mp_avq2/3-1*q1/3

```

```

main : IO Unit
main = putStrLn fd_mp_avq2/3-1*q1/3

```

Then we compile the Agda file and execute it. We obtain



It took 4.98 sec to compute the digits.

7.6.1 Testing

For some real numbers r e.g. $r = 123/456$ which is in the interval $[-1,1]$ we prove that it has signed digits representation, i.e. $p : \sim\mathbb{R} (123/456)$ holds. If our axioms are correct then $(\sim\mathbb{R}\text{toStream } r \ p)$ holds. This means there exists signed digits stream representation such that $r \sim d_0d_1d_2d_3\dots$ (or $d_0 :: d_1 :: d_2 :: d_3 :: \dots$)

$$r = \frac{d_0 + r_1}{2}, \quad r_1 = \frac{d_1 + r_2}{2}, \quad r_2 = \frac{d_2 + r_3}{2}, \quad \dots$$

and $r_i \in [-1, 1]$, $d_i \in \{-1, 0, 1\}$. We check this for some examples and see that this result is correct. We have checked it for the following numbers $(1/3)$, $(2/3)$, $(3/4)$, $(-1/6)$, $(5/12)$, $(7/24)$ up to 10 digits.

Why did we have to test this and not rely on the fact that our program is provably correct? That is because of the validation problem.

The possible validation problems could be:

- Have we translated the question, e.g. "what is SDR of 123/456" correctly into a question in Agda ?
- Is $p : \sim\mathbb{R} (123/456)$ really a proof that 123/456 has a SDR ?
- Is the list of digits we obtained from p really the sequence of digits contained in p ?
- And do we interpret the answer of Agda correctly ?

We know that p is a proof subject to postulated axioms which relied on the correctness of Agda (which is not fully proved: the theory of Agda might have mistakes, there is no formal proof, and in the implementation of Agda there might exist bugs). Despite these weaknesses, this proof is highly trustworthy so we assume that this proof is correct.

Validation is different. If we write a program and prove its correctness we can formally verify only that this program is correct against a specification. That the specification guarantees that the requirements are fulfilled is the validation problem which cannot be proved formally but only be investigated by hand, and

Chapter 7. Signed Digit Representation of Real Numbers in Agda

checked by testing. That is why testing it is still necessary (apart from the fact that we are interested in how long the computation takes).

Chapter 8

Extraction of Programs from Proofs in Agda

8.1 Program Extraction

The normal method of program extraction is to prove some properties s.t. $p : B$ in some calculation externally and then to extract the program from this proof using some external tools. The extracted program is outside the proof system. Our method is done instead completely internally. We construct a proof $b : B$ inside Agda and compute by normalisation. This is possible since there is no difference between proofs and programs in Agda. From a proof of $\forall x : A. \exists y : B. \varphi(x, y)$ we can define inside type theory the function $f : A \rightarrow B$ s.t. $\forall x : A. \varphi(x, f x)$ holds. We introduced the notion of approximatable real numbers $\sim\mathbb{R}$ which are in the interval $[-1,1]$ and have a signed digit representation, i.e $r \sim 0.a_0a_1a_2\dots$ where $a_i \in \{-1, 0, 1\}$. Then we defined a function $\sim\mathbb{R}\text{toList}$ which returns the list of the first n digits from a proof of $p : \sim\mathbb{R} r$, e.g.

$$\sim\mathbb{R}\text{toList } 7 r p = 0 :: -1 :: 0 :: 1 :: -1 :: -1 :: 0$$

means that from p we obtained $r \sim 0.0(-1)01(-1)(-1)0a_7a_8a_9\dots$ for some $a_7a_8a_9\dots$. Instead of List we even got a String representation, e.g.

$$\text{fd } 7 r p = "0.0(-1)01(-1)(-1)0"$$

We have proved that $\sim\mathbb{R}$ is closed under the average function `av` and the multiplication function `mp` for $r\ s : \mathbb{R}$. So $\sim\text{av } r\ s : \sim\mathbb{R} ((r+s)/2)$ and $\sim\text{mp } r\ s : \sim\mathbb{R} (r*s)$. So `(fd n ((r + s)/2) (~av r s))` determines the first `n` digits of $(r + s)/2$ and `(fd n (r * s) (~mp r s))` determines the first `n` digits of $r * s$. Therefore, $(r + s)/2$ and $r * s$ have signed digit approximations.

We have axioms which are postulates, which might prevent normalisation. In this chapter, we are going to prove a theorem that under certain conditions the postulated functions are not used when normalising elements of algebraic data types. The idea is that postulates only allow to derive postulated types and therefore do not influence algebraic data types. This proof contains a reduction of definitions by pattern matching to simple pattern matching. Because of our theorem elements of algebraic data types normalise to head normal form even in the presence of postulated axioms. `(fd 7 r p)` returns an actual concrete result which is correct.

In this Chapter we use classic logic.

8.2 Main Theorem: The Correctness of Program Extraction

In our proof we are going to reduce deep pattern matching to simple pattern matching. Deep pattern matching means for instance,

$$\begin{aligned} f &: \mathbb{N} \rightarrow \mathbb{N} \\ f\ 0 &= 2 \\ f\ (\text{suc } 0) &= 5 \\ f\ (\text{suc } (\text{suc } x)) &= 8 \end{aligned}$$

where `(suc 0)` and `(suc (suc x))` are deep patterns. Simple pattern matching only allows to define

$$\begin{aligned} f &: \mathbb{N} \rightarrow \mathbb{N} \\ f\ 0 &= 3 \\ f\ (\text{suc } x) &= 4 \end{aligned}$$

We will show that Agda code with deep pattern matching can be represented by simple pattern matching. This will be done in several steps. In order to prove termination of these steps we will use the multiset ordering. Therefore in the following we are going to develop the theory of multisets.

8.2.1 Mathematical Preliminaries on Multisets

In this section we review the well-known theory of multisets or bags (we use both words for the same concept). This will include a proof of the well-foundedness of bags (Lemma 8.2.11 (b)). We have derived the proof ourselves, although it is likely that this proof can be found in the literature. It is different from the proof found in [BN98], Theorem 2.2.5, p. 23.

Definition 8.2.1. *Let (A, \prec) be a set with a binary relation \prec on it. A is well-founded if only if there exists no infinitely descending sequence $a_0 \succ a_1 \succ a_2 \succ \dots$ in A . It is well-ordered if it is well-founded and linearly ordered.*

We introduce bags which are as sets but an element can occur more than once, so the multiplicity counts. Bags can be given by a function which determines the multiplicity of each element. We get the following definition:

Definition 8.2.2. *Let A be a Set. The set of bags $\text{Bag}(A)$ of A is defined as the set of functions $B : A \rightarrow \mathbb{N}$. We define*

- for a bag B and $a \in A$, $\text{multiplicity}_B(a) := B(a) \in \mathbb{N}$.
- If $a \in A, B \in \text{Bag}(A)$ then $a \in B$ if only if $\text{multiplicity}_B(a) > 0$.

If $B \in \text{Bag}(A)$ then B is considered as a collection of elements in B where each element can occur more than once. $B(a)$ denotes, how often an element occurs in the bag f . So the bag with two occurrences of 1 and one occurrence of 2 is given as $B : \{1, 2\} \rightarrow \mathbb{N}$, $B(1) = 2$ and $B(2) = 1$.

A bag $B \in \text{Bag}(A)$ is finite if $\{a \in A \mid \text{multiplicity}_B(a) > 0\}$ is finite. We write $\{|a_1, \dots, a_n|\}$ for finite bags, which has elements a_1, \dots, a_n and where the multiplicity of a_i is the number of occurrences of a_i in a_1, \dots, a_n . For example, the finite bag $A := \{|1, 2, 1|\} \in \text{Bag}(\mathbb{N})$ is the bag having elements 1,2 and

multiplicity_A(1) = 2, multiplicity_A(2) = 1. One easily sees that every finite bag can be written as $\{|x_1, \dots, x_n|\}$.

Definition 8.2.3. 1. For $B, C \in \text{Bag}(A)$ let $B \uplus C \in \text{Bag}(A)$ be defined by
multiplicity_{B \uplus C}(a) := multiplicity_B(a) + multiplicity_C(a).

2. For $B, C \in \text{Bag}(A)$ let $B \cap C \in \text{Bag}(A)$ be defined by
multiplicity_{B \cap C}(a) := min{multiplicity_B(a), multiplicity_C(a)}.

3. For $B, C \in \text{Bag}(A)$ let
 $B \subseteq C \Leftrightarrow \forall a \in A. \text{multiplicity}_B(a) \leq \text{multiplicity}_C(a)$.

4. For $B, C \in \text{Bag}(A)$ s.t. $B \subseteq C$, we define $C \setminus B \in \text{Bag}(A)$ by
multiplicity_{C \setminus B}(a) := multiplicity_C(a) - multiplicity_B(a).

Lemma 8.2.4. Assume $B, C \in \text{Bag}(A)$.

1. $B \cap C \subseteq B \subseteq B \uplus C$.
2. If $B \subseteq C$ then $C = (C \setminus B) \uplus B$.

Proof. (a) easy. (b) multiplicity_{(C \setminus B) \uplus B}(a) =

$$((\text{multiplicity}_C(a)) - \text{multiplicity}_B(a)) + \text{multiplicity}_B(a) = \text{multiplicity}_C(a)$$

since multiplicity_B(a) ≤ multiplicity_C(a) by $B \subseteq C$. □

Lemma 8.2.5. Let (A, \prec) be finite, transitive and anti-reflexive. Assume $P \subseteq A$. If there exists $a \in A$ s.t. $P(a)$ holds then there exists a maximum element $a \in A$ s.t. $P(a)$ holds (so we get $P(a) \wedge \forall a' \in A. a' \succ a. \neg P(a')$).

Proof. Assume there exists no maximum element s.t. $a \in A$. Let $a_1 \in A$ s.t. $P(a_1)$ holds. So there exists an $a_2 \in A$, $a_1 \prec a_2$ s.t. $P(a_2)$ holds. Then there exists an $a_3 \in A$, $a_2 \prec a_3$ s.t. $P(a_3)$ holds and so on. So we get $a_1 \prec a_2 \prec a_3 \prec \dots$. Since A is finite there exists an $a_i \in A$ and an $a_j \in A$ for $i, j \in \mathbb{N}$ s.t. $i < j$ and $a_i = a_j$ but then we have $a_i \prec a_j = a_i$ by transitivity. Therefore, we get a contradiction by anti-reflexivity. □

Remark 8.2.6. *By Lemma 8.2.5 it follows that every finite irreflexive, transitive order is well-founded. Since (A, \prec) has this properties, (A, \succ) has the same properties and by lemma 8.2.5 every nonempty finite set has a maximum element w.r.t. \succ which is minimal w.r.t. \prec .*

Definition 8.2.7. 1. $\text{Bag}^{\text{fin}}(A) := \{X \in \text{Bag}(A) \mid \{a \in A \mid a \in X\} \text{ is finite}\}$,
the set of finite bags in $\text{Bag}(A)$.

2. Assume \prec be a binary relation on A written infix. Then we define the relation \prec_{bag} on $\text{Bag}(A)$ by

$$\begin{aligned} B \prec_{\text{bag}} C &\Leftrightarrow C \not\subseteq B \wedge \\ &\exists X, Y, Y'. B = X \uplus Y \wedge C = X \uplus Y' \wedge \forall y \in Y. \exists y' \in Y'. y \prec y' \end{aligned}$$

Remark 8.2.8. *Assume \prec is transitive and anti-reflexive on A . If for $B, C \in \text{Bag}^{\text{fin}}(A)$ we have*

$$\begin{aligned} C \not\subseteq B \wedge \exists X, Y, Y'. B = X \uplus Y \wedge C = X \uplus Y' \wedge \\ \forall y \in Y. ((\exists y' \in Y'. y \prec y') \vee \text{multiplicity}_Y(y) < \text{multiplicity}_{Y'}(y)) \end{aligned}$$

then $B \prec_{\text{bag}} C$

Proof. of Remark: Let $U := Y \cap Y'$. By Lemma 8.2.4 we have $B = X \uplus Y = X \uplus (Y \setminus U) \uplus U = (X \uplus U) \uplus (Y \setminus U)$, similarly $C = (X \uplus U) \uplus (Y' \setminus U)$. We show $\forall y \in Y \setminus U. \exists y' \in Y' \setminus U. y \prec y'$, which implies $B \prec_{\text{bag}} C$. Assume $y \in Y \setminus U$ such that there exists no $y' \in Y' \setminus U$ s.t. $y \prec y'$. Let y be maximal, i.e. let y s.t. $\forall y_0 \succ y$ the property holds. This exists because of B being finite and Lemma 8.2.5.

Case 1: There exists $y' \in Y'$ s.t. $y \prec y'$. Let $y' \in Y'$ be maximal s.t. $y \prec y'$.

Subcase 1.1: $y' \in U$. Then $y' \in Y$. Either $y' \prec y''$ for some $y'' \in Y'$ or $\text{multiplicity}_Y(y') < \text{multiplicity}_{Y'}(y')$. In the first case there exists $y'' \in Y'$ s.t. $y' \prec y''$. But then $y \prec y'' \in Y'$ and $y' \prec y''$, so y' was not maximal, a contradiction.

In the second case $\text{multiplicity}_Y(y') < \text{multiplicity}_{Y'}(y')$. Since

$$\begin{aligned} \text{multiplicity}_U(y') &= \text{multiplicity}_{Y \cap Y'}(y') = \text{multiplicity}_Y(y'), \\ \text{multiplicity}_{Y' \setminus U}(y') &= \text{multiplicity}_{Y'}(y') - \text{multiplicity}_U(y') \\ &= \text{multiplicity}_{Y'}(y') - \text{multiplicity}_Y(y') > 0 \end{aligned}$$

Therefore, $y \prec y' \in Y' \setminus U$, a contradiction.

Subcase 1.2: $y' \notin U$. Then $y \prec y' \in Y' \setminus U$. So there exists $y' \in (Y' \setminus U)$ s.t. $y \prec y'$, a contradiction.

Case 2: $y \in Y$ and $\text{multiplicity}_Y(y) < \text{multiplicity}_{Y'}(y)$. Since $U := Y \cap Y'$, $\text{multiplicity}_U(y) = \text{multiplicity}_{Y \cap Y'}(y) = \text{multiplicity}_Y(y)$. Therefore, $\text{multiplicity}_{Y' \setminus U}(y) = 0$ and $y \notin Y' \setminus U$, a contradiction. \square

Definition 8.2.9. Let $(A, \prec), (B, \prec')$ be orderings.

1. $(A \times B, \prec_{\text{lex}})$ is the ordering given by

- Elements are pairs (a, b) for $a \in A, b \in B$.
- $(a, b) \prec_{\text{lex}} (a', b') \Leftrightarrow a \prec a' \vee (a = a' \wedge b \prec b')$ (lexicographic ordering).

2. $(A_{\text{weakdec}}, \prec_{\text{lex}})$ is given by

- Elements are weakly descending finite sequences, i.e. finite sequences (a_0, \dots, a_{n-1}) s.t. $a_0 \succeq a_1 \succeq \dots \succeq a_{n-1}$.
- $(a_0, \dots, a_{n-1}) \prec_{\text{lex}} (b_0, \dots, b_{m-1})$ iff
 - Either there exists an $i < \min\{n, m\}$, s.t. $\forall j < i. a_j = b_j$ and $a_i \prec b_i$.
 - Or $n < m$ and $\forall i < n. a_i = b_i$.

3. Assume (A, \prec) is linearly ordered. The bag ordering on A is A_{bag} , where $A_{\text{bag}} = (\text{Bag}^{\text{fin}}(A), \prec_{\text{bag}})$.

Lemma 8.2.10. Let (A, \prec) be a linearly ordered set, $X, X' \in A_{\text{bag}}^{\text{fin}}, X = \{|x_0, \dots, x_n|\}, X' = \{|x'_0, \dots, x'_m|\}$. Let l, l' be the result of reordering the lists $[x_0, \dots, x_n]$ and $[x'_0, \dots, x'_m]$ such that $x_0 \succeq x_1 \succeq \dots \succeq x_n, x'_0 \succeq x'_1 \succeq \dots \succeq x'_m$. Then $X \prec_{\text{bag}} X' \Leftrightarrow l \prec_{\text{lex}} l'$.

Proof. of “ \Rightarrow ”: Let $X = Y \uplus Z$, $X' = Y \uplus Z'$ s.t. $\forall z \in Z \exists z' \in Z'. z \prec z'$. Since $X' \not\subseteq X$ we have $Z \neq \emptyset$ or $Z' \neq \emptyset$. If $Z \neq \emptyset$, there exists $z \in Z$, $z \prec z'$ for some $z' \in Z'$, so in all cases $Z' \neq \emptyset$.

We introduce the notion of occurrence in Y : let $x_i \in X$. By x_i being an occurrence in Y (written as $x_i \tilde{\in} Y$) we mean that it is one of the first $\text{multiplicity}_Y(x_i)$ ones, otherwise an occurrence in Z (written as $x_i \tilde{\in} Z$ similarly for occurrences of $x' \in X'$ in Y or Z'). Let after ordering $X = [x_1, \dots, x_n]$, $X' = [x'_1, \dots, x'_m]$.

Case $Z = \emptyset$: Let z' be the maximum element of Z' , x'_i be the first occurrence of z' in Z' . For $j < i$, x_j is an occurrence in Y , therefore $x_j = x'_j$. x_i is an occurrence in Y , so $x_i = x'_{i'}$ for some $i' > i$, $x_i = x'_{i'} \prec x'_i$. Therefore, $l \prec_{\text{lex}} l'$.

Case $Z \neq \emptyset$: Let z be the maximum element of Z and z' be the maximum element of Z' . Then $z \prec z'$. Let x_i be the first occurrence of z in Z (x_i could be both an element of Y and Z) and x'_j be the first occurrence of z' in Z' (so there are exactly $\text{multiplicity}_Y(z')$ many occurrence of z' before). Let $Y_1 := \{|x_k \mid k < i|\}$, $Y'_1 := \{|x'_k \mid k < j|\} \subset Y$. For $u \succ z' \succeq z$ we have that $u \notin Z \uplus Z'$, since u is bigger than the greatest element in Z and Z' . Therefore, $\text{multiplicity}_X(u) = \text{multiplicity}_Y(u) = \text{multiplicity}_{X'}(u)$. Before x'_j occur $\text{multiplicity}_Y(z')$ many occurrence of z' which occur as well in X . So we get that $x_k = x'_k$ for $k < j$.

Case 1: $x_j \tilde{\in} Z$. Then $x_j \succeq x_i = z$. x_j was the largest element of Z , so $x_j = z$. We have the largest element in Z is less than the largest element in Z' therefore $x_j = z \prec z' = x'_j$, $l \prec_{\text{lex}} l'$.

Case 2: $x_j \tilde{\notin} Z$. Then $x_j \tilde{\in} Y$. $x_j \neq x'_j$ because x'_j is the first occurrence of z' , which is $\tilde{\notin} Y$, whereas x_j is an occurrence in Y . Since, $x_j \tilde{\in} Y$, $x_j = x'_l$ for some $l > j$, $x_j = x'_l \preceq x'_j$, $x_j \neq x'_j$, therefore $x_j \prec x'_j$. So $x_k = x'_k$ for $k < j$, $x_j \prec x'_j$, $l \prec_{\text{lex}} l'$.

Proof of “ \Leftarrow ”: Assume $X = \{|x_1, \dots, x_n|\}$, $X' = \{|x'_1, \dots, x'_m|\}$, $x_1 \succeq \dots \succeq x_n$, $x'_1 \succeq \dots \succeq x'_m$. Assume $(x_1, \dots, x_n) \prec_{\text{lex}} (x'_1, \dots, x'_m)$.

Case 1: $n < m$ and $x_i = x'_i$ for $i = 1, \dots, n$. Let $U := \{|x_1, \dots, x_n|\}$, $Y := \emptyset$, $Y' := \{|x'_{n+1}, \dots, x'_m|\}$. Then $X = U \uplus Y$, $X' = U \uplus Y'$ and $\forall y \in Y. \exists y' \in Y'. y \prec y'$. Furthermore, $X' \not\subseteq X$.

Case 2: There exists $i < n, m$, $x_j = x'_j$ for $j < i$, and $x_i \prec x'_i$. Let $U := \{|x_1, \dots, x_{i-1}|\}$, $Y := \{|x_i, \dots, x_n|\}$, $Y' := \{|x'_i, \dots, x'_m|\}$. Then $X = U \uplus Y$, $X' = U \uplus Y'$ and if $y \in Y$ then $y \preceq x_i \prec x'_i \in Y'$. Furthermore, $x_i \in Y$, $x_i \notin Y'$ so

$Y' \not\subseteq Y, X' \not\subseteq X$. □

Lemma 8.2.11. 1. If (A, \prec) and (B, \prec') are well-founded, so are $(A \times B, \prec_{\text{lex}})$ and $(A_{\text{weakdec}}, \prec_{\text{lex}})$.

2. If (A, \prec) is well-founded so is $(\text{Bag}^{\text{fin}}(A), \prec_{\text{bag}})$.

Proof. (Note: (A, \prec) is anti-reflexive because well-founded).

1. Proof of $(A \times B, \prec_{\text{lex}})$ is well-founded. Assume $(A, \prec), (B, \prec')$ are well-founded. Assume there exists an infinitely descending sequence in $(A \times B, \prec_{\text{lex}})$, i.e. there exists a function $f : \mathbb{N} \rightarrow (A \times B)$. s.t. $\forall n \in \mathbb{N}. f(n+1) \prec_{\text{lex}} f(n)$. Let $f(n+1) = (a_{n+1}, b_{n+1}), f(n) = (a_n, b_n)$ and $(a_{n+1}, b_{n+1}) \prec_{\text{lex}} (a_n, b_n)$, which means $(a_{n+1} \prec a_n) \vee (a_{n+1} = a_n \wedge b_{n+1} \prec' b_n)$. So we have $a_1 \succeq a_2 \succeq a_3 \succeq \dots$. Since $\neg(\exists f' : \mathbb{N} \rightarrow A. \forall n \in \mathbb{N}. f'(n+1) \prec f'(n))$, there exists an n s.t. $a_n = a_{n+1} = a_{n+2} = \dots$. Therefore, $b_n \succ b_{n+1} \succ b_{n+2} \succ \dots$. However, there exists no infinite descending sequence in (B, \prec') , so we get a contradiction.

Proof of $(A_{\text{weakdec}}, \prec_{\text{lex}})$ is well-founded. Assume (A, \prec) is well-founded. We have $A_{\text{weakdec}} := \{(a_0, a_1, a_2, \dots) \mid a_0 \succeq a_1 \succeq a_2 \succeq \dots\}$. We assume that there is an infinite descending sequence. s.t. $a_0 \succ_{\text{lex}} a_1 \succ_{\text{lex}} a_2 \succ_{\text{lex}} \dots$. Let

$$\begin{aligned} a_0 &= (a_1^0, a_2^0, \dots, a_{l_0}^0) \\ a_1 &= (a_1^1, a_2^1, \dots, a_{l_1}^1) \\ a_2 &= (a_1^2, a_2^2, \dots, a_{l_2}^2) \\ a_3 &= (a_1^3, a_2^3, \dots, a_{l_3}^3) \\ &\dots \end{aligned}$$

Since (A, \prec) is well founded and $a_1^0, a_1^1, a_1^2, \dots \in A$, there exists an n_1 s.t. $a_1^{n_1} = a_1^{n_1+1} = a_1^{n_1+2} = \dots$. We have $a_2^{n_1+k}$ always exists. (If a_{n_1+k} had length 1 then $a_{n_1+k} = (a_{n_1}^1)$. Then a_{n_1+k+1} cannot be smaller so $a_2^{n_1+k}$ always exists.) Therefore, $a_2^{n_1} \succeq a_2^{n_1+1} \succeq a_2^{n_1+2} \succeq \dots$. Again, because (A, \prec) is well-founded there exists an $n_2 \geq n_1$ s.t. $a_2^{n_2} = a_2^{n_2+1} = a_2^{n_2+2} = \dots$. As before $a_3^{n_2+k}$ must exist. Therefore, $a_3^{n_2} \succeq a_3^{n_2+1} \succeq a_3^{n_2+2} \succeq \dots$. Continuing this process we obtain natural number $n_1 \leq n_2 \leq n_3 \leq \dots$

s.t. $a_{n_i}^i = a_{n_i+1}^i = a_{n_i+2}^i = \dots$ (Note that $a_i^k \geq a_{i+1}^k$ since $a_i \in A_{\text{weakdec}}$)
 Therefore, we have $a_1^{n_1} = a_1^{n_1+1} = a_1^{n_1+2} = \dots = a_1^{n_2} \succcurlyeq a_2^{n_2} = \dots \succcurlyeq \dots \succcurlyeq a_k^{n_k} = \dots = a_k^{n_{k+1}} \succcurlyeq a_{k+1}^{n_{k+1}} = \dots$, where $a_{n_i} = (a_1^{n_i}, a_2^{n_i}, a_3^{n_i}, \dots) \in A_{\text{weakdec}}$.
 Therefore, we obtain $a_1^{n_1} \succcurlyeq a_2^{n_2} \succcurlyeq a_3^{n_3} \succcurlyeq \dots$, for $a_j^{n_i} \in A$. Because there is no infinite descending sequence in A s.t. $a_1^{n_1} \succcurlyeq a_2^{n_2} \succcurlyeq a_3^{n_3} \succcurlyeq \dots$, there exists a k s.t. $a_1^{n_1} \succcurlyeq a_2^{n_2} \succcurlyeq a_3^{n_3} \succcurlyeq \dots \succcurlyeq a_k^{n_k} = a_{k+1}^{n_{k+1}} = \dots$. Therefore, for n_k the sequences have the form

$$a_l = (a_1^{n_1}, a_2^{n_2}, a_3^{n_3}, \dots, \underbrace{a_k^{n_k}, \dots, a_k^{n_k}}_{m_0})$$

So there exists m_0 s.t. $\forall l \geq n_k. a_l = b_{m_0}$ where

$$b_{m_1} = (a_1^{n_1}, a_2^{n_2}, a_3^{n_3}, \dots, a_k^{n_k}, \underbrace{a_k^{n_k}, \dots, a_k^{n_k}}_{m_1})$$

We have $b_{m_0} \succ_{\text{lex}} b_{m_1}$ if only if $m_0 > m_1$. Therefore, by $a_l = b_{m_0} \succ_{\text{lex}} b_{m_1} = \dots \succ_{\text{lex}} \dots$ there exists an infinite descending sequence $m_0 > m_1 > \dots$ where $m_i \in \mathbb{N}$, so we get a contradiction.

2. Assume (A, \prec) is well-ordered. Then we get $(A_{\text{weakdec}}, \prec_{\text{lex}})$ is well-founded. By Lemma 8.2.10, so is $(\text{Bag}^{\text{fin}}(A), \prec_{\text{bag}})$: assume $X_1 \succ_{\text{bag}} X_2 \succ_{\text{bag}} X_3 \succ_{\text{bag}} \dots$ was an infinitely descending sequence in the multiset ordering. After reordering of each $X_1, X_2, X_3 \dots$, we obtain weakly descending lists: $l_1, l_2, l_3 \dots$ s.t. $l_1 \succ_{\text{lex}} l_2 \succ_{\text{lex}} l_3 \dots$. Since, $(A_{\text{weakdec}}, \prec_{\text{lex}})$ is well-founded, we get a contradiction.

□

8.2.2 Agda Normalises Elements of Algebraic Data Types to Normal Form

We want to show that the normal form of every term which is an algebraic data type starts with a constructor. This won't hold for arbitrary Agda code. A trivial example is as follows: assume postulate $n : \mathbb{N}$ then n is a closed term in normal form of type \mathbb{N} which does not start with a constructor. We will assume some

conditions on the Adaga code, which hold for our Agda code after some minor modifications. The Agda code written by us can be transformed so that it fulfils the restrictions.

8.2.2.1 Global Assumption - Restrictions on Agda Code

We will impose some restrictions on Agda and have Agda code accepted by Agda and generate from it other Agda code which has simple pattern matching. We refer to Agda code which is

- accepted by Agda
- in which no extra switches have been set (see below)
- where some other restrictions (imposed by us and checked by hand) are made

as checked Agda code and to code generated from it as generated Agda code. By Agda code we mean checked or generated Agda code.

By the extra switches we mean: normally in Agda we are allowed have lines

```
{-# OPTIONS -no-termination-check #-}  
{-# OPTIONS -no-coverage-check #-}  
{-# OPTIONS -no-positivity-check #-}
```

After these lines the termination, coverage and positivity checkers are switched off. So by checked Agda code we mean it does not contain such kind of lines.

We make in the following assumptions about checked and generated Agda code:

Assumption 8.2.12. (a) *We assume checked Agda code is strongly normalising (guaranteed by Agda's termination checker).*

(b) *We assume both checked and generated Agda code are confluent (guaranteed by the general setting of Agda).*

Assumption 8.2.13. *We assume in the following the Agda code:*

(a) No record types are used.

(b) No use of *let*- and *where*- expressions.

This assumption is just for simplicity. *let, where* – expression can always be omitted by defining corresponding global definitions. For example, if we have a function f defined as follows :

$$\begin{aligned}
 & f : A \rightarrow B \rightarrow C \\
 & f\ a\ b = t \\
 & \text{where} \\
 & \quad e : E \\
 & \quad e = s \\
 \\
 & \quad g : F \\
 & \quad g = s'
 \end{aligned}$$

it could be replaced by

$$\begin{aligned}
 & \text{mutual} \\
 & \quad e : A \rightarrow B \rightarrow E \\
 & \quad e\ a\ b = s \\
 \\
 & \quad g : A \rightarrow B \rightarrow F \\
 & \quad g\ a\ b = s'[e := e\ a\ b] \\
 \\
 & \quad f : A \rightarrow B \rightarrow C \\
 & \quad f\ a\ b = t[e := e\ a\ b, \quad g := g\ a\ b]
 \end{aligned}$$

We have to rename the functions if "e" and "g" are used before.

Remark: Because of Assumption 8.2.13 we know that the set of terms on Agda can be defined inductively as follows :

- Constants (which include function symbols, constructors) are terms.
- Variables are terms.
- If s, t are terms, then $s\ t$ is a term.

- If s is a term and x is a variable, then $\lambda x.s$ is a term.

Definition 8.2.14. *In the following by an algebraic data type we mean types declared as*

$$\begin{aligned} \text{data } A (a_1 : A_1) \cdots (a_n : A_n) : \text{Set} \quad \text{where} \\ C_1 : (x_1 : B_1^1) \rightarrow \cdots \rightarrow (x_{n_1} : B_{n_1}^1) \rightarrow A a_1 \cdots a_n \\ \dots \\ C_k : (x_k : B_1^k) \rightarrow \cdots \rightarrow (x_{n_k} : B_{n_k}^k) \rightarrow A a_1 \cdots a_n \end{aligned}$$

The result type of each constructor is $A a_1 \cdots a_n$ where $a_1 \cdots a_n$ are parameters.

Remark: $A a_1 \cdots a_n$ is the least set closed under C_1, \dots, C_k . The elements of $A a_1 \cdots a_n$ are $C_i a_1^i \cdots a_{n_i}^i$ for $a_j^i : B_j^i[x_1 := a_1^i, \dots, x_{j_y} := a_{j_y}^i]$. $C_i a_1 \cdots a_{n_i} = C_j b_1 \cdots b_{n_j}$ if only and if $i = j$ and $a_i = b_j$. B_j^i does not depend on A or B_j^i is of the form $(x_1 : D_1) \rightarrow \dots \rightarrow (x_k : D_k) \rightarrow A a_1 \cdots a_n$.

Remark: (Note that $a_1 \cdots a_n$ are just parameters, and C_i do not refer to $A b_1 \cdots b_n$ for $b_1 \neq a_1$ or ... or $b_n \neq a_n$). The canonical identity type (see section 2.1.6) is not an algebraic data type in the above sense. We will treat the canonical identity type as a special type. Furthermore, self-defined equalities can be different. For instance, we will define an equality on \mathbb{N} as follows :

$$\begin{aligned} \text{data Bool} : \text{Set} \quad \text{where} \\ \text{true} : \text{Bool} \\ \text{false} : \text{Bool} \end{aligned}$$

$$\begin{aligned} \text{data } \top : \text{Set} \quad \text{where} \\ \text{triv} : \top \end{aligned}$$

$$\text{data } \perp : \text{Set} \quad \text{where}$$

```
Eq : ℕ → ℕ → Set
Eq zero zero = ⊤
Eq zero (suc y) = ⊥
Eq (suc x) zero = ⊥
Eq (suc x) (suc y) = Eq x y
```

As indicated in Section 2.1.4 \top is the true formula (having a proof `triv`) and \perp is the false formula (having no proof). Then `Eq n m` is an algebraic data type.

Remark: In Definition 8.2.14 we don't allow mutual induction definitions. For instance, the following is not allowed:

```
mutual
  data Even : Set where
    Z : Even
    S : Odd → Even

  data Odd : Set where
    S : Even → Odd
```

neither do we allow indexed inductive definitions. For instance,

```
data A : B → Set where
  C1 : (x : D) → A t
  ⋮
  Ck : ⋯
```

where D is different from B won't occur in our Agda code. Note the difference to parametrised inductive definition. (Indexed inductive definitions do occur, which we discovered close to the submission date of this thesis. See a remark at the end of this section where we discuss how this problem could be fixed.)

Assumption 8.2.15. *We assume that postulated functions in checked and generated Agda code have as result type equalities, postulated types or `Set`.*

(In case of axioms we refer the result type as conclusion) So we allow

$$\text{postulate } f : (x_1 : A_1) \cdots (x_n : A_n) \rightarrow B$$

where B is an equality or a postulated type for all $x_1 : B_1, \dots, x_n : B_n$.

This assumption is necessary. Assume we have a postulated function f defined as follows :

$$\text{postulate } f : \mathbb{N} \rightarrow \mathbb{N}$$

or

$$\text{postulate } n : \mathbb{N}$$

Then $(f \text{ zero})$ and n are normal forms but they do not start with a constructor.

Assumption 8.2.16. *We assume functions defined by case distinction on equalities in checked and generated Agda code have as result type only equalities, postulated types, or Set.*

This assumption is necessary. If we have the function transfer with the following definition

$$\text{data } _ \wedge _ (A B : \text{Set}) : \text{Set} \quad \text{where}$$

$$\text{and} : A \rightarrow B \rightarrow A \wedge B$$

$$A : \mathbb{N} \rightarrow \text{Set}$$

$$A = \dots$$

$$B : \mathbb{N} \rightarrow \text{Set}$$

$$B = \dots$$

$$P : \mathbb{N} \rightarrow \text{Set}$$

$$P n = A n \wedge B n$$

$$\text{postulate } q : \text{zero} == \text{suc zero}$$

$$p : P \text{ zero}$$

$p =$ and

$\text{transfer} : (n, m : \mathbb{N}) \rightarrow (n == m) \rightarrow P\ n \rightarrow P\ m$

$\text{transfer} .n\ n\ \text{refl}\ p = p$

$(\text{transfer}\ \text{zero}\ (\text{suc}\ \text{zero})\ q\ p)$ is in normal form and is an element of $A\ (\text{suc}\ \text{zero}) \wedge B\ (\text{suc}\ \text{zero})$ but it doesn't start with a constructor.

Definition 8.2.17. *By a proper function we mean a function with result data type an algebraic data type. Improper functions are functions which have as result type an equality or a postulate type.*

Definition 8.2.18. 1. *Patterns are defined inductively as follows:*

- *A variable is a pattern.*
- *The absurd pattern $()$ is a pattern.*
- *If C is a constructor, t_1, \dots, t_n are patterns with different variables, then $C\ t_1 \cdots t_n$ is a pattern.*

2. *A proper pattern is a pattern of the form $C\ t_1 \cdots t_n$ (i.e. not a variable or $()$).*

We sometimes need to refer to a pattern which is either a specific variable x or $()$. We write \hat{x} for such a pattern.

We often deal with the situation where we have patterns which may or may not contain the absurd pattern. In case no absurd pattern occurs in a pattern, definition of a function for this pattern has the form $f\ t_1 \cdots t_n$ (e.g. if $t_1 = ()$, $f\ ()\ t_2 \cdots t_n$). Otherwise it has the form $f\ t_1 \cdots t_n = s$. We introduce in the next definition an abbreviation $f\ t_1 \cdots t_n = \hat{s}$ for dealing with this.

Definition 8.2.19. *Let A be some Agda code. Let f be a proper function symbol of an algebraic data type*

$$f : (x_1 : B_1) \rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow A$$

defined by pattern matching. We write

$$f\ t_1 \cdots t_m = \hat{s}$$

for a line of the pattern matching definition of f where

- there is no occurrence of absurd pattern $()$ in $t_1 \cdots t_m$ and the line is $f\ t_1 \cdots t_m = s$. Or
- there is an occurrence of absurd pattern $()$ in $t_1 \cdots t_m$ and the line is $f\ t_1 \cdots t_m = s$ (i.e. $= s$ is omitted).

Definition 8.2.20. 1. Let \mathcal{A} be an Agda code and f be a proper function symbol $f : (x_1 : B_1) \rightarrow (x_2 : B_2) \rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow A$. We define the set of matching trees for f which are finite finitely branching trees with nodes labelled by possible patterns for a definition of a function of this type.

- The tree with only one root node labelled $f\ x_1 \cdots x_n$ is a matching tree.
 - Assume s is a matching tree for f . Let m be a leaf labelled by $f\ t_1 \cdots t_n$. Let y be a variable, which has type an algebraic data type with constructors $C_1 \cdots C_r$ where C_i has n_i arguments and which occurs in $f\ t_1 \cdots t_n$. Let s' be obtained by adding leaves to m with labels $(f\ t_1 \cdots t_n)\ [y := C_i\ z_1 \cdots z_{n_i}]$ for new variables $z_1 \cdots z_{n_i}$ for $i = 1 \cdots k$. Then s' is a matching tree for f .
2. f is coverage complete for \mathcal{A} if there exists a matching tree such that each leaf of the matching tree occurs as a line in the definition of f (subject to renaming of variables and replacing variables, which are elements of an empty data type by the absurd pattern $()$).
3. \mathcal{A} is coverage complete if for all proper functions f we have f is coverage complete.

Note: In the above definition nothing was required for improper functions f . In the following we will not change such functions, so coverage completeness in a more general sense for those symbols (which we have not defined and would be more complicated to define) will automatically be preserved.

Assumption 8.2.21. *Checked Agda code A is coverage complete (guaranteed by Agda’s coverage checker).*

In Agda it is possible to give two overlapping patterns. For instance we can define

$$\begin{aligned} f &: \mathbb{N} \rightarrow \text{Bool} \\ f \text{ zero} &= \text{true} \\ f x &= \text{false} \end{aligned}$$

Agda resolves this conflict by using the equations in order: Agda will evaluate $(f t)$ by reducing t first to head normal form. If t reduces to zero and therefore $(f t)$ matches the first equation, then $(f t)$ evaluates to true. If t reduces to head normal form starting with a different constructor, $(f t)$ will not match the first equation but the second, which will then be used, and $(f t)$ reduces to false.

This causes problems since this means that the equations in the first version cannot be treated as judgements of Agda in type theory, one needs a concept of conditional judgements, which is conceptionally difficult, something of the form “if $x \neq \text{zero}$ then $f x = \text{false}$ ”. This causes in particular problems since we want to add later additional equations to an expanded version of Agda, which create additional judgements but no additional reductions. We can avoid the problem of overlapping patterns by observing that in this example Agda behaves as if we had the following definition

$$\begin{aligned} f &: \mathbb{N} \rightarrow \text{Bool} \\ f \text{ zero} &= \text{true} \\ f (\text{suc } y) &= \text{false} \end{aligned}$$

Agda would use the same reduction method in case of the first and of the second code.

Therefore our approach is to use this method systematically and resolve all overlapping pattern matchings: If there is an equation, which overlaps with a previous one, one investigates, which minimal expansion of the pattern matching is necessary, so that the second equation excludes the case of the overlapping pattern matching. This might have the effect of replacing an equation by several

new ones. For instance if we have

$$\begin{aligned}
 &\text{data } A : \text{Set where} \\
 &\quad C_1 : B \rightarrow A \\
 &\quad C_2 : B' \rightarrow A \\
 &\quad C_3 : A \rightarrow A \\
 \\
 &f : A \rightarrow A \rightarrow \text{Set} \\
 &f \ x \ (C_3 \ (C_1 \ y)) = t_1 \\
 &f \ x \ y = t_2
 \end{aligned}$$

then one resolves the overlapping of the second equation with the first one by replacing the definition by the following:

$$\begin{aligned}
 f \ x \ (C_3 \ (C_1 \ y)) &= t_1 \\
 f \ x \ (C_1 \ y) &= t_2[y := C_1 \ y] \\
 f \ x \ (C_2 \ y) &= t_2[y := C_2 \ y] \\
 f \ x \ (C_3 \ (C_2 \ y)) &= t_2[y := C_3 \ (C_2 \ y)] \\
 f \ x \ (C_3 \ (C_3 \ y)) &= t_2[y := C_3 \ (C_3 \ y)]
 \end{aligned}$$

When checking the Agda code of this thesis used for program extraction we didn't detect any overlapping patterns, but might have overlooked some. We assume that such a case could be resolved by using this technique. Since we discovered this problem at a very late stage we are not proving a general theorem how to resolve it in general. We assume that all overlapping patterns have been resolved and make the following assumption:

Assumption 8.2.22. *We assume that in checked and generated Agda code, when defining functions by pattern matching all patterns are disjoint, e.g. it is never the case that there are terms t_1, \dots, t_n (possibly containing free variables) such that $(f \ t_1 \ \dots \ t_n)$ matches two different patterns*

$$\begin{aligned}
 f \ s_1 \ \dots \ s_n &= t \\
 f \ s'_1 \ \dots \ s'_n &= t'
 \end{aligned}$$

Therefore the order of equations doesn't matter.

In Agda we can define sets by case distinction

```
Atom : Bool → Set
Atom true  = ⊤
Atom false = ⊥
```

If we allowed arbitrary equations we could define for an algebraic data type A and a postulated type B

```
C : Bool → Set
C true = A
C false = B
```

```
postulate p : B
```

If we had an inconsistency in the Agda code, this might allow us to prove $\text{true} = \text{false} : \text{Bool}$. Then $A = C \text{ true} = C \text{ false} = B$ and therefore $p : A$, which doesn't reduce to head normal form.

Furthermore, in the same situation we would get $\top = \text{Atom true} = \text{Atom false} = \perp$, so $\text{triv} : \perp$. If we define

```
f : ⊥ → ℕ
f ()
```

then $f \text{ triv}$ doesn't reduce to head normal form.

Therefore, we need to make sure that:

- Agda doesn't prove that two different algebraic data types are the same: If we had for instance $\top = \perp : \text{Set}$ then we would get $\text{triv} : \perp$. If we then define $g : \perp \rightarrow \mathbb{N}$ by the empty pattern matching $g ()$, then $g \text{ triv}$ would be an element of the algebraic data type \mathbb{N} which doesn't reduce to head normal form.
- Agda doesn't prove that an algebraic data type A and a postulated type or equation B are the same. Otherwise we could postulate $p : B$, and would get $p : A$, p doesn't reduce to head normal form.

- Agda doesn't equate two different constructor patterns. Otherwise we would obtain overlapping constructor patterns.
- Agda doesn't prove that $t : A$ for an algebraic data type A where t starts with a constructor which is not a constructor of A . If we had for instance an algebraic data type $A : \text{Set}$ with only constructor $C : A$, defined $f : A \rightarrow \mathbb{N}$ by $f C = \text{zero}$. If we had $C' : A$ for some different constructor, then $(f C')$ wouldn't be an element of \mathbb{N} which doesn't reduce to head normal form,

All these assumptions hold in the Agda code of this thesis used for program extraction: Agda checks whether two types and terms are the same, by reducing them to constructor normal form. A postulated set or algebraic data type only reduces to itself, so two different algebraic data types will never be the same, neither will be an algebraic and a non-algebraic data type. Furthermore type checking equalities will never equate different constructors, and will not show that an element of an algebraic data type has a different constructor.

We make those assumptions explicit, since we will later expand our Agda code by additional equations, and we need to make sure that we never obtain Agda code which violates those assumptions.

Definition 8.2.23. (Consistency of Agda) *Let \mathcal{A} be Agda code. \mathcal{A} is consistent if it fulfils the following conditions:*

- (a) *We assume that if \mathcal{A} proves $A = B : \text{Set}$, where both A and B are algebraic data types, postulated types or dependent function types, then if one is an algebraic data type, the other is the algebraic data type given by the same definition; if one is a postulated type or equality, the other is as well a postulated type or equality; if one is of the form $(x_1 : A_1) \rightarrow B_1$, the other is modulo α -conversion of the form $(x_1 : A'_1) \rightarrow B'_1$, and we have $A_1 = A'_1$ and for $x_1 : A_1$ that $B_1 = B'_1$.*
- (b) *We assume that \mathcal{A} proves $C t_1 \cdots t_n : B$ for an algebraic data type for a constructor C and terms t_i (depending on a context Γ), then C is a constructor of B , and, if it has type $(x_1 : B_1) \rightarrow \cdots (x_n : B_n) \rightarrow B$, then $t_i : B_i[x_1 := t_1, \dots, x_{i-1} := t_{i-1}]$.*

(c) We assume that if \mathcal{A} doesn't prove $C t_1 \cdots t_n = C' t'_1 \cdots t'_k : A$ for different constructors C, C' and terms t_i (depending on a context Γ), and if it proves $C t_1 \cdots t_n = C t'_1 \cdots t'_n : A$, then it proves that the arguments are equal elements of the corresponding types.

Assumption 8.2.24. *We assume that checked Agda code is consistent.*

8.2.2.2 Pattern Matching Can Be Restricted to Simple Patterns

Pattern matching can be very complicated if a function does pattern matching on several arguments which might be even deeper nested. For instance, the proof of transitivity on natural number is defined as follows (note $==$ is not the canonical equality but a special equality defined for natural number which is an algebraic data type)

```

transitive : (n m l : ℕ) → n == m → m == l → n == l
transitive zero zero zero p p' = p
transitive zero zero (S l) p ()
transitive zero (S m) l () p'
transitive (S n) zero l () p'
transitive (S n) (S m) zero p ()
transitive (S n) (S m) (S l) p p' = transitive n m l p p'

```

We will show in the following that any Agda code can be replaced by Agda code which uses only simple pattern matching on one argument which is not the empty type. Therefore, we introduce the notion of simple pattern matching, which is pattern matching only on one type argument each time and without any nested patterns.

There is one problem, namely that we sometimes lose definitional equalities. For instance, the Agda code \mathcal{A}

```

f : ℕ → ℕ → ℕ
f x 0 = 0
f 0 (suc y) = suc y
f (suc x) (suc y) = suc (suc y)

```

will be replaced by Agda code \mathcal{A}' (note that 0 can be used for the constructor zero even in patterns)

$$\begin{aligned}
 f &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
 f \ 0 \ 0 &= 0 \\
 f \ (\text{suc } x) \ 0 &= 0 \\
 f \ 0 \ (\text{suc } y) &= \text{suc } y \\
 f \ (\text{suc } x) \ (\text{suc } y) &= \text{suc } (\text{suc } y)
 \end{aligned}$$

which in a third step will be replaced by Agda code \mathcal{A}''

$$\begin{aligned}
 &\text{mutual} \\
 &f : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
 &f \ 0 \ x = e \ x \\
 &f \ (\text{suc } x) \ y = e' \ x \ y \\
 \\
 &e : \mathbb{N} \rightarrow \mathbb{N} \\
 &e \ 0 = 0 \\
 &e \ (\text{suc } y) = \text{suc } y \\
 \\
 &e' : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
 &e' \ x \ 0 = 0 \\
 &e' \ x \ (\text{suc } y) = \text{suc } (\text{suc } y)
 \end{aligned}$$

In \mathcal{A} we have $f \ x \ 0 = 0$ but in \mathcal{A}' we don't have this. This causes problems since the language changes. In \mathcal{A} we have

$$\lambda B \lambda n \lambda x . x : (B : \mathbb{N} \rightarrow \text{Set}) \rightarrow (n : \mathbb{N}) \rightarrow B \underbrace{(f \ n \ 0)}_{=0} \rightarrow B \ 0$$

but this term doesn't have this type in \mathcal{A}' , since $(f \ n \ 0)$ and 0 are not definitionally equal (see Section 2.1.6). We note that this is only a problem for type checking. We will show later that if in \mathcal{A}'' a term, which is an element of algebraic data types, reduces to head normal form, then the same applies to the same term in \mathcal{A} . So if we can prove that Agda terms with simple pattern matching for reductions

have the property that every element of an algebraic data type reduces to head normal form, the same applies to \mathcal{A} provided all elements of algebraic data types in \mathcal{A} are elements of the same algebraic data types in \mathcal{A}'' . In order to obtain the latter property we need to extend \mathcal{A}'' by additional equalities (in the example $f\ x\ 0 = 0$) which are not used for normalisation but only for type checking.

Definition 8.2.25. *Extended Agda code is Agda code plus some additional definitional equalities. These definitional equalities are used for type checking only. When computing the normal form of a term only the original rules are used.*

An example would be

$$\begin{aligned} f &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ f\ 0\ 0 &= 0 \\ f\ (\text{suc } x)\ 0 &= 0 \\ f\ 0\ (\text{suc } y) &= \text{suc } 0 \\ f\ (\text{suc } x)\ (\text{suc } y) &= \text{suc } (\text{suc } 0) \\ \text{extended by} \\ f\ x\ 0 &= 0 \end{aligned}$$

This means that in type checking we add the axiom

$$x : \mathbb{N} \implies f\ x\ 0 = 0 : \mathbb{N}$$

However, we don't have the reduction rule

$$f\ x\ 0 \longrightarrow 0$$

In the following all Agda code will be extended Agda code unless specified differently. We write Agda code for extended Agda code and non-extended Agda code for original Agda code.

Definition 8.2.26. *1. An Agda code has simple pattern matching, if, whenever pattern matching for a function f on a directly non-empty algebraic data type occurs, then this function does pattern matching only on one ar-*

gument, and the pattern is non nested. So the pattern is

$$\begin{aligned}
 f &: (x_1 : B_1) \rightarrow (x_2 : B_2) \rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow A \\
 f \hat{x}_1 \cdots \hat{x}_{i-1} (C_1 \hat{y}_1^1 \cdots \hat{y}_{l_1}^1) \hat{x}_{i+1} \cdots \hat{x}_m &= \hat{t}_1 \\
 \cdots & \\
 f \hat{x}_1 \cdots \hat{x}_{i-1} (C_k \hat{y}_1^k \cdots \hat{y}_{l_k}^k) \hat{x}_{i+1} \cdots \hat{x}_m &= \hat{t}_k
 \end{aligned}$$

And we require that in column i there is no occurrence of pattern x consisting of a variable, so we won't have $f \hat{x}_1 \cdots \hat{x}_{i-1} x_i \hat{x}_{i+1} \cdots \hat{x}_m = \hat{t}$. (Additional equalities added are not considered here, they can be arbitrary.)

2. Agda code has the head normal form property, if every closed normal term which is an element of an algebraic data type starts with a constructor.
3. \mathcal{A}' extends \mathcal{A} if all judgements derivable in \mathcal{A} are derivable in \mathcal{A}' as well.
4. Assume Agda code \mathcal{A}' extends \mathcal{A} . \mathcal{A}' induces the head normal form property on \mathcal{A} if whenever B is an algebraic data type, \mathcal{A} proves $t : B$ and t has in \mathcal{A}' a normal form starting with a constructor then t has in \mathcal{A} a normal form starting with the same constructor.

Remark: If 3 and 4 hold, \mathcal{A} , \mathcal{A}' are normalising, \mathcal{A}' has the head normal form property, so does \mathcal{A} . In order to prove this assume \mathcal{A} proves $t : B$ where B is an algebraic data type. Then \mathcal{A}' proves as well $t : B$. \mathcal{A}' has the head normal form property, therefore t starts with a constructor in \mathcal{A} . But then by 4 t has in \mathcal{A} a normal form starting with the same constructor.

We are going to transform Agda code without simple pattern matching into one with simple pattern matching. This is done in several steps from Agda code \mathcal{A}_0 to \mathcal{A}_1 to \dots to \mathcal{A}_n in such way that \mathcal{A}_{i+1} induces head normal form property on \mathcal{A}_i . Therefore, \mathcal{A}_n induces it on \mathcal{A}_0 .

We give an example of the transformation for the subtraction function for natural numbers in \mathcal{A} :

$$\begin{aligned} & _-_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ & m \text{ - zero} = m \\ & \text{zero - (suc } n) = \text{zero} \\ & (\text{suc } m) \text{ - (suc } n) = m \text{ - } n \end{aligned}$$

In \mathcal{A}_1 we first make sure that all lines make case distinction on the first argument:

$$\begin{aligned} & _-_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ & \text{zero - zero} = \text{zero} \\ & (\text{suc } m) \text{ - zero} = \text{suc } m \\ & \text{zero - (suc } n) = \text{zero} \\ & (\text{suc } m) \text{ - (suc } n) = m \text{ - } n \end{aligned}$$

In \mathcal{A}_2 we reorder lines:

$$\begin{aligned} & _-_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ & \text{zero - zero} = \text{zero} \\ & \text{zero - (suc } n) = \text{zero} \\ & (\text{suc } m) \text{ - zero} = \text{suc } m \\ & (\text{suc } m) \text{ - (suc } n) = m \text{ - } n \end{aligned}$$

In \mathcal{A}_3 $_-_$ function only makes simple pattern matching on the first argument and delegates the cases to new functions e and f :

$$\begin{aligned} & \text{mutual} \\ & _-_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ & \text{zero - } n = e \text{ } n \\ & (\text{suc } m) \text{ - } n = f \text{ } m \text{ } n \end{aligned}$$

$$\begin{aligned}
 e &: \mathbb{N} \rightarrow \mathbb{N} \\
 e \text{ zero} &= \text{zero} \\
 e (\text{suc } n) &= \text{zero}
 \end{aligned}$$

$$\begin{aligned}
 f &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
 f \ m \ \text{zero} &= \text{suc } m \\
 f \ m \ (\text{suc } n) &= m - n
 \end{aligned}$$

Definition 8.2.27. Let \mathcal{A} , \mathcal{A}' be two Agda codes. We define properties (Subset), (HNF), (m), (c), (n) and (cons) between \mathcal{A} , \mathcal{A}' as follows:

(Subset) \mathcal{A}' extends \mathcal{A} .

(HNF) \mathcal{A}' induces the head normal form property on \mathcal{A} .

(c) If \mathcal{A} is coverage complete, so is \mathcal{A}' (we don't take into account extra equalities).

(n) If \mathcal{A} is strongly normalising, so is \mathcal{A}' (we don't take into account extra equalities).

(cons) If \mathcal{A} is consistent, so is \mathcal{A}' .

Theorem 8.2.28. Assume Agda code \mathcal{A} . Then we can define Agda code \mathcal{A}' has the same algebraic data types, postulated functions and types, the same function symbols (with different pattern matching rules) and some additional ones defined by pattern matching on algebraic data types, such that the following holds:

- \mathcal{A} and \mathcal{A}' fulfil (Subset), (HNF), (c), (n) and (cons).
- \mathcal{A}' has simple pattern matching.

The **proof** of Theorem 8.2.28 will be given in the following.

Definition 8.2.29. For a pattern t we define its length $|t| \in \mathbb{N}$ as follows :

- $|x| := |()| := 0$.
- $|C \ t_1 \cdots t_n| := 1 + |t_1| + \cdots + |t_n|$.

Definition 8.2.30. *Let \mathcal{A} be some Agda code. Let f be a proper function symbol defined by pattern matching on an algebraic data type with definition*

$$\begin{aligned} f &: (x_1 : B_1) \rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow A \\ f \ t_1^1 \cdots t_m^1 &= \hat{s}_1 \\ \cdots \\ f \ t_1^k \cdots t_m^k &= \hat{s}_k \end{aligned}$$

and some additional equalities (which don't count towards the measure). We define the following measure

$$m^{\mathcal{A}}(f) := \begin{cases} 0 & \text{the pattern matching for } f \text{ is simple} \\ \sum_{i,j} |t_j^i| & \text{otherwise} \end{cases}$$

Definition 8.2.31. *Assume some Agda code \mathcal{A} . We define its measure $m(\mathcal{A})$ as*

$$\{|m^{\mathcal{A}}(f) \mid f \text{ function symbol is defined by pattern matching in } \mathcal{A}|\}$$

Note that $\{|a_1, \dots, a_n|\}$ was our notation for bags.

Remark 8.2.32. *If f is defined by pattern matching then this pattern matching is simple if and only if $m^{\mathcal{A}}(f) = 0$.*

Lemma 8.2.33. *Assume \mathcal{A} has no simple pattern matching. Then there exists an Agda code \mathcal{A}' such that the following holds*

1. \mathcal{A} and \mathcal{A}' fulfil (Subset), (HNF), (c), (n) and (cons).
2. $m(\mathcal{A}') \prec_{\text{bag}} m(\mathcal{A})$, where \prec_{bag} is the ordering on bags.

We will transform the Agda code from \mathcal{A} to \mathcal{A}' on steps. First, several steps 1 are carried out followed by one instance of step 2. So the transformation is going from $\mathcal{A} = \mathcal{A}_0$ to \mathcal{A}_1 to \mathcal{A}_2 to \cdots to \mathcal{A}_n which is \mathcal{A}' . When carrying out step 1 or step 2 we will refer to \mathcal{A} as the Agda code before the transformation (ie. $\mathcal{A} = \mathcal{A}_i$) and \mathcal{A}' for the Agda code after the transformation (ie. $\mathcal{A}' = \mathcal{A}_{i+1}$).

In step 1 (see below) we assume \mathcal{A} and define \mathcal{A}' s.t. properties (Subset), (HNF), (c), (n) and (cons) hold. In step 2 (see below) we assume \mathcal{A} to which

step 1 cannot be applied anymore and define \mathcal{A}' s.t. properties (Subset), (HNF), (c), (n) and (cons) hold. Finally we show $m(\mathcal{A}_n) \prec_{\text{bag}} m(\mathcal{A}_0)$. Note that because $\mathcal{A}_i, \mathcal{A}_{i+1}$ fulfil (Subset), (HNF), (c), (n) and (cons) for all i , so do $\mathcal{A} = \mathcal{A}_0$ and $\mathcal{A}' = \mathcal{A}_n$.

Proof of Lemma 8.2.33 and therefore of Theorem 8.2.28: Let f be a proper function symbol defined by a non simple pattern matching on an algebraic data type. Let f be defined by

$$\begin{aligned} f &: (x_1 : B_1) \rightarrow \dots \rightarrow (x_n : B_n) \rightarrow A \\ f \ t_1^1 \dots t_m^1 &= \hat{s}_1 \\ \dots & \\ f \ t_1^k \dots t_m^k &= \hat{s}_k \end{aligned}$$

Let i be minimal such that there exists t_i^j for some j , which is a proper pattern, $t_i^j = C \ s_1 \dots s_k$. So $t_{i'}^j$ are improper patterns for $i' < i$. W.l.o.g. $j = 1$. By renaming improper patterns we can assume that for each $i' < i$, $t_{i'}^j$ are improper pattern $\hat{x}_{i'}^j$ such that if $\hat{x}_{i'}^j$ and $\hat{x}_{i'}^{j'}$ are variables they are the same variable $x_{i'}$ (as on the type of f). If $\hat{x}_{i'}^j = y$ and $\hat{x}_{i'}^{j'} = z$ then $y = z$. So the definition of f is

$$\begin{aligned} f &: (x_1 : B_1) \rightarrow \dots \rightarrow (x_n : B_n) \rightarrow A \\ f \ \hat{x}_1^1 \dots \hat{x}_{i-1}^1 \ t_i^1 \dots t_m^1 &= \hat{s}_1 \\ \dots & \\ f \ \hat{x}_1^k \dots \hat{x}_{i-1}^k \ t_i^k \dots t_m^k &= \hat{s}_k \end{aligned}$$

We will carry the transformation out in two steps. Firstly, if we have a variable pattern in column i , then we carry out step 1 (see below) several times until all patterns in column i start with a constructor (we must have a pattern which starts with a constructor). This transformation will take place finitely many times since such variable patterns in column i are limited and no new lines having a variable in column i will be introduced in step 1. Then we will carry out step 2. As said before, in intermediate steps (Subset), (HNF), (c), (n) and (cons) are guaranteed. And we will see that if \mathcal{A}' is the Agda code after step 1 and step 2 have been carried out, then $m(\mathcal{A}') < m(\mathcal{A})$.

Step 1: There is a j s.t. $t_i^j = \hat{x}$ is an improper pattern. There exists an i s.t. $t_i^j = C s_1 \cdots s_k$. We renumber the rows in the pattern, so that $t_i^1 = x$, $t_i^2 = C s_1 \cdots s_k$. B_i cannot depend properly on variables which are instantiated by $()$, therefore $B_i[x_1 := \hat{x}_1^1, \dots, x_{i-1} := \hat{x}_{i-1}^1]$ is the same as $B_i[x_1 := \hat{x}_1^2, \dots, x_{i-1} := \hat{x}_{i-1}^2]$. $B_i[x_1 := \hat{x}_1^2, \dots, x_{i-1} := \hat{x}_{i-1}^2]$ is a directly non-empty algebraic data type because t_i^2 starts with a constructor. Therefore, $B_i[x_1 := \hat{x}_1^1, \dots, x_{i-1} := \hat{x}_{i-1}^1]$ is a directly non-empty algebraic data type and $t_i^j = x$. Let B_i have constructors with arguments $(C_1 y_1^1 \cdots y_{l_1}^1), \dots, (C_r y_1^r \cdots y_{l_r}^r)$, where the variables don't occur in any of the patterns, and w.l.o.g. $C_1 = C$. So in code \mathcal{A} the definition of f is

$$\begin{aligned}
 f : (x_1 : B_1) &\rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow A \\
 f \hat{x}_1^1 \cdots \hat{x}_{i-1}^1 x t_{i+1}^1 \cdots t_m^1 &= \hat{s}_1 \\
 f \hat{x}_1^2 \cdots \hat{x}_{i-1}^2 (C_1 s_1 \cdots s_k) t_{i+1}^2 \cdots t_m^2 &= \hat{s}_2 \\
 f \hat{x}_1^3 \cdots \hat{x}_{i-1}^3 t_i^3 t_{i+1}^3 \cdots t_m^3 &= \hat{s}_3 \\
 \dots & \\
 f \hat{x}_1^k \cdots \hat{x}_{i-1}^k t_1^k t_{i+1}^k \cdots t_m^k &= \hat{s}_k
 \end{aligned}$$

Replace this definition by

$$\begin{aligned}
 f : (x_1 : B_1) &\rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow A \\
 f \hat{x}_1^1 \cdots \hat{x}_{i-1}^1 (C_1 y_1^1 \cdots y_{l_1}^1) t_{i+1}^1 \cdots t_m^1 &= \hat{s}_1[x := C_1 y_1^1 \cdots y_{l_1}^1] \\
 \dots & \\
 f \hat{x}_1^1 \cdots \hat{x}_{i-1}^1 (C_r y_1^r \cdots y_{l_r}^r) t_{i+1}^1 \cdots t_m^1 &= \hat{s}_1[x := C_r y_1^r \cdots y_{l_r}^r] \\
 f \hat{x}_1^2 \cdots \hat{x}_{i-1}^2 (C_1 s_1 \cdots s_k) t_{i+1}^2 \cdots t_m^2 &= \hat{s}_2 \\
 f \hat{x}_1^3 \cdots \hat{x}_{i-1}^3 t_i^3 t_{i+1}^3 \cdots t_m^3 &= \hat{s}_3 \\
 \dots & \\
 f \hat{x}_1^k \cdots \hat{x}_{i-1}^k t_1^k t_{i+1}^k \cdots t_m^k &= \hat{s}_k
 \end{aligned}$$

If a line has more than one absurd pattern $()$ then the later ones can be replaced by variables.

If

$$\begin{aligned}
 f t_1 \cdots t_k \\
 f t'_1 \cdots t'_k
 \end{aligned}$$

are identical except for having variables versus absurd pattern $()$, for instance, the occurrence of a variable x in $t_{i'}$ which occurs in t'_i as $()$. Then we can delete the second of the same lines.

We add all additional equalities added to f originally and

$$f \hat{x}_1^1 \cdots \hat{x}_{i-1}^1 x t_{i+1}^1 \cdots t_m^1 = \hat{s}_1$$

which is a line in the original code as an extra equality, provided we didn't have an absurd pattern (i.e. provided \hat{s}_1 is a term). Note that this new equation was a judgement in \mathcal{A} because we have non-overlapping patterns. \mathcal{A}' fulfils Assumption 8.2.24 since all judgements provable in \mathcal{A}' are provable in \mathcal{A} as well. It obviously has no overlapping patterns.

For \mathcal{A} and \mathcal{A}' as transformed in one step we have (Subset), (HNF), (c), (n) and (cons). Proof for (Subset) is trivial since all equality rules in \mathcal{A} are equality rules in \mathcal{A}' . **Proof for (HNF)** Assume \mathcal{A}' has the head normal form property. If t is a term which is an element of a directly non-empty algebraic data type in \mathcal{A}' s.t. $t \longrightarrow^* C t_1 \cdots t_k$ in \mathcal{A}' then all reductions in \mathcal{A}' are reductions in \mathcal{A} . Therefore, $t \longrightarrow^* C t_1 \cdots t_k$ in \mathcal{A} as well with the same constructor.

Proof for (c) A matching tree for f in \mathcal{A}' is obtained from a matching tree for \mathcal{A} by adding to leaves with label $f x_1 \cdots x_{i-1} x_i t'_{i+1} \cdots t'_m$ subnodes

$$\begin{aligned} & f x_1 \cdots x_{i-1} (C_1 y_1^1 \cdots y_{l_1}^1) t'_{i+1} \cdots t'_m \\ & \dots \\ & f x_1 \cdots x_{i-1} (C_r y_1^r \cdots y_{l_r}^r) t'_{i+1} \cdots t'_m \end{aligned}$$

Proof for (n) Assume \mathcal{A} is normalising which means there is no infinite reduction sequences in \mathcal{A} s.t. $s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow \dots$. Then there is no infinite reduction sequences in \mathcal{A}' because all reductions in \mathcal{A}' are reductions in \mathcal{A} . **Proof for (cons)** is trivial since \mathcal{A} proves $t = t' : B$ if only and if \mathcal{A}' proves $t = t' : B$.

Step 2 We assume all t_i^j have the form $C s_1, \dots, s_k$. As before B_i must be a directly non-empty algebraic data type for all $\hat{x}_1^j, \dots, \hat{x}_{i-1}^j$. Let the constructors of $B_i(x_1, \dots, x_k)$ be $(C_1 y_1^1 \cdots y_{l_1}^1), \dots, (C_r y_1^r \cdots y_{l_r}^r)$, where the variables don't

occur in any of the patterns, and we have

$$\begin{aligned} C_1 &: (y_1^1 : D_1^1) \rightarrow \cdots \rightarrow (y_{l_1}^1 : D_{l_1}^1) \rightarrow B \\ &\cdots \\ C_r &: (y_1^r : D_1^r) \rightarrow \cdots \rightarrow (y_{l_r}^r : D_{l_r}^r) \rightarrow B \end{aligned}$$

After renumbering, we can assume that the first patterns refer to C_1 , then the next ones to C_2 etc. So by giving different names to the terms we have

$$\begin{aligned} f &: (x_1 : B_1) \rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow A \\ f \hat{x}_1^{1,1} \cdots \hat{x}_{i-1}^{1,1} (C_1 s_1^{1,1} \cdots s_{l_1}^{1,1}) t_{i+1}^{1,1} \cdots t_m^{1,1} &= \hat{s}_{1,1} \\ &\cdots \\ f \hat{x}_1^{1,j_1} \cdots \hat{x}_{i-1}^{1,j_1} (C_1 s_1^{1,j_1} \cdots s_{l_1}^{1,j_1}) t_{i+1}^{1,j_1} \cdots t_m^{1,j_1} &= \hat{s}_{1,j_1} \\ f \hat{x}_1^{2,1} \cdots \hat{x}_{i-1}^{2,1} (C_2 s_1^{2,1} \cdots s_{l_2}^{2,1}) t_{i+1}^{2,1} \cdots t_m^{2,1} &= \hat{s}_{2,1} \\ &\cdots \\ f \hat{x}_1^{2,j_2} \cdots \hat{x}_{i-1}^{2,j_2} (C_2 s_1^{2,j_2} \cdots s_{l_2}^{2,j_2}) t_{i+1}^{2,j_2} \cdots t_m^{2,j_2} &= \hat{s}_{2,j_2} \\ &\cdots \\ f \hat{x}_1^{r,1} \cdots \hat{x}_{i-1}^{r,1} (C_r s_1^{r,1} \cdots s_{l_r}^{r,1}) t_{i+1}^{r,1} \cdots t_m^{r,1} &= \hat{s}_{r,1} \\ &\cdots \\ f \hat{x}_1^{r,j_r} \cdots \hat{x}_{i-1}^{r,j_r} (C_r s_1^{r,j_r} \cdots s_{l_r}^{r,j_r}) t_{i+1}^{r,j_r} \cdots t_m^{r,j_r} &= \hat{s}_{r,j_r} \end{aligned}$$

We have for all $x_1 \cdots x_{i-1}$ that $(x_i : B_i) \times (x_{i+1} : B_{i+1}) \times \cdots \times (x_n : B_n)$ is covered by

$$\begin{aligned} &(C_1 s_1^{1,1} \cdots s_{l_1}^{1,1}) \quad t_{i+1}^{1,1} \cdots t_m^{1,1} \\ &\cdots \\ &(C_1 s_1^{1,j_1} \cdots s_{l_1}^{1,j_1}) \quad t_{i+1}^{1,j_1} \cdots t_m^{1,j_1} \\ &(C_2 s_1^{2,1} \cdots s_{l_2}^{2,1}) \quad t_{i+1}^{2,1} \cdots t_m^{2,1} \\ &\cdots \\ &(C_r s_1^{r,1} \cdots s_{l_r}^{r,1}) \quad t_{i+1}^{r,1} \cdots t_m^{r,1} \\ &\cdots \\ &(C_r s_1^{r,j_r} \cdots s_{l_r}^{r,j_r}) \quad t_{i+1}^{r,j_r} \cdots t_m^{r,j_r} \end{aligned}$$

We redefine \mathcal{A} to obtain \mathcal{A}' by adding new function symbols g_1, \dots, g_r and new rules for f as follows:

mutual

$$\begin{aligned}
 f &: (x_1 : B_1) \rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow A \\
 f \ x_1 \cdots x_{i-1} \ (C_1 \ y_1^1 \cdots y_{l_1}^1) \ x_{i+1} \cdots x_m &= g_1 \ x_1 \cdots x_{i-1} \ y_1^1 \cdots y_{l_1}^1 \ x_{i+1} \cdots x_m \\
 \dots & \\
 f \ x_1 \cdots x_{i-1} \ (C_r \ y_1^r \cdots y_{l_r}^r) \ x_{i+1} \cdots x_m &= g_r \ x_1 \cdots x_{i-1} \ y_1^r \cdots y_{l_r}^r \ x_{i+1} \cdots x_m
 \end{aligned}$$

$$\begin{aligned}
 g_1 &: (x_1 : B_1) \rightarrow \cdots \rightarrow (x_{i-1} : B_{i-1}) \\
 &\rightarrow (y_1^1 : D_1^1) \rightarrow \cdots \rightarrow (y_{l_1}^1 : D_{l_1}^1) \\
 &\rightarrow (x_{i+1} : B_{i+1}[x_i := C_1 \ y_1^1 \cdots y_{l_1}^1]) \rightarrow \cdots \rightarrow (x_n : B_n[x_i := C_1 \ y_1^1 \cdots y_{l_1}^1]) \\
 &\rightarrow A[x_i := C_1 \ y_1^1 \cdots y_{l_1}^1] \\
 g_1 \ \hat{x}_1^{1,1} \cdots \hat{x}_{i-1}^{1,1} \ s_1^{1,1} \cdots s_{l_1}^{1,1} \ t_{i+1}^{1,1} \cdots t_m^{1,1} &= \hat{s}_{1,1} \\
 \dots & \\
 g_1 \ \hat{x}_1^{1,j_1} \cdots \hat{x}_{i-1}^{1,j_1} \ s_1^{1,j_1} \cdots s_{l_1}^{1,j_1} \ t_{i+1}^{1,j_1} \cdots t_m^{1,j_1} &= \hat{s}_{1,j_1}
 \end{aligned}$$

$$\begin{aligned}
 g_r &: (x_1 : B_1) \rightarrow \cdots \rightarrow (x_{i-1} : B_{i-1}) \\
 &\rightarrow (y_1^r : D_1^r) \rightarrow \cdots \rightarrow (y_{l_r}^r : D_{l_r}^r) \\
 &\rightarrow (x_{i+1} : B_{i+1}[x_i := C_r \ y_1^r \cdots y_{l_r}^r]) \rightarrow \cdots \rightarrow (x_n : B_n[x_i := C_r \ y_1^r \cdots y_{l_r}^r]) \\
 &\rightarrow A[x_i := C_r \ y_1^r \cdots y_{l_r}^r] \\
 g_r \ \hat{x}_1^{r,1} \cdots \hat{x}_{i-1}^{r,1} \ s_1^{r,1} \cdots s_{l_r}^{r,1} \ t_{i+1}^{r,1} \cdots t_m^{r,1} &= \hat{s}_{r,1} \\
 \dots & \\
 g_r \ \hat{x}_1^{r,j_r} \cdots \hat{x}_{i-1}^{r,j_r} \ s_1^{r,j_r} \cdots s_{l_r}^{r,j_r} \ t_{i+1}^{r,j_r} \cdots t_m^{r,j_r} &= \hat{s}_{r,j_r}
 \end{aligned}$$

We take as extra equalities in \mathcal{A}' all extra equalities in \mathcal{A} . \mathcal{A}' obviously has no overlapping patterns. \mathcal{A}' fulfils Assumption 8.2.24: If \mathcal{A}' proves $t = t' : A$, $t : A$, $A : \text{Set}$ or $A = A' : \text{Set}$, then \mathcal{A} proves the same judgements, if we replace every occurrence of $(g_i \ t_1 \cdots t_{i-1} \ s_1 \cdots s_{l_i} \ t_{i+1} \cdots t_n)$ by $(f_i \ t_1 \cdots t_{i-1} \ (C_i \ s_1 \cdots s_{l_i}) \ t_{i+1} \cdots t_n)$. Therefore Assumption 8.2.24 for \mathcal{A}' follows from this assumption about \mathcal{A} .

For \mathcal{A} and \mathcal{A}' as transformed in step 2 we have **(Subset)**, **(c)**, **(cons)**, **(n)** and **(HNF)**. **(Subset)** holds since all equalities in \mathcal{A} are provable in \mathcal{A}' as well. **Proof for (c)** f is obviously coverage complete in \mathcal{A}' . f was coverage completed for \mathcal{A} . We show $g_{i'}$ is coverage complete. Let T be a matching tree for

f . A matching tree for $g_{i'}$ is obtained as follows : replace labels

$$f \ x_1 \cdots x_{i-1} \ x_i \ t_{i+1} \cdots t_m$$

by

$$f \ x_1 \cdots x_{i-1} \ (C_{i'} \ y_1 \cdots y_r) \ t_{i+1} \cdots t_m$$

Then delete occurrences of nodes with label

$$f \ x_1 \cdots x_{i-1} \ (C_j \ y'_1 \cdots y'_r) \ t_{i+1} \cdots t_m$$

where $i' \neq j$. If $f \ x_1 \cdots x_{i-1} \ (C_{i'} \ y'_1 \cdots y'_r) \ t_{i+1} \cdots t_m$ is now a subnode of $f \ x_1 \cdots x_{i-1} \ (C_{i'} \ y_1 \cdots y_r) \ t_{i+1} \cdots t_m$ then we contract both nodes into one and rename the variables y'_i in all subnodes to y_i .

Finally we replace everywhere $f \ x_1 \cdots x_{i-1} \ (C_{i'} \ s_1 \cdots s_r) \ t_{i+1} \cdots t_m$ by

$$g_{i'} \ x_1 \cdots x_{i-1} \ s_1 \cdots s_r \ t_{i+1} \cdots t_m$$

Then we obtain a matching tree for $g_{i'}$.

Proof for (cons) We check only one case, namely (c), \mathcal{A} is consistent. Assume \mathcal{A}' proves $C \ t_1 \cdots t_n = C' \ t'_1 \cdots t'_k : A$ for constructors C, C' and algebraic data type A . By replacing in the derivation of this equation all terms

$$g_i \ t_1 \cdots t_{i-1} \ s_1 \cdots s_r \ t_{i+1} \cdots t_m$$

by

$$f \ t_1 \cdots t_{i-1} \ (C_i \ s_1 \cdots s_r) \ t_{i+1} \cdots t_m$$

we get a derivation of

$$C \ \hat{t}_1 \cdots \hat{t}_n = C' \ \hat{t}'_1 \cdots \hat{t}'_k : \hat{A}$$

in \mathcal{A} . Therefore by \mathcal{A} being consistent we get $C = C'$ and $n = k$, $\hat{t}_1 = \hat{t}'_1, \dots, \hat{t}_n = \hat{t}'_n$. However, in \mathcal{A} we have $t_i = \hat{t}_i, t'_i = \hat{t}'_i$ since t_i, \hat{t}_i differ only in the contraction of

$$g_i \ t_1 \cdots t_{i-1} \ s_1 \cdots s_r \ t_{i+1} \cdots t_m$$

to

$$f t_1 \cdots t_{i-1} (C_i s_1 \cdots s_r) t_{i+1} \cdots t_m$$

So $t_1 = t'_1, \dots, t_n = t'_n$.

Proof for (n) Assume there exists an infinite reduction sequence in \mathcal{A}' , $r_0 \longrightarrow' r_1 \longrightarrow' r_2 \longrightarrow' \dots$. By replacing

$$g_j t_1 \cdots t_{i-1} t'_1 \cdots t'_{l_j} t_{i+1} \cdots t_m$$

back to

$$f t_1 \cdots t_{i-1} (C_j t'_1 \cdots t'_{l_j}) t_{i+1} \cdots t_m$$

We get $r'_0 \cong r'_1 \cong r'_2 \cong \dots$ obtain terms r'_i in \mathcal{A} and obtain that all terms are terms in \mathcal{A} of the same type. If $r_{i'} \longrightarrow' r_{i'+1}$ in \mathcal{A}' because of the reduction s.t.

$$f t_1 \cdots t_{i-1} (C_j t'_1 \cdots t'_{l_j}) t_{i+1} \cdots t_m \longrightarrow' g_j t_1 \cdots t_{i-1} t'_1 \cdots t'_{l_j} t_{i+1} \cdots t_m$$

then we have $r'_{i'+1} = r'_{i'}$. If $r_{i'} \longrightarrow' r_{i'+1}$ in \mathcal{A}' because of the reduction s.t.

$$g_j t_1 \cdots t_{i-1} t'_1 \cdots t'_{l_j} t_{i+1} \cdots t_m \longrightarrow' s_{j,j'}$$

then we have $r'_{i'} \longrightarrow r'_{i'+1}$ because of the reduction s.t.

$$f t_1 \cdots t_{i-1} (C_j t'_1 \cdots t'_{l_j}) t_{i+1} \cdots t_m \longrightarrow s_{j,j'}$$

If $r_{i'} \longrightarrow' r_{i'+1}$ because of some other reduction then $r'_{i'} \longrightarrow r'_{i'+1}$ because of the same reduction.

If $r'_{i'} = r'_{i'+1}$ in this reduction, one occurrence of f vanishes and no new f is generated (on the right hand side of the reductions for f in \mathcal{A}' , no f occurs) but there are only finite f 's. Hence, there is no infinite sequence of uninterrupted equal(=) chain in \mathcal{A} s.t. $r_j = r_{j+1} = r_{j+2} = \dots$. Therefore, the reduction sequence is $r'_1 = r'_2 = \dots = r'_{i'_1} \longrightarrow r'_{i'_1+1} = \dots = r'_{i'_2} \longrightarrow r'_{i'_2+1} = \dots = r'_{i'_3} \longrightarrow r'_{i'_3+1} = \dots$. Then we obtain an infinite reduction sequence in \mathcal{A} s.t. $r'_{i'_1} \longrightarrow r'_{i'_2} \longrightarrow r'_{i'_3} \longrightarrow r'_{i'_4} \longrightarrow \dots$. However, \mathcal{A} was assumed to be normalising, we get a contradiction.

Proof for (HNF) We know the initial \mathcal{A} was normalising. Step 1 preserved normalisation, so the Agda code in the beginning of step 2 is normalising. \mathcal{A}' induces the head normal form property on the code \mathcal{A} before carrying out step 2: Assume \mathcal{A}' has this property. Let r be a closed term in \mathcal{A} in normal form which is an element of an algebraic data type. However, r might not be in normal form in \mathcal{A}' . Since \mathcal{A}' is normalising, r has a normal form r' w.r.t. \mathcal{A}' . We write \longrightarrow for reductions in \mathcal{A} and \longrightarrow' for reductions in \mathcal{A}' . Let the reduction chain from r to r' in \mathcal{A}' be $r = r_0 \longrightarrow' r_1 \longrightarrow' \dots \longrightarrow' r_n = r'$. Replace in r_i each occurrence of $(g_j t_1 \dots t_{i-1} t'_1 \dots t'_{l_j} t_{i+1} \dots t_m)$ by $(f t_1 \dots t_{i-1} (C_j t'_1 \dots t'_{l_j}) t_{i+1} \dots t_m)$ and obtain terms r'_i . Now if a reduction $r_k \longrightarrow' r_{k+1}$ was a reduction using f , then we have $r'_k = r'_{k+1}$. If it was a reduction using g_j , then we have $r'_k \longrightarrow r'_{k+1}$. If it was a reduction using some other symbols or rules, then we have $r'_k \longrightarrow r'_{k+1}$. So we obtain $r'_0 \longrightarrow^* r'$. But $r = r_0 = r'_0$ and $r'_n = r_n = r'$, so $r \longrightarrow^* r'_n$ in \mathcal{A} . Since r was in normal form in \mathcal{A} we have $r = r'_n$. Then r'_n is obtained from r_n by replacing $(g_j t_1 \dots t_{i-1} t'_1 \dots t'_{l_j} t_{i+1} \dots t_m)$ by $(f t_1 \dots t_{i-1} (C_j t'_1 \dots t'_{l_j}) t_{i+1} \dots t_m)$. Since r_n is in normal form in \mathcal{A}' , it must start with constructor, therefore $r = r'_n$ as well.

Proof that resulting code has smaller measure. Let \mathcal{A}_0 be the code before steps 1 were carried out. We show that $m(\mathcal{A}_n) < m(\mathcal{A}_0)$: We have

- $m^{\mathcal{A}_n}(f) = 0 < m^{\mathcal{A}_0}(f)$ since f has simple pattern.
- We show $m^{\mathcal{A}_n}(g_k) < m^{\mathcal{A}_0}(f)$: Consider a line of g_k .

$$g_k \hat{x}_1^{k,1} \dots \hat{x}_{i-1}^{k,1} s_1^{k,1} \dots s_{l_k}^{k,1} t_{i+1}^{k,1} \dots t_m^{k,1} = s_{k,1}$$

This line originates from a line for f before step 2 which was

$$f \hat{x}_1^{k,1} \dots \hat{x}_{i-1}^{k,1} (C_k s_1^{k,1} \dots s_{l_k}^{k,1}) t_{i+1}^{k,1} \dots t_m^{k,1} = s_{k,1}$$

Before step 1 in \mathcal{A}_0 this was either a line of the same form or we had $s_i^{k,1} = y_i$ and the line in \mathcal{A}_0 was

$$f \hat{x}_1^{k,1} \dots \hat{x}_{i-1}^{k,1} y t_{i+1}^{k,1} \dots t_m^{k,1} = s'_{k,1}$$

Furthermore, no two lines in \mathcal{A}_n originate from the same line in \mathcal{A}_0 . We have two cases:

- The line originates from

$$f \hat{x}_1^l \cdots \hat{x}_{i-1}^l (C_k s_1 \cdots s_{l_k}) t_{i+1}^l \cdots t_m^l = s_l$$

Then we have that the sum of the length of the patterns of this line is one bigger than the sum of the lengths of the patterns in the corresponding line in \mathcal{A}_n (because of the occurrence of C_k).

- The line originates from

$$f \hat{x}_1^l \cdots \hat{x}_{i-1}^l y t_{i+1}^l \cdots t_m^l = s_l$$

Then the new line is

$$g_k \hat{x}_1^{k,1} \cdots \hat{x}_{i-1}^{k,1} z_1 \cdots z_l t_{i+1}^{k,1} \cdots t_m^{k,1} = s'_{k,1}$$

Since variables have length 0, the length of the line in \mathcal{A}_n is the same as the length of the line in \mathcal{A}_0 .

If there is at least one line for the first case, then the sum of the lengths of the lines in \mathcal{A}_n is less than the sum of lengths of lines in \mathcal{A}_0 . If this is not the case, then all lines originate from lines in \mathcal{A}_0 for which the i 's column had a variable. However, there was at least one line in \mathcal{A}_0 which had a proper pattern in column i (starting with a constructor $C_{k'}$ for $k' \neq k$). This line has length > 0 , so the sum of the length of the lines in \mathcal{A}_0 is at least the sum of the lengths of the lines in \mathcal{A}_n plus the length of this line. So again it is bigger. Hence, we obtain in all cases $m^{\mathcal{A}_n}(g_k) < m^{\mathcal{A}_0}(f)$.

- Therefore, $m(\mathcal{A}_n)$ is obtained from $m(\mathcal{A}_0)$ by replacing $m^{\mathcal{A}_0}(f)$ by 0 and several values $m^{\mathcal{A}_n}(g_k)$ which are smaller. So $m(\mathcal{A}_n) < m(\mathcal{A})$.

8.2.2.3 Proof That Agda Normalises Elements of Algebraic Data Types to Head Normal Form

Lemma 8.2.34. *If a is a term, then a can be written as $a = s_0 \cdots s_n$, where s_0 is not an application.*

Proof. Induction on the length of terms in Agda. If a is not an application, $a = s_0$, $n = 0$. Otherwise, $a = s t$. By IH, $s = s_0 \cdots s_n$ where s_0 is not an application then $a = s t = s_0 \cdots s_n t$. \square

Theorem 8.2.35. *Let \mathcal{A} be checked Agda code. Assume \mathcal{A} proves $a : A$, where a is a closed term in normal form and A is an algebraic data type, then a must start with a constructor i.e. $a = C a_1 \dots a_n$, where C is a constructor and $a_1 \dots a_n$ are terms.*

Proof. Replacing \mathcal{A} by checked Agda code we can by theorem 8.2.28 w.l.o.g. assume that \mathcal{A} has only simple pattern matching. Lemma 8.2.34 has shown us $a = s_0 \cdots s_n$, where s_0 is not an application by induction on the length of a . We know that s_0 cannot be a variable, since a is a closed term. If $s_0 = \lambda x.s$ and $n = 1$, then $a = \lambda x.t$ which is not an element of an algebraic data type. If $s_0 = \lambda x.s$ and $n \geq 2$, we would have a β -reduction. Therefore, $s_0 = f$ where f is a defined function or $s_0 = C$, $a = C s_1 \cdots s_n$ where C is a constructor.

Case 1: $s_0 = f$, $a = f s_1 \cdots s_n$

Let the type of f be

$$f : (a_1 : A_1) \rightarrow \dots \rightarrow (a_m : A_m) \rightarrow B$$

where B is not a function type. Note that by Assumption 8.2.24 m is unique. If $n < m$ then $a = f s_1 \cdots s_n : (a_{n+1} : B_n[a_1 := s_1, \dots, a_n := s_n]) \rightarrow \dots \rightarrow (a_m : B_n[a_1 := s_1, \dots, a_n := s_n]) \rightarrow B[a_1 := s_1, \dots, a_n := s_n]$ which is not an algebraic data type. If $n > m$ then $a = f s_1 \cdots s_m : B_n[a_1 := s_1, \dots, a_n := s_n]$ which cannot be applied to s_{m+1} . So $n = m$, $f : (a_1 : A_1) \rightarrow \dots \rightarrow (a_n : A_n) \rightarrow B$.

Subcase 1.1: f is a directly defined function. f is defined directly as

$$\begin{aligned} f & : (a_1 : A_1) \rightarrow \dots \rightarrow (a_n : A_n) \rightarrow B \\ f \ x_1 \cdots x_m & = t \end{aligned}$$

where B is an algebraic data type and t is a term. f depends on variables $x_1 \cdots x_m$.

It is impossible to have $m > n$, because in this case $f \ x_1 \cdots x_n : B[a_1 := x_1, \dots, a_n := x_n]$ which is an algebraic data type and not a function type, so $f \ x_1 \cdots x_n$ cannot to be applied to x_{n+1} . Therefore, $m \leq n$ and $a = f \ s_1 \cdots s_n = (f \ s_1 \cdots s_m) \ s_{m+1} \ \dots \ s_n \rightarrow t[a_1 := x_1, \dots, a_m := x_m] \ s_{m+1} \ \dots \ s_n$. So a has one reduction, a is not in normal form, a contradiction.

Subcase 1.2: f is a postulated function. So we postulated f by having

$$\text{postulate } f : (a_1 : A_1) \rightarrow \dots \rightarrow (a_n : A_n) \rightarrow B$$

where B is a postulated type or an equality.

Then $a = f \ s_1 \cdots s_n$ has a type $B[a_1 := s_1, \dots, a_n := s_n]$ which is a postulated type or an equality and therefore by Assumption 8.2.24 not equal to an algebraic data type.

Subcase 1.3: f is a function defined by pattern matching as follows

$$\begin{aligned} f & : (a_1 : A_1) \rightarrow \dots \rightarrow (a_n : A_n) \rightarrow B \\ f \ \hat{x}_1 \cdots \hat{x}_{k-1} \ (C_i \ \hat{y}_1 \cdots \hat{y}_l) \ \hat{x}_{k+1} \cdots \hat{x}_m & = \hat{s} \\ & \vdots \end{aligned}$$

where A_i and B are algebraic data types. W.l.o.g. $m = n$ (otherwise, n -expand). So by IH and Assumption 8.2.24 s_i must be of the form $(C_i \ t_1 \cdots t_l)$. Assume one of \hat{x}_i or \hat{y}_i is the absurd pattern $()$ then s_j or t_i is an element of a directly empty algebraic data type. It must start with a constructor of this type (by Assumption 8.2.24) but the empty algebraic data type has no constructor, so this case does not occur. By Theorem 8.2.28, we can assume f is defined by simple pattern matching. So f matches a pattern, therefore $a = f \ s_1 \cdots s_{k-1} \ (C_i \ t_1 \cdots t_l) \ s_{k+1} \cdots s_n \longrightarrow s[x_1 := s_1, \dots, x_{k-1} := s_{k-1}, y_1 :=$

$t_1, \dots, y_l := t_l, x_{k+1} := s_{k+1}, \dots, x_n := s_n]$ (note that the pattern is determined uniquely by Assumption 8.2.24). So a has one reduction, a is not in normal form. We get a contradiction.

Case 2: $s_0 = C$, $a = C s_1 \dots s_n$ Then a starts with a constructor C . □

Corollary 8.2.36. *If checked Agda code \mathcal{A} proves that $t : \mathbb{N}$ then t reduces to a natural number (zero or $\text{suc}(\text{suc}(\text{suc}(\dots(\text{suc zero})\dots)))$). If t is an element of a finite term then t reduces to a constructor.*

Corollary 8.2.37. *If checked Agda code \mathcal{A} proves that $t : \text{List Digit}$ then t reduces to $d_0 :: d_1 :: d_2 :: d_3 :: \dots :: d_n :: []$ where $d_i = 0$ or -1 or 1 . If \mathcal{A} proves that $t : \text{String}$ then t reduces to " $a_0 \dots a_n$ " where a_i are characters.*

Note that in the theorem above, the correctness of program extraction, doesn't cover indexed (co)inductive definitions. However, $\sim\mathbb{R}$ is a restricted indexed coinductive definition. One way of avoiding this problem is to extend our theorem to restricted indexed (co)inductive definition data type (see next section). An alternative would be to replace the data type $\sim\mathbb{R}$ by this new parametrised inductive definition

```

data  $\sim\mathbb{R} (r : \mathbb{R}) : \text{Set}$  where
  C : ( $\sim r : \mathbb{N} \rightarrow \mathbb{R}$ )
       $\rightarrow (\sim d : \mathbb{N} \rightarrow \text{Digit})$ 
       $\rightarrow (p : (n : \mathbb{N}) \rightarrow \sim r n \in [-1, 1])$ 
       $\rightarrow (q : (n : \mathbb{N}) \rightarrow \sim r (\text{suc } n) == \text{!}2 * \sim r n - \text{embedD } (\sim d n))$ 
       $\rightarrow \sim r 0 == r$ 
       $\rightarrow \sim\mathbb{R} r$ 

```

So $\sim\mathbb{R} r$ holds if there exists a sequence of digits $\sim d n$ for $(n : \mathbb{N})$, and of $\sim r n \in [-1, 1]$ for $(n : \mathbb{N})$ such that $\sim r 0 = r$ and $\sim r (n + 1) = 2 * \sim r n - \sim d n$. So $\sim d n$ are the digits and $\sim r n$ are the real numbers occur in an proof of $\sim\mathbb{R} r$ using

the original definition. This is just a way of writing the dependent product type

$$\begin{aligned}
 \sim\mathbb{R} r = & (\sim r : \mathbb{N} \rightarrow \mathbb{R}) \\
 & \times (\sim d : \mathbb{N} \rightarrow \text{Digit}) \\
 & \times (p : (n : \mathbb{N}) \rightarrow \sim r n \in [-1, 1]) \\
 & \times (q : (n : \mathbb{N}) \rightarrow \sim r (\text{suc } n) == \text{r2} * \sim r n - \text{embedD } (\sim d n)) \\
 & \times \sim r 0 == r
 \end{aligned}$$

So if $C \sim r \sim d p q l : \sim\mathbb{R} r$ then $r \sim 0.(\sim d 0)(\sim d 1)(\sim d 2)(\sim d 3) \dots$. We can now replace proofs using the old definition of $\sim\mathbb{R} r$ by proofs using the new version. As an example we prove $\sim\mathbb{R} r$ for all rational numbers $r \in [-1, 1]$ by the function $\sim\mathbb{R}\text{embedQ}$ as follows

$$\begin{aligned}
 \sim\mathbb{R}\text{embedQ} : (q : \mathbb{Q}) \rightarrow (\text{embed } q \in [-1, 1]) \rightarrow \sim\mathbb{R} (\text{embed } q) \\
 \sim\mathbb{R}\text{embedQ } q p = C \sim r \sim d p' q' (\text{refl} == (\sim r 0))
 \end{aligned}$$

where

mutual

$$\begin{aligned}
 \sim d : \mathbb{N} \rightarrow \text{Digit} \\
 \sim d n = \pi^l (2q - d \in [-1, 1] - \text{Prod } (\sim q n) (p' n))
 \end{aligned}$$

$$\begin{aligned}
 \sim q : \mathbb{N} \rightarrow \mathbb{Q} \\
 \sim q 0 = q \\
 \sim q (\text{suc } n) = \text{r2} * \sim q n - \text{embed}_{(d)} \rightarrow \mathbb{Q} (\sim d n)
 \end{aligned}$$

$$\begin{aligned}
 p' : (n : \mathbb{N}) \rightarrow (\sim r n) \in [-1, 1] \\
 p' n = \dots
 \end{aligned}$$

$$\begin{aligned}
 \sim r : \mathbb{N} \rightarrow \mathbb{R} \\
 \sim r n = \text{embed } (\sim q n)
 \end{aligned}$$

$$\begin{aligned}
 q' : (n : \mathbb{N}) \rightarrow (\sim r' (\text{suc } n)) == (\text{r2} * (\sim r n) - \text{embedD } (\sim d n)) \\
 q' n = \text{refl} == (\sim r (\text{suc } n))
 \end{aligned}$$

where the definition of $2q-d \in [-1, 1]$ -Prod can be found in Section 7.2.3.

8.2.2.4 Extensions of this Theorem

It seems our method works as well for restricted indexed inductive definitions (data and codata types). In order to introduce this we introduce first generalised indexed inductive definition:

A generalised indexed inductive definition allows us to define

$$\begin{aligned} \text{data } A : B \rightarrow \text{Set} \quad \text{where} \\ C_1 : \dots \rightarrow (x : A \ b_2) \rightarrow \dots \rightarrow A \ b_1 \\ C_2 : \dots \rightarrow (x : A \ b_4) \rightarrow \dots \rightarrow A \ b_3 \\ \vdots \end{aligned}$$

So the constructors have result types A applied to an arbitrary element of B and might refer to other $b : B$. An example would be

$$\begin{aligned} \text{data } A : \mathbb{N} \rightarrow \text{Set} \quad \text{where} \\ C_1 : A \ \underbrace{3}_{\text{index}} \\ C_2 : (n : \mathbb{N}) \rightarrow A \ (n * 2) \rightarrow A \ (n + 17) \end{aligned}$$

So the result type can have an arbitrary index and the arguments can have arbitrary different arguments. Case distinction doesn't work on generalised indexed inductive definition. If we have

$$f : A \ x \rightarrow B$$

we don't know whether C_1 is an element of $A \ x$ or not, so we don't know the constructors on which we make case distinction. On the other hand a restricted indexed inductive definition is as follows

$$\begin{aligned} \text{data } A : \mathbb{N} \rightarrow \text{Set} \quad \text{where} \\ C_1 : (n : \mathbb{N}) \rightarrow A \ n \\ C_2 : (n : \mathbb{N}) \rightarrow A \ (n + 17) \rightarrow A \ n \end{aligned}$$

where the first argument in both constructors is an index and the result type is A applied to this index. The second argument $A (n + 17)$ in C_2 refers to different $A n$. So the first argument of the constructor is a variable of the index type, the result type is the type applied to this variable but we can refer to the type applied to different indexes. If we have $A t$ for some t we know which constructor occurs and we can make a case distinction on it which is in Agda written as follows

$$\begin{aligned} f &: A t \rightarrow B \\ f (C_1 .t) &= \dots \\ f (C_2 .t t') &= \dots \end{aligned}$$

In order to reduce deep patterns by simple patterns the above could be replaced by:

$$\begin{aligned} f &: A t \rightarrow B \\ f (C_1 .t) &= \dots \\ f (C_2 .t y) &= g_2 y \end{aligned}$$

$$\begin{aligned} g_2 &: A (t + 17) \rightarrow B \\ g_2 t' &= \dots \end{aligned}$$

So the first argument of C_1 stays in f and the other arguments are passed on to the function g_2 . It seems our proof works as well for restricted indexed inductive definition. Working out the details is left for future work.

A further extension would be to extend our theorem to codata types. Since we need normalisation, we cannot use codata type verbally, instead we use the coalgebraic definition, i.e. by defining

$$\begin{aligned} \text{coalg } A &: \text{Set} \quad \text{where} \\ \text{elim} &: A \rightarrow B \end{aligned}$$

where B refers to A strictly positively. It seems that Theorem 8.2.35 can be extended, however we need to prove simultaneously:

- if $a : A$, A is an algebraic data type, a closed and in normal form, then a starts with a constructor.

- if $a : A$, A is a coalgebraic type, a closed and in normal form, then $\text{elim } a$ has a reduction.

Finally we need to combine the two extensions in order to deal with $\sim\mathbb{R}$, which is in fact a restricted indexed coalgebraic data type, namely

$$\begin{aligned} \text{codata } \sim\mathbb{R} & : \mathbb{R} \rightarrow \text{Set} \quad \text{where} \\ \text{cons} & : (d : \text{Digit})(r : \mathbb{R}) \\ & \rightarrow \sim\mathbb{R} (\text{r2} * r - \text{embedD } d) \\ & \rightarrow r \in [-1, 1] \\ & \rightarrow \sim\mathbb{R} r \end{aligned}$$

More precisely we need to exchange the first two arguments in the constructors of $\sim\mathbb{R}$ in order to obtain a restricted indexed coinductive data type. This is no problem since these arguments are independent of each other. We don't adopt this since we discovered this problem close to the submission date of this thesis. Neither do we adopt our proof to incorporate restricted indexed coinductive definitions because of time constraints.

Chapter 9

Conclusion

9.1 Achievements

In this thesis we have extracted programs from proofs about real number computations in Agda. The main achievement of the thesis is to determine conditions which guarantee normalisation of the given extracted functions and that we have shown that under these conditions program extraction works. This included a reduction of pattern matching to simple pattern matching which can be used for other purposes as well and uses a sophisticated proof.

We began by formalising

- \mathbb{N} (the natural numbers),
- \mathbb{N}^+ (natural numbers plus one),
- \mathbb{Z} (the integers),
- \mathbb{Q} (the rational numbers)

with operations. We axiomatized the real numbers \mathbb{R} by using postulated data types and functions. Then we have investigated some properties on real numbers constructed by Cauchy sequences: we have introduced

- \mathbb{Q} (the set rational numbers in \mathbb{R} which are rational numbers).

- the Cauchy reals \mathbb{Q}' (so $\{r : \mathbb{R} \mid \mathbb{Q}' r\}$ which are the real numbers which are limits of Cauchy sequences of rational number).
- \mathbb{Q}'' (which are the real numbers which are limits of elements in \mathbb{Q}').

We gave proofs that

- \mathbb{Q}' is closed under addition and multiplication;
- \mathbb{Q}' is Cauchy complete, i.e. $\mathbb{Q}'' \subseteq \mathbb{Q}'$. Therefore, every Cauchy sequence in \mathbb{Q}' has a limit in \mathbb{Q}' .

We introduced the real numbers which have binary signed digit stream representation. Here, the signed digit stream $d_0 :: d_1 :: d_2 :: \dots$ (commonly written as $r \sim 0.d_0d_1d_2\dots$) represents the number

$$\sum_{i=0}^{\infty} d_i * 2^{-i+1}$$

We showed that signed digit representable real numbers are Cauchy reals. In order to work on computation of binary signed digit stream we introduced

- $\sim\mathbb{R}$ (which are the real numbers in the interval $[-1,1]$ which have a signed digit representation (SDR));
- transfer $\sim\mathbb{R}$ function s.t transfer $\sim\mathbb{R} : (r : \mathbb{R}) \rightarrow \sim\mathbb{R} r \rightarrow (s : \mathbb{R}) \rightarrow s == r \rightarrow \sim\mathbb{R} s$ (this overcomes our restriction that case distinctions on equalities need to have result types equalities or postulated types);
- $\sim\mathbb{R}$ embed \mathbb{Q} function (embedding of \mathbb{Q} into SDR);

and we have given

- proofs that $\sim\mathbb{R}$ is closed under average and multiplication.

We also defined a function

$$\text{fd} : \mathbb{N} \rightarrow (r : \mathbb{R}) \rightarrow \sim\mathbb{R} r \rightarrow \text{String}$$

such that $(\text{fd } n \ r \ p)$ returns the first n digits of p . Then we compiled those proofs into Haskell (we used the fact that Agda can be compiled into Haskell which can be executed effectively due to lazy evaluation). Furthermore we gave a new proof that forming the multiset ordering in finite bags preserves well-foundedness.

9.1.1 Our Program Extraction Method

We adopted the feature of proofs as programs inside Agda and provided a correctness of our method which is an innovative approach to program extraction.

Our method of program extraction in the thesis is as follows: the type of the real numbers \mathbb{R} which can be postulated without giving any computation rules was introduced. The coinductive data type of the real numbers in $[-1,1]$ which can be approximated arbitrarily close by signed digit binary floating point numbers $0.a_0a_1 \cdots a_N$, where $a_i \in \{-1, 0, 1\}$ was introduced. The resulting data type is $\sim\mathbb{R} : \mathbb{R} \rightarrow \text{Set}$. Then we proved theorems of the form

$$\forall r_1, \dots, r_n : \mathbb{R}. \sim\mathbb{R} \ r_1 \rightarrow \cdots \rightarrow \sim\mathbb{R} \ r_n \rightarrow \exists r. \varphi(r_1, \dots, r_n, r) \wedge \sim\mathbb{R} \ r$$

From an $r : \mathbb{R}$ s.t. $(\sim\mathbb{R} \ r)$ holds we extracted a stream of signed digits representing it. Furthermore, for every stream s is computed a real number r s.t. $\sim\mathbb{R} \ r$ holds. Therefore we obtained in Agda from the theorem a function

$$f : \text{Stream Digit} \rightarrow \cdots \rightarrow \text{Stream Digit} \rightarrow \text{Stream Digit}$$

s.t. if s_1, \dots, s_n are the streams extracted from signed digits r_1, \dots, r_n , then the $\sim\mathbb{R} \ r$ given by the theorem extracts to stream s .

Using these functions we computed in Agda a stream $s : \text{Stream Digit}$ representing a real number which has a certain property. We defined a function $\text{toList } f : \text{Stream Digit} \rightarrow \mathbb{N}^+ \rightarrow \text{List Digit}$ s.t. $(\text{toList } s \ n)$ returns the first n digits of s . If our theorem $(\text{toList } s \ n)$ normalises, we have computed its first n digits. Note that s was referring to postulated data types and theorems used.

9.1.2 Correctness Theorem

We proved a theorem showing that under certain conditions $(\text{toList } s \ n)$ always normalises to a list of signed digits and therefore doesn't make use of the axioms. The conditions mainly guarantee that a postulated function or theorem has as result type only a postulated type, so the computation of elements of algebraic data types to head normal form will not refer to these postulates. Therefore, $(\text{toList } s \ n)$ returns a list of n digits.

The method has been used for showing that the signed digit approximable real numbers are closed under average, multiplication, and contain the rational numbers. Therefore, we obtain in Agda a provably correct program which executes the corresponding operations on signed digit streams.

9.2 Future Work

The main tasks that were left for future work are:

- 1 Extension to irrational numbers such as π , e , or functions like \sin , exponential function as operations on SDR. Note that the operations discussed in this thesis only generate rational numbers from rational numbers and therefore don't allow to generate the SDR of any irrational number. In a paper Berger and Hou [BH08] have shown the real numbers (in the interval $[-1,1]$) with SDR are closed under division and shown that in general the real numbers with SDR are closed under all continuous functions provided they stay in the interval $[-1,1]$. One next step would be to carry this out in Agda.
- 2 Berger has considered other representations of real numbers. In the approach used in this thesis $r \sim 0.d_0d_1d_2\dots$ means

$$r = \text{av}_{d_0} \circ \text{av}_{d_1} \circ \text{av}_{d_2} \circ \dots$$

where

$$\text{av}_{d_i} = \frac{d_i + \text{av}_{d_{i+1}} \circ \text{av}_{d_{i+2}} \circ \dots}{2}$$

So signed digits correspond to representing real numbers as an infinite composition of functions av_{-1} , av_0 and av_1 . These functions av_{-1} , av_0 , av_1 can be replaced by other functions, and the next step would be to explore such representations in Agda.

- 3 Extend our method in Chapter 8 and work out the correctness in the presence of codata types and restricted indexed inductive and coinductive definitions. This is necessary since our method uses codata types and restricted indexed inductive definitions.

Beside if one would like to work with program extraction on the proofs in Chapter 5, one would redefine the data type \mathbb{Q} as follows:

$$\mathbb{Q} r := (q : \mathbb{Q}) \times (\text{embed}\mathbb{Q} \rightarrow \mathbb{R} q == r)$$

and work with this data type. Then the embedding \mathbb{Q} to \mathbb{Q} is $\pi^l(p)$ for $p : \mathbb{Q} r$. This would overcome the problem (as we mentioned in Chapter 5.1) that the axiom $\neg (r0 \# r0)$ has computational content and is therefore not allowed. This axiom is necessary for the embedding of the data type \mathbb{Q} into \mathbb{Q} used in the thesis. Then we could extract programs from the proofs in Chapter 5 on Cauchy sequences for addition and multiplication of Cauchy sequences. We could even translate signed digit reals into elements of \mathbb{Q}' and vice versa (restricted to the interval $[-1,1]$) and define other operations for defining and transforming Cauchy sequences and therefore obtain from signed digits via their Cauchy sequences signed digit representations of the result of these operations.

An alternative solution would be to introduce a second version of \perp

$$\text{postulate } \perp' : \text{Set}$$

together with

$$\text{postulate } \text{efq}' : (A : \text{Set}) \rightarrow \perp' \rightarrow A$$

which can be used only for A begin a postulated type or an equality. The above would fulfil our conditions, provided efq' is used only for postulated types and

equalities. Now we introduce

$$\text{postulate } \text{-}0\#0 : \text{r}0 \# \text{r}0 \rightarrow \perp'$$

which is an axiom. In order to define

$$\begin{aligned} f &: (r : \mathbb{R}) \rightarrow (p : \mathbb{Q} \ r) \rightarrow \mathbb{Q} \\ f \ .(\text{recip } r \ p) \ (\text{closerecip } r \ n \ p) &= \{! \ !\} \end{aligned}$$

where $p : r \# \text{r}0$, we can define it as $1/(f \ r \ p)$ if $f \ r \ p \neq 0$ and as $1/1$ otherwise.

When showing that the embedding \mathbb{Q} into \mathbb{R} is the inverse of f , i.e.

$$\text{lemmainverse} : (r : \mathbb{R}) \rightarrow (p' : \mathbb{Q} \ r) \rightarrow \text{embed}\mathbb{Q} \ (f \ r \ p') == r$$

we could in case $r = (\text{recip } r' \ p)$, $p = \text{closerecip } r \ p'$, $f \ r \ p == \text{r}0$ use the IH that therefore $r == \text{r}0$. Since $p : r \# \text{r}0$ we would get $\text{zero} \# \text{zero}$, by $(\text{r}0 \# \text{r}0) \rightarrow \perp'$ therefore a proof of \perp' from which using an allowed instance of efq' we obtain $\text{embed}\mathbb{Q} \ (f \ r \ p') == r$.

Alternative approaches to type theory: explicit mathematics and Frege structures. It would be interesting to investigate the use of explicit mathematics [Fef75] and Frege structures [Kah99] in this context. One question would be to find out whether the representation of streams in Frege structures could be obtained by program extraction.

9.3 Possible Simplification

In this thesis we have proved most properties in Agda and therefore verified the correctness using an interactive theorem prover. This is not strictly necessary for program extraction, only for guaranteeing correctness. If we did it again, instead we could only prove the theorems with computational content (which have as result type an algebraic data type) and leave the rest as postulated axioms. In

spite of the fact that we loose proofs, we would still know that the resulting programs terminate by Theorem 8.2.34 as long as the postulated axioms are correct. This has been tested in Appendix A.

Appendix A

This is an example of computing the first 1000 digits of $29/37 * -29/3998$. We postulate functions which have as result type a type which has no computational content i.e. $<$, $==$, \leq . Since their result type is not an algebraic data type the code fulfils our conditions. Firstly we define rational numbers $29/37$, $-29/3998$, $\sim\mathbb{R} (29/37)$ and $\sim\mathbb{R} (-29/3998)$ as follow

$q_{29/37} : \mathbb{Q}$

$q_{29/37} = \text{pos}(28 + 1) \%' (36 + 1)$

$\sim\mathbb{R} q_{29/37} : \sim\mathbb{R} (\text{embed} \mathbb{Q} \rightarrow \mathbb{R} q_{29/37})$

$\sim\mathbb{R} q_{29/37} = \sim\mathbb{R} \text{embed} \mathbb{Q} (q_{29/37}) (\text{embed} \mathbb{Q} \rightarrow \mathbb{R} q_{29/37}) w$
 $(\text{refl} == (\text{embed} \mathbb{Q} \rightarrow \mathbb{R} q_{29/37}))$

where

postulate $l : -^r 1 \leq \text{embed} \mathbb{Q} \rightarrow \mathbb{R} q_{29/37}$

postulate $r : \text{embed} \mathbb{Q} \rightarrow \mathbb{R} q_{29/37} \leq^r 1$

$w : \text{embed} \mathbb{Q} \rightarrow \mathbb{R} q_{29/37} \in [-1, 1]$

$w = \text{and } l \ r$

$q_{-29/3998} : \mathbb{Q}$

$q_{-29/3998} = \text{neg} (28 + 1) \%' (3997 + 1)$

$\sim\mathbb{R} q_{-29/3998} : \sim\mathbb{R} (\text{embed} \mathbb{Q} \rightarrow \mathbb{R} q_{-29/3998})$

$\sim\mathbb{R} q_{-29/3998} = \sim\mathbb{R} \text{embed} \mathbb{Q} (q_{-29/3998}) (\text{embed} \mathbb{Q} \rightarrow \mathbb{R} q_{-29/3998}) w$
 $(\text{refl} == (\text{embed} \mathbb{Q} \rightarrow \mathbb{R} q_{-29/3998}))$

where

postulate $l : -1 \leq \text{embed}\mathbb{Q} \rightarrow \mathbb{R} \ q - 29/3998$

postulate $r : \text{embed}\mathbb{Q} \rightarrow \mathbb{R} \ q - 29/3998 \leq 1$

$w : \text{embed}\mathbb{Q} \rightarrow \mathbb{R} \ q - 29/3998 \in [-1, 1]$

$w = \text{and } l \ r$

We have rational numbers $29/37$, $29/3998$ and compute $\sim\mathbb{R}(29/37)$ and $\sim\mathbb{R}(29/3998)$.

We can compute $\sim\mathbb{R}(29/37 * -29/3998)$ by using mp function:

```
mpq29/37*q-29/3998 : ~ℝ (embedℚ → ℝ q29/37 * embedℚ → ℝ q-29/3998)
mpq29/37*q-29/3998 = ~mp (embedℚ → ℝ q29/37)
                        (embedℚ → ℝ q-29/3998)
                        (embedℚ → ℝ q29/37 * embedℚ → ℝ q-29/3998)
                        ~ℝq29/37 ~ℝq-29/3998
                        (refl== (embedℚ → ℝ q29/37 * embedℚ → ℝ q-29/3998))
```

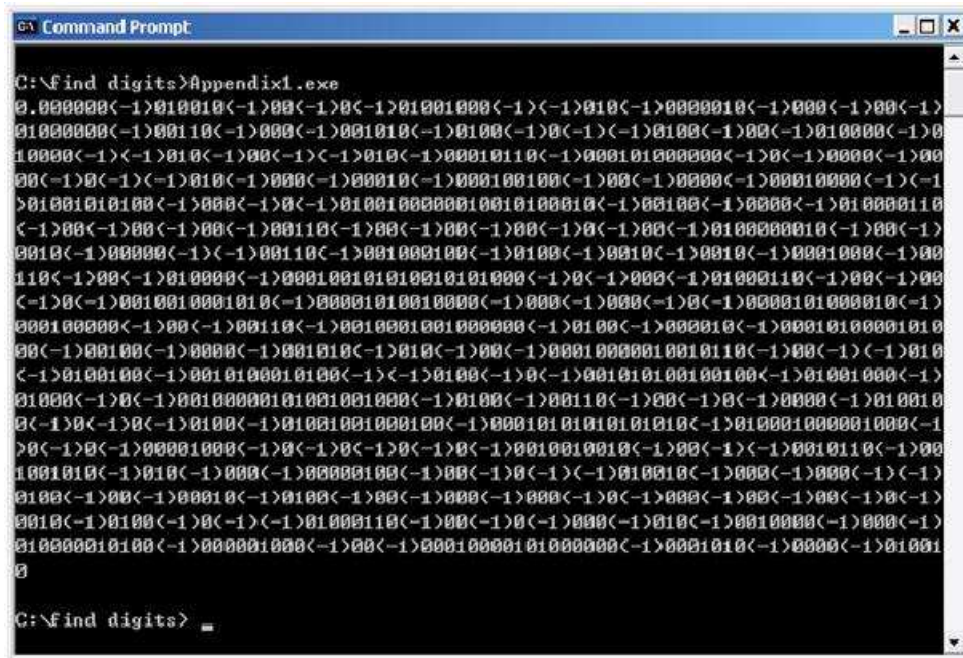
We now can compute 1000 digits $\sim\mathbb{R}(29/37 * -29/3998)$ using fd function:

```
fd_mpq29/37*q-29/3998 : String
fd_mpq29/37*q-29/3998 = fd 1000
                        (embedℚ → ℝ q29/37 * embedℚ → ℝ q-29/3998)
                        (mpq29/37 * q-29/3998)
```

main : IO Unit

main = putStrLn fd_mpq29/37*q-29/3998

Then we compile the Agda file and execute it and obtain



```
C:\find digits>Appendix1.exe
0.000000<-1>010010<-1>00<-1>0<-1>01001000<-1><-1>010<-1>0000010<-1>000<-1>00<-1>
01000000<-1>00110<-1>000<-1>001010<-1>0100<-1>0<-1><-1>0100<-1>00<-1>010000<-1>0
10000<-1><-1>010<-1>00<-1><-1>010<-1>00010110<-1>000101000000<-1>0<-1>0000<-1>00
00<-1>0<-1><-1>010<-1>000<-1>00010<-1>000100100<-1>00<-1>0000<-1>00010000<-1><-1>
>01001010100<-1>000<-1>0<-1>010010000010010100010<-1>00100<-1>0000<-1>010000110
<-1>00<-1>00<-1>00<-1>00110<-1>00<-1>00<-1>00<-1>00<-1>0<-1>00<-1>0100000010<-1>00<-1>
0010<-1>00000<-1><-1>00110<-1>001000100<-1>0100<-1>0010<-1>0010<-1>0001000<-1>00
110<-1>00<-1>010000<-1>000100101010010101000<-1>0<-1>000<-1>01000110<-1>00<-1>00
<-1>0<-1>0010010001010<-1>00001010010000<-1>000<-1>000<-1>0<-1>0000101000010<-1>
000100000<-1>00<-1>00110<-1>0010001001000000<-1>0100<-1>000010<-1>00010100001010
00<-1>00100<-1>0000<-1>001010<-1>010<-1>00<-1>00010000010010110<-1>00<-1><-1>010
<-1>0100100<-1>0010100010100<-1><-1>0100<-1>0<-1>001010100100100<-1>01001000<-1>
01000<-1>0<-1>00100000101001001000<-1>0100<-1>00110<-1>00<-1>0<-1>0000<-1>010010
0<-1>0<-1>0<-1>0100<-1>01001001000100<-1>00010101010101010<-1>010001000001000<-1>
>0<-1>0<-1>00001000<-1>0<-1>0<-1>0<-1>0<-1>0010010010<-1>00<-1><-1>0010110<-1>00
1001010<-1>010<-1>000<-1>00000100<-1>00<-1>0<-1><-1>010010<-1>000<-1>000<-1><-1>
0100<-1>00<-1>00010<-1>0100<-1>00<-1>000<-1>000<-1>00<-1>000<-1>00<-1>00<-1>00<-1>
0010<-1>0100<-1>0<-1><-1>01000110<-1>00<-1>0<-1>000<-1>010<-1>0010000<-1>000<-1>
010000010100<-1>000001000<-1>00<-1>0001000010100000<-1>0001010<-1>0000<-1>01001
```

(it took 15.44 seconds to compute the digits)

Bibliography

- [Aug98] L. Augustsson. Cayenne—a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1998. ACM.
- [Bar93] M. Barr. Terminal coalgebras in well-founded set theory. *Theoretical Computer Science*, 114(2):299–315, 1993.
- [BBS06] U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82:25–49, 2006.
- [BBS+98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II of Applied Logic Series. Kluwer, 1998. Dordrecht (1998) 41–71.
- [BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
- [Ben07] M. Benke. Alonzo – a Compiler for Agda. In *TYPES 2007, Conference of the Types Project*, 2007. Available from <http://www.mimuw.edu.pl/~ben/Papers/TYPES07-alonzo.pdf>.

- [Ber93] U. Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 91–106. Springer Verlag, 1993.
- [Ber05a] U. Berger. Uniform Heyting arithmetic. *Annals Pure Applied Logic*, 133:2005, 2005.
- [Ber05b] Y. Bertot. Coinduction in Coq. In *Lecture Notes of the TYPES Summer School 2005, Sweden, Volume II*, 2005.
- [Ber07] Y. Bertot. Affine functions and series with co-inductive real numbers. *Mathematical. Structures in Comp. Sci.*, 17(1):37–63, 2007.
- [Ber09a] U. Berger. From coinductive proofs to exact real arithmetic. In E. Grädel and R. Kahle, editors, *Computer Science Logic*, volume 5771 of *LNCS*, pages 132–146. Springer Verlag, 2009.
- [Ber09b] U. Berger. Realisability and adequacy for (Co)induction. In Andrej Bauer, Peter Hertling, and Ker-I Ko, editors, *6th Int’l Conf. on Computability and Complexity in Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.
- [BH08] U. Berger and T. Hou. Coinduction for exact real number computation. *Theory of Computing Systems*, 43:394–409, 2008.
- [Bla05] J. Blanck. Efficient exact computation of iterated maps. *Journal of Logic and Algebraic Programming*, 64:41–59, 2005.
- [BN98] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.
- [BS05] U. Berger and M. Seisenberger. Applications of inductive definitions and choice principles to program synthesis. In *From Sets and Types to Topology and Analysis. Towards practicable foundations for constructive mathematics*, volume 48 of *Oxford Logic Guides*, pages 137–148. Oxford University Press, 2005.

- [BS10a] U. Berger and M. Seisenberger. Program Extraction via Typed Realisability for Induction and Coinduction. In Ralf Schindler, editor, *Ways of Proof Theory*, Ontos Series in Mathematical Logic, pages 153 – 178. Ontos Verlag, 2010.
- [BS10b] U. Berger and M. Seisenberger. Proofs, Programs, Processes. CiE 2010, LNCS 6158, 2010.
- [BSB02] U. Berger, H. Schwichtenberg, and W. Buchholz. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114:3–25, 2002.
- [CG06] A. Ciaffaglione and P. Gianantonio. A certified, corecursive implementation of exact real numbers. *Theoretical Computer Science*, 351(1):39–51, 2006.
- [Coq05] C. Coquand. Agda version 1 official web site. <http://unit.aist.go.jp/cvs/Agda/>, 2005.
- [Coq09] Coq Development Team. The Coq proof assistant. <http://coq.inria.fr/>, 2009.
- [Ebb91] H.-D. Ebbinghaus. *Numbers*. Number 123 in Graduate Texts in Mathematics. Springer, 1991.
- [Ebe02] M. Eberl. *Normalization by Evaluation*. PhD thesis, Mathematisches Institut der Universität München, 2002.
- [EH02] A. Edalat and R. Heckmann. Computing with real numbers - I. The LFT Approach to Real Number Computation - II. A Domain Framework for Computational Geometry. In *Proc APPSEM Summer school in Portugal*, pages 193–267. Springer Verlag, 2002.
- [Fef75] S. Feferman. A language and axioms for explicit mathematics. In John Crossley, editor, *Algebra and Logic. Proc. 1974, Monash Univ Australia*, volume 450 of *Springer Lecture Notes in Mathematics*, pages 87 – 139, 1975.

- [Gor94] A. Gordon. A tutorial on co-induction and functional programming. In *Glasgow functional programming workshop*, pages 78–95. Springer, 1994.
- [Has] Haskell wiki. <http://www.haskell.org/haskellwiki/Haskell>.
- [HGW07] B. Spitters H. Geuvers, M. Niqui and F. Wiedijk. Constructive analysis, types and exact real numbers. *Mathematical Structures in Computer Science*, 17:3–36, 2007.
- [HS99] P. Hancock and A. Setzer. The IO monad in dependent type theory. In *Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27-28 March 1999*, 1999. Available via <http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99.html>.
- [HS00a] P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic. 14th international workshop, CSL 2000*, Springer Lecture Notes in Computer Science, Vol. 1862, pages 317 – 331, 2000.
- [HS00b] P. Hancock and A. Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal, 7 July 2000*, 2000. Electronic proceedings, Available via <http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html>.
- [HS04] P. Hancock and A. Setzer. Interactive programs and weakly final coalgebras (extended version). In T. Altenkirch, M. Hofmann, and J. Hughes, editors, *Dependently typed programming*, number 04381 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2004. Available via <http://drops.dagstuhl.de/opus/>.
- [HS05] P. Hancock and A. Setzer. Guarded induction and weakly final coalgebras in dependent type theory. In L. Crosilla and P. Schuster, editors, *From Sets and Types to Topology and Analysis. Towards Practical Foundations for Constructive Mathematics*, pages 115 – 134, Oxford, 2005. Clarendon Press.

- [Isa09] Isabelle community. Isabelle, 2009.
<http://www.cl.cam.ac.uk/research/hvg/Isabelle/>.
- [Kah99] R. Kahle. Frege structures for partial applicative theories. *Journal of Logic and Computation*, 9(5):683–700, 1999.
- [Kea96] R. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2:95–112, 1996.
- [Kon04] M. Konečný. Real functions incrementally computable by finite automata. *Theoretical Computer Science*, 315(1):109–133, 2004.
- [Krä98] W. Krämer. A priori worst case error bounds for floating-point computations. *IEEE Trans. Comput.*, 47(7):750–756, 1998.
- [Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North Holland, Amsterdam, 1959.
- [MKJ08] S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming using dependent types. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction*, volume 5133 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-70594-9_15.
- [MKJ09] S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(05):545–579, 2009.
- [ML190] The standard ML language. <http://www.lfcs.inf.ed.ac.uk/software/ML/>, 1990.
- [ML84] P. Martin-Löf. *Intuitionistic Type Theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980*. Number 1 in *Studies in Proof Theory*. Bibliopolis, Naples, 1984.
- [MM08] C. McBride and J. McKinna. Epigram Homepage. <http://www.epig.org/>, 2008.

- [MN94] L. Magnusson and B. Nordström. The Alf proof editor and its proof engine. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, pages 213–237, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.
- [MRE07] J. Raymundo M. Romero and M. Escardó. Semantics of a sequential language for exact real-number computation. *Theoretical Computer Science*, 379(1-2):120–141, 2007.
- [Niq08] M. Niqui. Coinductive formal reasoning in exact real arithmetic. *Logical Methods in Computer Science*, 4(3:6):1–40, September 2008.
- [Nor07] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
- [Nor08] U. Norell. Dependently typed programming in Agda. In *Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.
- [Nor09] U. Norell. The Agda Wiki.
<http://wiki.portal.chalmers.se/agda/agda.php>, 2009.
- [NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.
- [Nuo10] L. Nuo. Representing numbers in Agda. Third Year Project, Dept. of Computer Science, University of Nottingham. Available from http://www.cs.nott.ac.uk/~nzl/Home_Page/Homepage.html, May 2010.
- [OTK09] H. Ozaki, M. Takeyama, and Y. Kinoshita. Agate-an agda-to-haskell compiler. *Computer Software*, 26(4):107–119, 2009.
- [PEE97] P. Potts, A. Edalat, and M. Escardo. Semantics of exact real arithmetic. In *LICS '97: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, page 248, Washington, DC, USA, 1997. IEEE Computer Society.

- [Plo77] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):225–255, 1977.
- [Plu] D. Plume. A calculator for exact real number computation. <http://www.dcs.ed.ac.uk/home/mhe/plume/report.html>.
- [Sch08] H. Schwichtenberg. Realizability interpretation of proofs in constructive analysis. *Theor. Comp. Sys.*, 43(3):583–602, 2008.
- [Sei01] M. Seisenberger. Kruskal’s tree theorem in a constructive theory of inductive definitions. In P. Schuster, U. Berger, and H. Osswald, editors, *Reuniting the Antipodes - Constructive and Nonstandard Views of the Continuum . Proceedings of a Symposium in San Servolo/Venice Italy 1999 May 17-22*. Synthese Library 306, Kluwer Academic Publishers, Dordrecht, 2001.
- [Sei02] M. Seisenberger. An inductive version of Nash-Williams’ minimal-bad-sequence argument for Higman’s lemma. In *TYPES ’00: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 233–242, London, UK, 2002. Springer-Verlag.
- [Sei03] M. Seisenberger. *On the computational content of proofs*. PhD thesis, Ludwigs-Maximilians-Universität München, 2003.
- [Sei08] M. Seisenberger. Programs from proofs using classical dependent choice. *Annals of Pure and Applied Logic*, 153(1-3):97 – 110, 2008. Special Issue: Classical Logic and Computation (2006).
- [Set09] A. Setzer. Coalgebras and codata in Agda. Presentation at the 3rd Wessex Theory Seminar, 2009.
- [Set10a] A. Setzer. Coalgebras in dependent type theory. Talk given in the workshop on Dependently Type Programming associated with LICS, July 2010. <http://sneezy.cs.nott.ac.uk/darcs/dtp10/>.
- [Set10b] A. Setzer. Coalgebras in dependent type theory – the saga continues. Talk given at Agda Intensive Meeting Xii 2010, Nottingham, 1. - 7. September 2010.

- [Set10c] A. Setzer. Extraction of programs from proofs about real numbers in dependent type theory, joint work with Chi Ming Chuang, invited talk. In *Workshop on Program Extraction and Constructive Proofs, Workshop in honour of Helmut Schwichtenberg, Satellite workshop of CSL 2010 and MFCS 2010*, Brno, August 21 2010. Slides Available at <http://www.cs.swan.ac.uk/~csetzer/slides/index.html>.
- [Tat98] M. Tatsuta. Realizability of monotone coinductive definitions and its application to program synthesis. In *MPC '98: Proceedings of the Mathematics of Program Construction*, pages 338–364, London, UK, 1998. Springer-Verlag.
- [Tro73] A. Troelstra. Metamathematical investigation of intuitionistic arithmetic and analysis. In *Lecture Notes in Mathematics*, volume 344, 1973.
- [YD94] C. Yap and T. Dubé. The exact computation paradigm. *World Scientific Press*, 1994.