Incompleteness & Completeness

Formalizing Logic and Analysis in Type Theory

Russell O'Connor

Incompleteness & Completeness:
Formalizing Logic and Analysis in Type Theory

Een wetenschappelijke proeve op het gebied
van de Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus
prof. mr. S.C.J.J. Kortmann,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
maandag 5 oktober 2009
des namiddags om 13.30 uur precies
door

Russell Steven Shawn O'Connor

geboren op 19 december 1977
te Winnipeg, Canada

Promotor:
Prof. dr. J.H. Geuvers

Copromotor:
Dr. B. Spitters

Manuscriptcommissie:

Prof. dr. B. Jacobs	
Prof. dr. G. Dowek	École Polytechnique, France
Dr. J. Harrison	Intel Corportation, Oregon, VS
Dr. N. Müller	Universität Trier
Prof. dr. H. Schwichtenberg	Ludwig-Maximilians-Universität München

ISBN: 978-90-9024455-6
Printed by Ipskamp Drukkers B.V., Enschede
IPA Dissertation Series 2009-19

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

In memory of my father, Patrick Adrian O'Connor (1948–2008).

Acknowledgements

I would like to first thank my promoter and co-promoter, Herman Geuvers and Bas Spitters. I was able to count on them to listen to my questions and ideas, and they both provided helpful suggestions and ideas. Without their help this thesis would not be possible.

I would like to thank the members of my manuscript committee: Gilles Dowek, John Harrison, Norbert Müller, and Helmut Schwichtenberg for taking the time to review and comment on my draft thesis. In particular, I would like to thank Bart Jacobs for heading my manuscript committee and providing expert comments on my use of category theory.

I would like to thank Henk Barendregt, who holds the chair of the *Foundations of Mathematics and Computer Science* group at Radboud University Nijmegen. His vision of computer formalized mathematics inspires all members of the group. I would also like to thank all the members of the group that I had the privilege of working with: Dick van Leijenhorst, James McKinna, Freek Wiedijk, Venanzio Capretta, Pierre Corbineau, Olga Tveretina, Tonny Hurkens, Milad Niqui, Dan Synek, Cezary Kaliszyk, Lionel Mamane, Iris Loeb, Dimitri Hendriks, and Jasper Stein. I would especially like to thank Nicole Messink for helping me navigate both the university and the Dutch immigration administrations.

From the *Logic and Methodology of Science* group at the University of California at Berkeley, I would like to thank Leo Harrington for being my adviser and providing advice on the incompleteness theorem. I would also like to thank the other post-graduate students who provided lots of tea time discussion: Johanna Franklin, Peter Gerdes, John Goodrick, Alice Medvedev, Kenny Easwaran, and many others.

I would like to give a big thank you to Robert Tupelo-Schneck for introducing me to dependent type theory and to Coq as a way of formalizing mathematics. I would like to thank Nikita Borisov for letting me use some of his computer resources, and I would like to thank Lila Patton Herbst for helping me copy edit my thesis.

Finally, I would like to thank the *Natural Sciences and Engineering Research Council of Canada* (NSERC) for providing me with funding during my work developing my formal proof of the incompleteness theorem.

This thesis has been written using the GNU TeXmacs text editor (see `www.texmacs.org`). The cover design was generated using Gnofract 4D version 3.9.

Table of Contents

Acknowledgements vii

List of Figures xiii

1 Introduction 1

1.1 Main Topics and Contributions 1
    1.1.1 Part I 2
    1.1.2 Part II 2

2 Preliminaries 3

2.1 Dependently Typed Functional Programming 5
    2.1.1 Evaluation 7
    2.1.2 Formalizing Mathematics with a Computer 7
2.2 A Brief Introduction to Type Theory 8
    2.2.1 Inductive types 9
    2.2.2 Constructive Functions 9
    2.2.3 Curried Functions 9
    2.2.4 Anonymous Functions 10
    2.2.5 Extensional Equality 10
    2.2.6 No Subtypes 10
    2.2.7 Predicates Instead of Sets 10
    2.2.8 Setoids Instead of Quotients 11
        2.2.8.1 Rewrite Automation 11
    2.2.9 Apartness Instead of "Not Equal" 11
    2.2.10 Classical Propositions Have One Proof 11
    2.2.11 Omitted Parameters 12
2.3 Coq Specific Issues 12
    2.3.1 Intensional Equality 12
    2.3.2 `Prop` versus `Set` 12
    2.3.3 Opaque Objects 13
    2.3.4 Coq Notation 14

I Incompleteness Theorem 15

3 Introduction 17

4 First-Order Classical Logic 19

4.1 Data Structures for `Language` 19

4.2 Data Structures for `Term` and `Formula` . . . 19
4.3 Definition of `substituteFormula` . . . 23
4.4 Definition of `Prf` . . . 24
4.5 Definition of `SysPrf` . . . 25
4.6 The Deduction Theorem . . . 26
4.7 Substitution Lemmas . . . 27
4.8 Languages and Theories of Number Theory . . . 29

5 Representing Functions . . . 31

5.1 Coding . . . 31
5.2 Primitive Recursive Functions . . . 32
5.3 Creating Primitive Recursive Functions . . . 33
5.3.1 `codeSubFormula` is Primitive Recursive . . . 33
5.3.2 `checkPrf` is Primitive Recursive . . . 36
5.3.3 Efficiency of Primitive Recursive Functions . . . 36
5.4 Representing Primitive Recursive Functions . . . 36

6 The Incompleteness Theorem . . . 39

6.1 Statement of the Incompleteness Theorem . . . 39
6.2 The Fixed Point Theorem . . . 41
6.3 Provability Predicate . . . 41
6.4 Essential Incompleteness of NN . . . 42
6.5 Essential Incompleteness of Peano Arithmetic . . . 42
6.6 Model Theory . . . 43

7 Remarks . . . 45

7.1 Trusting the Proof . . . 45
7.2 Creating a Constructive Proof . . . 45
7.3 Extracting the Sentence . . . 46
7.4 Robinson's System Q . . . 46
7.5 Comparisons with Shankar's 1986 Proof . . . 46
7.6 Gödel's Second Incompleteness Theorem . . . 47
7.7 Proof Development . . . 48
7.8 Statistics . . . 49

II Exact Real Arithmetic . . . 51

8 Introduction . . . 53

8.1 Notation . . . 54
8.2 Rational Numbers . . . 55
8.3 Regular Functions of Rationals . . . 56

9 Metric Spaces . . . 59

9.1 Uniform Continuity . . . 60
9.2 Prelength Spaces . . . 61

9.3 Classification of Metric Spaces . . . 62
9.4 Completion of a Metric Space . . . 63
9.5 Remarks . . . 66
9.5.1 Infinite Distance . . . 66
9.5.2 Dedekind Cuts . . . 66
9.5.3 Identity of Indiscernibles . . . 67
9.5.4 Modulus of Continuity . . . 67
9.5.5 Prelength Space . . . 67
9.5.6 Notation for Projections of Dependent Records . . . 68

10 Completion Is a Monad . . . 69

10.1 Monad Laws . . . 72
10.2 Lifting Binary Functions . . . 74
10.3 Remarks . . . 78
10.3.1 Curried Uniformly Continuous Functions . . . 78
10.3.2 Cartesian but not Closed . . . 78
10.3.3 Errors Found During Formalization . . . 79

11 Real Numbers . . . 81

11.1 Binary Functions of Real Numbers . . . 82
11.2 Inequalities . . . 83
11.3 Reciprocal . . . 83
11.4 Power Series . . . 84
11.4.1 Summing Series . . . 86
11.4.2 π . . . 88
11.5 Improving Efficiency . . . 89
11.5.1 Compression . . . 89
11.5.2 Square Root . . . 89
11.5.3 More Efficient Polynomials . . . 91
11.5.3.1 Implementing More Efficient Polynomials . . . 92
11.5.4 More Efficient Power Series . . . 93
11.5.5 More Efficient Periodic Functions . . . 93
11.5.6 Summing Lists . . . 93
11.6 Correctness . . . 94
11.7 Timings . . . 95
11.8 Solving Strict Inequalities Automatically . . . 95
11.9 Remarks . . . 96
11.9.1 Using this Development . . . 96
11.9.2 Reworking Power Series . . . 97
11.9.3 Regular Functions vs Cauchy Sequences . . . 97
11.9.4 Haskell Prototype . . . 97
11.9.5 Comparisons With Other Implementations . . . 98
11.9.5.1 Comparison with Classical Reals . . . 99

12 Integration over $\mathbb{R}$ . . . 101

12.1 Informal Presentation of Riemann Integration . . . 101
12.1.1 Step Functions . . . 102

12.1.2 Step Functions Form a Monad . . . 104
12.1.3 Applicative Functors . . . 104
12.1.4 The Step Function Applicative Functor . . . 105
12.1.5 Two Metrics for Step Functions . . . 106
12.1.6 Integrable Functions and Bounded Functions . . . 108
12.1.7 Riemann Integral . . . 109
12.1.8 Stieltjes Integral . . . 110
12.1.9 Distributing Monads . . . 111
12.2 Implementation in Coq . . . 111
12.2.1 Glue and Split . . . 111
12.2.2 Equivalence of Step Functions . . . 112
12.2.3 Common Partitions . . . 112
12.2.4 Combinators . . . 113
12.2.5 Lifting Theorems . . . 114
12.2.6 The Identity Bounded Function . . . 115
12.2.7 Correctness . . . 116
12.2.8 Timings . . . 116
12.3 Remarks . . . 116

13 Compact Sets . . . 119

13.1 Product Metrics . . . 120
13.2 Hausdorff Metrics . . . 121
13.2.1 Finite Enumerations . . . 122
13.2.2 Mixing Classical and Constructive Reasoning . . . 123
13.3 Metric Space of Compact Sets . . . 123
13.3.1 Correctness of Compact Sets . . . 124
13.3.2 Distribution of $\mathfrak{F}$ over $\mathfrak{C}$ . . . 126
13.3.3 Compact Image . . . 126
13.4 Plotting Functions . . . 127
13.4.1 Graphing Functions . . . 127
13.4.2 Rasterizing Compact Sets . . . 127
13.4.3 Plotting the Exponential Function . . . 127
13.5 Alternative Hausdorff Metric Definition . . . 128
13.6 Compactness and Computability . . . 129
13.6.1 A Constructive Approach to Filled Julia Sets . . . 132

14 Conclusion . . . 133

Bibliography . . . 135

Glossary of Symbols . . . 141

Index . . . 143

Samenvatting . . . 147

Curriculum Vitae . . . 151

List of Figures

Verifying the trace of the computation of substitution in the case of $\forall y.\, \varphi$. 35

Inductive definition of lazy natural numbers for Coq. 87

The Coq function `iterate_pos` iterates `F` a given number of times. 87

Given two step functions f and g, the step function $f \rhd o \lhd g$ is f squeezed into $[0,o]$ and g squeezed into $[o,1]$. 102

Given a uniformly continuous function f and a step function s_4 that approximates the identity function, the step function $\mathsf{map}(f)(s_4)$ (or f♂s_4) approximates f in the familiar Riemann way. 109

A theorem in Coq stating that a plot on a 42 by 18 raster is close to the graph of the exponential function on $[-6, 1]$. 128

One of these images represents the computable set $\mathcal{K}_{\frac{1}{4}+\epsilon}$, but which one? 131

Chapter 1
Introduction

In the late nineteenth and early twentieth century, formal systems for mathematical reasoning were developed by pioneers such as Frege [Frege, 1879], Russell, and Whitehead [Whitehead and Russell, 1925]. In principle all mathematical arguments could be written in some formal system and verifying the validity of these proofs would be a simple, but tedious, syntactic check. No person would want to engage in such a task, but modern computers excel at quickly performing simple, tedious tasks. Automated proof checkers have been developed that can quickly verify the validity of mathematical arguments that are written in their formal systems.

Proof checkers are an ideal way of verifying mathematical arguments because they are fast and reliable. What remains difficult is converting standard mathematical arguments on paper into a formal language that is suitable for software verification. In a formal proof, no details of an argument can be omitted.

To aid people in the composition of formalized mathematics, proof assistants have been developed. These proof assistants contain an automated proof checker in the kernel of their code. The rest of the software supports the interactive development of formal proofs. They manage what goals remain to be proven and which assumptions are available to prove those goals. They provide tools for automatically solving or reducing certain goals. However, even with all this assistance, creating formal proofs is a difficult and time consuming task for all but the simplest mathematical arguments. In this thesis I will develop some new tools that will lift some of this burden.

1.1 Main Topics and Contributions

This thesis is divided into two parts. In Part I, I discuss my development and verification of the Gödel-Rosser incompleteness theorem, which I undertook in order to better understand the difficulties of creating formal proofs. This new formal proof is more general than the previous formal proof of the incompleteness theorem [Shankar, 1994]. In Part II, using lessons learned about formalization from Part I, I develop a constructive theory of real numbers. This formalization extends previous work on constructive real analysis [Cruz-Filipe, 2004] by adding feasible evaluation of approximations of elementary real number expressions. The two parts are independent of each other and can be read separately. The theorems in this thesis have been developed and verified with the Coq proof assistant [The Coq Development Team, 2004].

1.1.1 Part I

I chose to verify the Gödel-Rosser incompleteness theorem because the proof is simple enough that it is covered in undergraduate mathematics courses, and I felt it was an extreme example of a mathematical proof whose details are omitted during its presentation. I tried to follow a traditional mathematical argument, and I mostly succeeded, but I learned that many of the traditional structures and arguments for this proof are difficult to encode in a formal proof. I have identified several aspects of the proof where non-traditional structures and arguments may be easier to use. Another lesson I learned is that one should spend time researching and thinking about how to simplify aspects of a proof before formalizing it. This pressure to invent more general data structures and proofs is a useful side effect of the formalization process.

1.1.2 Part II

In Part II, I develop a new approach to metric spaces. I define the completion of an arbitrary metric space, and I give detailed paper proofs of properties of my metric spaces. The paper proofs were written before formalization so that I could anticipate difficulties and adjust definitions. I then develop the real numbers as the analytic completion of the rationals and define common trigonometric operations on them. Because my data structures and functions have reasonable efficiency, the operations are feasible to execute. Formal proofs ensure that the operations are correct.

The primary purpose of developing a completion operation for metric spaces was to create an efficient real number implementation. However, the resulting completion operation is generic and can be used to complete other metric spaces. To illustrate this, I develop two other complete metric spaces: the completion of rational step functions, yielding integrable functions, and the completion of finite sets, yielding the compact sets.

The library I have developed could be used to create a scientific calculator that can evaluate real number expressions to arbitrary precision and guarantee that the results are correct. Furthermore, by using the theory of compact sets, one could create a graphing calculator with provably correct plots of functions. Because the computations are certified with formal proofs, the computations can be used inside proofs.

Using computation inside proofs is a powerful technique. The proof of the four colour theorem [Apple and Haken, 1976] and Kepler's conjecture [Hales, 2002] both make heavy use of computation in their proofs. Verified computation is an essential ingredient needed to formalize these proofs. Gonthier has successfully used this technique to verify the four colour theorem with the Coq proof assistant [Gonthier, 2005]. I believe that my theory of real numbers provides the tools needed to verify Hales's proof of the Kepler conjecture as well as other proofs that make use of real number computation, such as the disproof of Mertens conjecture [Odlyzko and teRiele, 1985]. More importantly, I believe the ideas in this part will prove useful for creating the next generation of effective implementations of constructive analysis.

Chapter 2
Preliminaries

Constructive logic is usually presented as a restriction of classical logic where proof by contradiction and the law of the excluded middle are not allowed. However, constructive logic can also be presented as an extension of classical logic.

Consider formulas constructed from universal quantification ($\forall$), implication ($\Rightarrow$), conjunction ($\wedge$), true ($\top$), false ($\bot$), and equality for natural numbers ($=_{\mathbb{N}}$). Define negation using these primitives.

Definition 2.1. $\neg\varphi := \varphi \Rightarrow \bot$.

We call a formula φ *stable* when $\neg\neg\varphi \Rightarrow \varphi$ holds. One can (constructively) prove φ is stable for any formula φ generated from this set of connectives by induction on the structure of φ because $m =_{\mathbb{N}} n$ is provably stable, and the connectives listed above preserve stability. Thus, one can deduce classical results with constructive proofs for formulas generated from this restricted set of connectives.

This set of connectives is not really restrictive though; it can be used to define the other classical connectives.

Definition 2.2. *Classical connectives:*

$$\begin{array}{rcl} \varphi \mathbin{\tilde{\vee}} \psi & := & \neg(\neg\varphi \wedge \neg\psi) \\ \tilde{\exists} x.\, \varphi(x) & := & \neg(\forall x. \neg\varphi(x)). \end{array}$$

With this full set of connectives, one can produce classical mathematics. The law of the excluded middle ($\varphi \mathbin{\tilde{\vee}} \neg\varphi$) has a constructive proof when the classical disjunction is used. As long as we have a stable goal we can do case analysis on the classical disjunction.

Theorem 2.3. *Assuming that $\varphi \mathbin{\tilde{\vee}} \psi$ holds, we have $\varphi \Rightarrow \theta$ and $\psi \Rightarrow \theta$, and θ is stable, meaning $\neg\neg\theta \Rightarrow \theta$, then θ holds.*

Proof. Because θ is stable, it is sufficient to prove that $\neg\neg\theta$ holds. Assume that $\neg\theta$ holds. From this and the contrapositive of $\varphi \Rightarrow \theta$ we can conclude that $\neg\varphi$ holds. Similarly $\neg\psi$ holds. But we have that $\neg(\neg\varphi \wedge \neg\psi)$ holds from the definition of $\varphi \mathbin{\tilde{\vee}} \psi$. Thus we have a contradiction, as required. □

We have a similar theorem for the classical existential.

Theorem 2.4. *Assuming that* $\tilde{\exists} x.\, \varphi(x)$ *holds, we have* $\forall x.\, \varphi(x) \Rightarrow \theta$*, and* θ *is stable, meaning* $\neg\neg\theta \Rightarrow \theta$*, then* θ *holds.*

Proof. Because θ is stable, it is sufficient to prove that $\neg\neg\theta$ holds. Assume that $\neg\theta$ holds. From this and the contrapositive of $\varphi(x) \Rightarrow \theta$ we can conclude that $\forall x. \neg\varphi(x)$ holds. But we have that $\neg(\forall x. \neg\varphi(x))$ holds from the definition of $\tilde{\exists} x.\, \varphi(x)$. Thus we have a contradiction, as required. □

So far, these two theorems can always be used because for any given formula θ, we can prove $\neg\neg\theta \Rightarrow \theta$.

Now let us extend this logic by adding two new connectives, the constructive disjunction ($\vee$) and the constructive existential ($\exists$). These new connectives come equipped with their constructive rules of inference given by natural deduction [Thompson, 1991]. These constructive connectives are slightly stronger than their classical counterparts. Constructive excluded middle ($\varphi \vee \neg\varphi$) cannot be deduced in general, and our inductive argument that $\neg\neg\varphi \Rightarrow \varphi$ holds no longer goes through if φ uses these constructive connectives. Therefore, statements such as $(\neg\varphi \Rightarrow \neg\psi) \Rightarrow (\psi \Rightarrow \varphi)$ are not general tautologies in constructive logic, but one can prove such statements when φ is a stable formula.

We wish to use constructive reasoning because constructive proofs have a computational interpretation. A constructive proof of $\varphi \vee \psi$ tells which of the two disjuncts hold. A proof of $\exists n \colon \mathbb{N}.\, \varphi(n)$ gives an explicit value for n that makes $\varphi(n)$ hold. Most importantly, we have a functional interpretation of $\Rightarrow$ and $\forall$. A proof of $\forall n \colon \mathbb{N}. \exists m \colon \mathbb{N}.\, \varphi(n, m)$ is interpreted as a function with an argument n that returns an m paired with a proof of $\varphi(n, m)$.

The classical fragment also admits this functional interpretation, but formulas in the classical fragment typically end in ... $\Rightarrow \perp$. These functions take their arguments and return a proof of false. Of course, there is no proof of false, so it must be the case that the arguments cannot simultaneously be satisfied. Therefore, these functions can never be executed. It turns out that only trivial functions such as this are created by proofs of classical formulas. This is why constructive mathematics aims to strengthen classical results. We wish to create proofs with non-trivial functional interpretations.

Constructive logic turns out to be compatible with classical logic. If one replaces the constructive existential and disjunction with their classical counterparts, then the resulting formula is a theorem if the original theorem was. In fact, the deductive rules of constructive logic are a subset of the deductive rules of classical logic when connectives are reinterpreted this way. This means that constructive reasoning can be understood by classical mathematicians, although classical mathematicians may find some of the reasoning unusual at times because they have a different interpretation of the connectives.

Constructive mathematics is mathematics done with constructive logic. Some care needs to be taken when selecting the non-logical axioms to ensure that the computational interpretation of the logic remains valid. Standard ZFC set theory used in mathematics implicitly assumes a classical logic, so it is not a useful choice; one would be essentially restricted to the classical portion of constructive logic. I will use a type theory called the *calculus of (co)inductive constructions* (CiC) [The Coq Development Team, 2004] as the mathematical foundation for this thesis. CiC adds inductive and coinductive data structures, such as natural numbers, lists, and streams, to constructive logic in a way that is compatible with the existing natural deduction style rules for constructive logic. In fact, most of the logical connectives are defined as inductive datatypes. Only ($\forall$) remains primitive and ($\Rightarrow$) is defined as a special case of ($\forall$) (see Section 2.2.2).

CiC is the foundation used by the Coq proof assistant, which is the system that I have used to verify the theorems in my thesis. I will attempt to present my work independent of any particular constructive foundation as much as possible so that the ideas can easily be transferred to other systems. However, I will make use of the notation from type theory in my presentation.

From now on, I will generally leave out the word "constructive" from phrases like "constructive disjunction" and "constructive existential" and simply write "disjunction" and "existential". This follows the standard practice in constructive mathematics of using names from classical mathematics to refer to some stronger constructive notion. I will explicitly use the word "classical" when I wish to refer to classical concepts. Rather than saying a statement is *true* in constructive logic, which suggests a *true/false* dichotomy, I will say a statement *holds* or is *proved*. I may also say a statement is *constructed*, but I will usually reserve this term for constructive theorems outside the classical subset.

2.1 Dependently Typed Functional Programming

This computational interpretation of constructive deductions is given by the Curry-Howard isomorphism [Thompson, 1991]. This isomorphism associates formulas with dependent types, and proofs of formulas with functional programs of the associated dependent types. For example, the identity function $\lambda x\colon A. x$ of type $A \Rightarrow A$ represents a proof of the tautology $A \Rightarrow A$. Table 2.1 lists the association between logical connectives and type constructors.

Logical Connective	**Type Constructor**
implication: $\Rightarrow$	function type: $\Rightarrow$
conjunction: $\wedge$	product type: $\times$
disjunction: $\vee$	disjoint union type: $+$
true: $\top$	unit type: $()$
false: $\bot$	void type: $\emptyset$
for all: $\forall x. \varphi(x)$	dependent function type: $\Pi x. \varphi(x)$
exists: $\exists x. \varphi(x)$	dependent pair type: $\Sigma x. \varphi(x)$

Table 2.1. The association between formulas and types given by the Curry-Howard isomorphism.

In dependent type theory, functions from values to types are allowed. Using types parametrized by values, one can create dependent pair types, $\Sigma x\colon A.\,\varphi(x)$, and dependent function types, $\Pi x\colon A.\,\varphi(x)$. A dependent pair consists of a value x of type A and a value of type $\varphi(x)$. The type of the second value depends on the first value, x. A dependent function is a function from the type A to the type $\varphi(x)$. The type of the result depends on the value of the input.

The logical connectives and the type constructors from Table 2.1 mean the same thing, and I will use them interchangeably; however, I will usually prefer to use the logical connectives. Note that Coq has a sort of distinction between the two classes of connectives (see Section 2.3.2).

The association between logical connectives and types can be carried over to constructive mathematics. We associate mathematical structures, such as the natural numbers, with inductive types in functional programming languages. We associate atomic formulas with functions returning types. For example, we can define equality on the natural numbers, $x =_{\mathbb{N}} y$, as a recursive function:

$$\begin{aligned}
0 =_{\mathbb{N}} 0 &:= \top \\
\mathsf{S}(x) =_{\mathbb{N}} 0 &:= \bot \\
0 =_{\mathbb{N}} \mathsf{S}(y) &:= \bot \\
\mathsf{S}(x) =_{\mathbb{N}} \mathsf{S}(y) &:= x =_{\mathbb{N}} y
\end{aligned}$$

One catch is that general recursion is not allowed when creating functions. The problem is that general recursion allows one to create a fixed point operator $\mathsf{fix} : (\varphi \Rightarrow \varphi) \Rightarrow \varphi$ that corresponds to a proof of a logical inconsistency. To prevent this, we allow only well-founded recursion over an argument with an inductive type. Because well-founded recursion ensures that functions always terminate, the language is not Turing complete. However, one can still express fast-growing functions, such as the Ackermann function, without difficulty by using higher-order functions [Thompson, 1991].

Because proofs and programs are written in the same language, we can freely mix the two. For example, I will represent the real numbers (see Definition 9.16 and Definition 11.1) by the type

$$\exists f\colon \mathbb{Q}^+ \Rightarrow \mathbb{Q}. \forall \varepsilon_1\, \varepsilon_2. |f(\varepsilon_1) - f(\varepsilon_2)| \leq \varepsilon_1 + \varepsilon_2. \tag{2.1}$$

A value of this type is a pair of a function $f\colon \mathbb{Q}^+ \Rightarrow \mathbb{Q}$ and a proof of $\forall \varepsilon_1\, \varepsilon_2. |f(\varepsilon_1) - f(\varepsilon_2)| \leq \varepsilon_1 + \varepsilon_2$. The idea is that a real number is represented by a function f that maps any requested precision $\varepsilon\colon \mathbb{Q}^+$ to a rational approximation of the real number. Not every function of type $\mathbb{Q}^+ \Rightarrow \mathbb{Q}$ represents a real number. Only those functions that have coherent approximations should be allowed. The proof object paired with f witnesses the fact that f has coherent approximations. This is one example of how mixing functions and formulas allows one to create precise datatypes.

2.1.1 Evaluation

This computational interpretation of constructive mathematics is more than a nice theoretical feature. The algorithms contained inside constructive proofs can actually be executed inside the type theory. One can view this as part of Bishop's program to see constructive mathematics as a programming language. For example, the expression $2 + 2$ and the expression 4 are considered equivalent. They can be substituted for each other in any context, and the system will evaluate $(2+2)$ to 4 when necessary.

A technique called reflection [Barendregt and Geuvers, 2001] uses this evaluation mechanism to allow one to create proofs by using computation. One example of this technique is used in Section 11.4.2 to prove the correctness of my definition of π. Thus the computational and logical parts of type theory enhance each other. Correct algorithms can be used to create proofs, and proofs verify the logical correctness of algorithms.

2.1.2 Formalizing Mathematics with a Computer

There are several different approaches to formalizing mathematics on a computer [Wiedijk, 2006]. Often people use a system that provides classical logic. Many systems implement classical higher-order logic, such as HOL Light, Isabelle/HOL, and PVS. A few systems implement classical set theory, such as Mizar and Isabelle/ZF.

One of the main benefits of formalizing mathematics with a computer, aside from providing high assurance of correctness, is the ability for the computer to automate reasoning that would be too tedious to formalize by hand. This automated reasoning can range from the mundane, automatically differentiating an expression, to the extreme, the huge case analysis needed in the four colour theorem [Gonthier, 2005].

Systems based on classical logic must perform this computation outside the logic, whereas a constructive system, such as Coq, can perform computation both on the outside, as Coq does with its tactic system, and on the inside, as Coq does with its evaluation mechanism.

To prove the correctness of an algorithm using a classical system, generally one writes out the function first. Once the function is defined, one can use classical reasoning to prove properties about that function. In a constructive system, one has the option to mix proofs into the definition of the function. This allows one to prove properties of a function as the function is defined. However, one still has the option of writing the function first and then reasoning about it.

One can see the classical approach to correctness as taking a constructive theorem and Skolemizing the constructive existentials (and constructive disjunctions). The functions one initially writes become the instances of the Skolem functions in the transformed theorem. The resulting theorem is in the classical fragment of constructive logic and classical arguments can be used to prove it.

I do not believe that there is anything to be gained by forcing a separation between the (classical) logical aspects and the algorithmic aspects when proving properties of algorithms. Sometimes this separation is a useful way to proceed, but usually the "flow" of the proof follows the "flow" of data in the algorithm. It makes sense to attach proof objects to values inside the algorithm, particularly when one wants to express an invariant of a datatype. I feel working with data and proofs together clarifies the development of algorithms and makes proving properties easier. Sometimes one wants to prove a property of how two or more functions interact; for example, when proving one function distributes over another. In this case, it is usually best to provide a proof separate from either function.

When working in a constructive system, one has the option to avoid algorithmic development entirely and simply write a constructive proof. By its constructive nature, this proof can be executed to compute constructive witnesses. However, if no attention is paid to the construction, the resulting algorithms are often inefficient. Therefore, it is wise to think of creating a constructive proof as a programming task as well as a mathematical task. One should take the same algorithmic considerations into account that one would when writing any other functional program.

2.2 A Brief Introduction to Type Theory

Types in type theory are similar to sets in set theory in that they are a collection of values. One important difference between type theory and set theory is that in type theory every value belongs to a unique type. I write $a\colon A$ when a value a belongs to a type A. This means that the natural number $1_{\mathbb{N}} : \mathbb{N}$ is distinct from the rational number $1_{\mathbb{Q}} : \mathbb{Q}$ because they belong to different types. However, I will often just write 1 and leave the distinction implicit. Values are also called "objects"; the two terms are synonymous.

Quantifiers always range over some type; for example, $\forall x\colon \mathbb{N}. x = 0 \vee \exists y\colon \mathbb{N}. x = \mathsf{S}(y)$. Sometimes I will omit writing the type for the quantified variable when it is clear from context.

One can form new types from existing types. Given types A and B, one can create the disjoint union type, $A + B$, the Cartesian product type, $A \times B$, and the type of functions from A to B, $A \Rightarrow B$. These types are the same as the logical connectives $(\vee)$, $(\wedge)$, and $(\Rightarrow)$ respectively. The functions $\pi_1\colon A \times B \Rightarrow A$ and $\pi_2\colon A \times B \Rightarrow B$ are the first and second projection functions of the Cartesian product type.

In type theory, proofs are objects, and propositions are types. This is known as the propositions-as-types interpretation. Propositions are types whose objects consist of all of their proofs. Therefore, one writes $p\colon \varphi$ when p is a proof of φ. When a proposition has no proofs, it has no members. Thus it is the inhabited propositions that are the valid propositions. Propositions are also values that belong to the type of propositions. In Coq, the type of propositions is called `Prop` (see Section 2.3.2); however I will denote this type as $\star$ and write $\varphi\colon \star$. Predicates can be seen as functions whose codomains are propositions. For example, predicates over the natural numbers will have the type $\mathbb{N} \Rightarrow \star$.

2.2.1 Inductive types

CiC has inductive and coinductive types, so standard inductive types such as Booleans, natural numbers, integers, lists, binary trees, etc. are available as well as coinductive types such as infinite streams. I will use standard mathematical notation for such types.

A record is an inductive data type that has only one constructor. Coq has special notation for defining records that allows one to create the inductive type and the projection functions for its fields at the same time.

A value of an inductive data type can be consumed by a structurally recursive function. These recursive functions can be evaluated inside type theory as mentioned in Section 2.1.1.

In addition to inductive types, inductive type families (also called inductive predicates) can be defined. In this case, the inductive definition produces not a type but a predicate. See Section 4.4 for an example of an inductively defined type family.

2.2.2 Constructive Functions

In classical mathematics, a classical function is a type of binary relation F satisfying the property

$$(\forall x. \tilde{\exists} y. F(x, y)) \wedge (\forall x y z. F(x, y) \wedge F(x, z) \Rightarrow y = z).$$

In constructive mathematics, a (constructive) function is a value of type $A \Rightarrow B$, where $\Rightarrow$ is a primitive connective. This $\Rightarrow$ connective used in the function type is identical to the implication logical connective. In type theory, values of this constructive function type are formed by specifying how the output is constructed from the input, usually using a lambda expression (see Section 2.2.4). Constructive functions can be evaluated internally in type theory as mentioned in Section 2.1.1.

Additionally there is the dependent function type, written $\forall a{:}\ A. B(a)$, which is identical to the universal quantifier. In this case the codomain of the function can depend on the input parameter. Like the regular function type, values of a dependent function type are specified by lambda expressions and can be evaluated internally in type theory. In fact, $A \Rightarrow B$ is simply notation for $\forall a{:}\ A.B$ where B does not depend on a.

2.2.3 Curried Functions

A multi-argument function in mathematics is typically represented as a function whose domain is a Cartesian product. For example, a binary function would have the type $A \times B \Rightarrow C$. It is traditional in functional programming to use curried functions. A curried binary function has type $A \Rightarrow B \Rightarrow C$ (meaning $A \Rightarrow (B \Rightarrow C)$ by the usual associativity rule for $\Rightarrow$), which can easily be shown to be isomorphic to $A \times B \Rightarrow C$. When applying a curried binary function it would be more technically accurate to write $f(a)(b)$, but for clarity I will sometimes write $f(a, b)$ as if the function were not curried.

2.2.4 Anonymous Functions

I will on occasion use anonymous functions. Anonymous functions—common in functional programming—are declared using the λ symbol. For example, the reciprocal function can be written as $\lambda x.x^{-1}$ and can be applied to arguments, as in $\left(\lambda x.x^{-1}\right)\left(\frac{2}{3}\right)$ is equal to $\frac{3}{2}$.

2.2.5 Extensional Equality

In this thesis I will use the equality sign $(=)$ for *extensional equality*. Two functions f, g of the same type are considered extensionally equal when, for any input given to both functions, the outputs of the functions are extensionally equal:

$$f = g := \forall a.\, f(a) = g(a)$$

Two values of an inductive type are extensionally equal when their constructors are the same and all parameters are extensionally equal.

Extensional equality is the finest equality we will need in my thesis. However, Coq uses a finer equality called intensional equality (see Section 2.3.1) for its fundamental equality.

Another sort of equality that we will frequently use is setoid equality (see Section 2.2.8), which is generally coarser than extensional equality.

2.2.6 No Subtypes

Coq does not have subtyping.[2.1] This means that $\mathbb{Z}$ cannot be considered as a subtype of $\mathbb{Q}$. Functions, usually injections, are instead used to change data from one type to the other type. Coq has a coercion mechanism that will automatically infer many of these injections and hide the injections from being printed. Defining a coercion from $\mathbb{Z}$ to $\mathbb{Q}$ allows one to use values of $\mathbb{Z}$ as if they were values of $\mathbb{Q}$.

2.2.7 Predicates Instead of Sets

Sets do not exist as such in type theory. Usually sets are replaced by predicates over a particular type. For example, subsets of the natural numbers are identified with $\mathbb{N} \Rightarrow \star$, predicates on the natural numbers. A value x satisfies a predicate P when $P(x)$ is an inhabited type. Often I will use set theoretic notation to denote membership of a predicate:

$$\begin{array}{rcl} x \in P & := & P(x) \\ \forall x \in P.\varphi & := & \forall x.P(x) \Rightarrow \varphi \\ \exists x \in P.\varphi & := & \exists x.P(x) \wedge \varphi \end{array}$$

Similar quantifier notation will be used for inequalities such as $\forall i < n.\, \varphi$ and $\exists i \geq n.\, \varphi$.

[2.1] Technically, Coq has subtyping in its universe levels, but we will not concern ourselves with universe levels in this thesis.

2.2.8 Setoids Instead of Quotients

A quotient type is a type modulo a given equivalence relation on that type. For instance, the type $\mathbb{Q}$ is often considered as a quotient of the type $\mathbb{Z} \times \mathbb{N}^+$. CiC does not have quotient types. One instead passes around the equivalence relation in question. To do this, one often uses a data structure called a setoid. A setoid $(A, \asymp_A)$ is a type paired with an equivalence relation on that type. Functions between setoids that preserve their equivalence relations are called *respectful*. Proving that a function is respectful consists of the same work in traditional mathematics needed to prove that a function over quotients is well-defined. Respectful functions are also called *morphisms*.

2.2.8.1 Rewrite Automation

Coq has special support for reasoning about setoids through its `setoid_rewrite` and `setoid_replace` tactics [Coen, 2004]. These tactics will automatically create the deductions for substitution of setoid equivalent terms into respectful functions and relations. This support makes reasoning about setoid equivalence almost as easy as reasoning about equality in Coq.

Furthermore, Coq has the ability to define a database of rewrite lemmas. These lemmas have terms of the form $a \asymp_A b$ for their conclusions. When they are added to the database the user indicates which way substitution should be performed (the same lemma can be added to different databases with different directions). The user can then use the database as a rewrite system to process a hypothesis or goal. The `autorewrite <database>` tactic will repeatedly try to use the lemmas in the named database to rewrite the goal. Well crafted rewrite databases can be used to quickly transform or simplify expressions.

2.2.9 Apartness Instead of "Not Equal"

In analysis, when considering Hausdorff spaces, it is often the case that being "not equal" is a non-stable atomic formula [Bauer and Taylor, 2008]. However, $x \neq y$ is notation for $\neg x = y$, and it is always stable. Therefore, constructive analysis uses a relation called *apartness*, written $x \# y$, which is not necessarily stable. For example, the apartness relation for real numbers, $x \# y := x < y \vee y < x$, is not provably stable. In these cases a *c-setoid* (a constructive setoid) data structure is often used instead of a setoid. A c-setoid $(A, \#_A)$ is a type paired with an apartness relation. In this case, a *respectful* function from a c-setoid A to a c-setoid B, is a function $f: A \Rightarrow B$ such that $\forall x, y: A. f(x) \#_B f(y) \Rightarrow x \#_A y$.

2.2.10 Classical Propositions Have One Proof

All classical propositions have at most one proof, meaning that if $p: \varphi$ and $q: \varphi$ are two proof objects of the same classical proposition φ, then we can prove that p and q are (extensionally) equal, $p = q$. In general, I will denote all the unique proofs of classical propositions as $*$ (e.g. $*: \top$ and $*: \bot \Rightarrow \varphi$).

2.2.11 Omitted Parameters

Throughout this thesis there are many places where I formally need to provide proofs of propositions as arguments to functions. However, to allow for a clearer presentation, I will usually omit these parameters. For example, the logarithm on rational numbers, $\ln_{\mathbb{Q}}(a)$, requires a proof object showing that $0 <_{\mathbb{Q}} a$, but I do not show this parameter (see Section 11.4).

2.3 Coq Specific Issues

The previous section detailed the properties of the type theory I will be informally working in for this thesis. Although I made reference to the Coq system at times, the type theory I describe is fairly generic and should be easily adapted to work in many different implementations of type theory. However, my formal work has been done in the Coq proof assistant, so below I address some specific properties of Coq that are relevant to my formalization.

2.3.1 Intensional Equality

In type theory, functions are represented by the program that does the computation. In Coq, two functions can only be proven equal if after normalization they are the same program. This means that if two functions are extensionally equal, they may not be intensionally equal in Coq.

For inductive data types without functions as parameters to constructors, intensional equality coincides with extensional equality, so much of the time this distinction does not matter. On those occasions where I need to deal with extensional equality between functions in Coq, I can often just expand out the definition of extensional equality. Thus, rather than trying to prove $f = g$, I prove $\forall a. f(a) = g(a)$ instead.

2.3.2 `Prop` versus `Set`

The Coq system has a special universe called `Prop`. The `Prop` universe is like $\star$ in that it has types as members. The regular type universe is called `Set` and also has types as members, like $\star$.[2.2] These two universes behave largely the same. Both universes have inductive types and function/implication types. When defining an inductive type, one specifies which universe the inductive type will inhabit. Functions in both universes can be evaluated inside Coq, as long as the functions are not opaque (see Section 2.3.3).

[2.2] There is also an infinite number of universe levels containing `Set` and `Prop`, but we can safely ignore this for our needs.

Generally the two universes serve to divide up types that are otherwise identical according to whether their values are intended to play a logical or functional role (recall Table 2.1). For each entry in Table 2.1, Coq has at least two operators, one for the logical side and one for the functional side. For example, Coq has one product for logical conjunction and another product for product types. Inductive types for both universes can have parameters whose types belong to either universe. In this way types from both universes can be mixed together. The only major difference between the `Prop` and `Set` universes is in the way that case analysis for inductive types behave. Case analysis on an inductive type from the `Prop` universe is not allowed to result in a value in the `Set` universe. This prevents information from flowing from the `Prop` universe to the `Set` universe. This restriction allows Coq to optimize program extraction by throwing away all values from the `Prop` universe during program extraction [Paulin-Mohring, 1989]. This is safe because the case analysis restriction means that no value from the `Set` universe can depend on choices made in the `Prop` universe.

There is an important exception to this case analysis rule. For the special case of inductive types declared in the `Prop` universe that have at most one constructor whose parameters are all from the `Prop` universe, case analysis is allowed to return values in the `Set` universe. Case analysis is safe in this instance because inductive types with at most one constructor provide no information; There will be only one branch in the case analysis. During program extraction, the case analysis is simply replaced by the code of this one branch. Since all the parameters are from the `Prop` universe, no parameters from the case analysis will occur in the code extracted from the one branch.

Generally I only put inductive types in the `Prop` universe if they satisfy this special condition.[2.3] This allows me to enjoy unrestricted case analysis and still allow for efficient program extraction.

I will not be making a `Prop`/`Set` distinction in my general discussion; however, I may refer to it when making comments specifically about the Coq implementation.

2.3.3 Opaque Objects

Theorems and definitions can be marked as *opaque* in Coq. Opaque definitions cannot be unfolded during computation. In this sense they behave much like axioms in Coq. Generally I make all objects in the `Prop` universe opaque and leave all objects in the `Set` universe as transparent. This usually works well because I ensure that all my objects from the `Prop` universe have no information (see Section 2.3.2).

[2.3] There are some cases when I would put inductive type families in `Prop`, even if they have multiple constructors. However, all inductive type families in this thesis will be put in the `Set` universe.

One notable exception to making all `Prop` objects opaque would be a proof that a relation is well-founded. Even though the accessibility predicate is in the `Prop` universe and satisfies the one constructor rule given in Section 2.3.2, it generally needs to be transparent so that well-founded induction over the relation can proceed. The other notable exception is that proofs of equality occasionally need to be transparent to allow "safe type-casts"[2.4] to be evaluated.

Declaring objects as opaque is useful because it can prevent computation from expanding irrelevant definitions during evaluation. This is so useful that Coq also has a mechanism to temporarily mark certain definitions as opaque and make them transparent again later.

2.3.4 Coq Notation

On occasion I will use Coq's concrete syntax to clearly specify the formal theorems I have proven. For those not familiar with Coq syntax, here is a short list of notation.

- `->`, `/\`, `\/`, and `~` are the logical connectives $\Rightarrow$, $\wedge$, $\vee$, and $\neg$.
- `A + B`, `A * B`, and `A -> B` form disjoint union types, Cartesian product types, and function types.
- `*`, `+`, and `S` are the arithmetic operations of multiplication, addition, and successor.
- `inl` and `inr` are the left and right injection functions of types `A -> A + B` and `B -> A + B`.
- `::`, and `++` are the list operations cons and append.
- `_` is an omitted parameter that Coq can infer itself.

Function application is indicated by juxtaposition. Thus $f(x)$ is written simply as `f x` in Coq. For more details see the Coq 8.0 reference manual [The Coq Development Team, 2004].

[2.4] A safe type-cast allows a: A to be transformed into a value of type B when given a proof of $A = B$.

Part I

Incompleteness Theorem

Chapter 3

Introduction

In this part, I discuss my formal proof of the Gödel-Rosser incompleteness theorem for arithmetic. The proof shows that any complete first-order theory of a suitable axiom system using only the symbols $+$, $\times$, $\mathbf{0}$, $\boldsymbol{S}$, and $<$ is inconsistent. The axiom system must contain the nine axioms of a system called NN. These nine axioms define the five symbols. The axiom system must also be expressible in itself. This restriction prevents the incompleteness theorem from applying to axioms systems such as the true first order sentences in $\mathbb{N}$. Finally, the axiom system must be decidable, so that it is constructively decidable what is and is not a proof. This restriction is subtly, but significantly, different from requiring the axiom system to be computable. The difference is discussed later in Section 6.1.

A computer-verified proof of Gödel's incompleteness theorem is not new. In 1986 Shankar created a proof of the incompleteness of Z2, hereditarily finite set theory, in the Boyer-Moore theorem prover [Shankar, 1994]. My work is the first computer-verified proof of the essential incompleteness of arithmetic. Harrison recently completed a proof in HOL Light [Harrison, 2000] of the essential incompleteness of Σ_1-complete theories, but has not shown that any particular theory is Σ_1-complete.

My proof was developed and checked in Coq 7.3.1 using Proof General under XEmacs. It is part of the user contributions to Coq and can now be checked in Coq 8.0 [The Coq Development Team, 2004]. Examples of source code in this document use the new notation introduced in Coq 8.0.

This part points out some of the more interesting problems I encountered formalizing the incompleteness theorem. My proof loosely follows the presentation of incompleteness given in my undergraduate logic textbook, *An Introduction to Mathematical Logic* by Hodel [Hodel, 1995]. I use a more standard Hilbert-style deduction system rather than the one defined by Hodel. I referred to the supplementary text for the book *Logic for Mathematics and Computer Science* [Burris, 1997] to construct Gödel's β-function, and I take some inspiration from Shankar's work as described in *Metamathematics, Machines, and Gödel's Proof* [Shankar, 1994]. I also use Caprotti and Oostdijk's contribution of Pocklington's criterion [Caprotti and Oostdijk, 2001] to prove the Chinese remainder theorem. My proof is entirely constructive.

In this part, I first discuss my formalization of first-order classical logic over an arbitrary language (Chapter 4). This is followed by the definition of an axiom system called NN and the axiom system for Peano arithmetic. Then I discuss coding formulas and proofs as natural numbers followed by a discussion about primitive recursive functions operating on these codes (Chapter 5). Next I give the statement of the essential incompleteness of NN (Chapter 6). I discuss the fixed point theorem, Rosser's incompleteness theorem, and the incompleteness of Peano arithmetic. Finally, I give some general remarks about my experience and how to extend my work in order to formalize Gödel's second incompleteness theorem (Chapter 9.5).

This part is an expansion of my publication, "Essential Incompleteness of Arithmetic Verified by Coq" [O'Connor, 2005a].

Chapter 4
First-Order Classical Logic

The first step to prove the incompleteness theorem is to develop a theory of first order logic inside Coq. In essence Coq's logic is a formal metalogic to reason about this internal logic. I created inductive types of well-formed terms and formulas parametrized over a first order language. I then created an inductive definition of Hilbert-style deductions for classical logic.

I make essential use of Coq's dependent type system. `Terms` and `Formulas` are types that depend on the language. Also, dependent types are used in the definition of function and relation symbols so that the type system enforces that all terms and formulas be well-formed. This approach differs from Harrison's definition of first order terms and formulas in HOL Light [Harrison, 1998a] because HOL Light does not have dependent types.

4.1 Data Structures for `Language`

I defined `Language` to be a dependent record of types for symbols and an arity function from symbols to $\mathbb{N}$. The Coq code is:

```
Record Language : Type := language
  {Relations : Set;
   Functions : Set;
   arity : Relations + Functions -> nat}.
```

In retrospect it would have been slightly more convenient to use two arity functions instead of using the disjoint union type.

The entire development of first-order logic is parametrized over this structure by using Coq's section mechanism.

4.2 Data Structures for `Term` and `Formula`

For any language, a `Term` is either a variable indexed by a natural number or a function symbol plus a list of n terms where n is the arity of the function symbol. My first attempt at writing this in Coq failed.

```
Variable L : Language.
(* Invalid definition *)
Inductive Term0 : Set :=
  | var0 : nat -> Term0
  | apply0 : forall (f : Functions L) (l : List Term0),
       (arity L (inr _ f))=(length l) -> Term0.
```

Some people may think that the occurrence of the List Term0 type in apply0 causes Coq to reject this definition; however, this is actually not problematic. It is the type (arity L (inr _ f))=(length l) that fails to meet Coq's positivity requirement for inductive types. Expanding the definition of length reveals a hidden occurrence of Term0 which is passed as an implicit argument to length. It is this occurrence that violates the positivity requirement.

My second attempt met the positivity requirement, but it had other difficulties. First, one creates the well-known inductive family of types lists of length n, usually called Vector.

```
Inductive Vector (A : Set) : nat -> Set :=
  | Vnil : Vector A 0
  | Vcons : forall (a : A) (n : nat),
       Vector A n -> Vector A (S n).
```

Using this I could have defined Term as follows.

```
Variable L : Language.

Inductive Term1 : Set :=
  | var1 : nat -> Term1
  | apply1 : forall f : Functions L,
       (Vector Term1 (arity L (inr _ f))) -> Term1.
```

My difficulty with this definition was that the induction principle generated by Coq is too weak to work with. The generated induction principle Term1_ind is:

```
Term1_ind
     : forall P : Term1 -> Prop,
       (forall n : nat, P (var1 n)) ->
       (forall (f : Functions L)
          (v : Vector Term1 (arity L (inr (Relations L) f))),
        P (apply1 f v)) -> forall b : Term1, P b.
```

Term1_ind requires showing P (apply1 f v) for all f and v without any inductive hypothesis about v. If, for instance, one wants prove that a particular variable is not used in a term, one will need to verify that this holds for each term in the vector v; however, the inductive hypothesis for each term of the vector v is not available.

Instead I created two mutually inductive types: Term and Terms.

```
Variable L : Language.
```

```
Inductive Term : Set :=
  | var : nat -> Term
  | apply : forall f : Functions L,
       Terms (arity L (inr _ f)) -> Term
with Terms : nat -> Set :=
  | Tnil : Terms 0
  | Tcons : forall n : nat,
       Term -> Terms n -> Terms (S n).
```

Again the automatically generated induction principle is too weak, but the `Scheme` command generates suitable mutual-inductive principles. For example, the command

```
Scheme Term_Terms_rec := Minimality for Term Sort Set
  with Terms_Term_rec := Minimality for Terms Sort Set.
```

generates a term of the following type.

```
Term_Terms_rec
     : forall (P : Set) (P0 : nat -> Set),
       (nat -> P) ->
       (forall f : Functions L,
        Terms (arity L (inr (Relations L) f)) ->
        P0 (arity L (inr (Relations L) f)) -> P) ->
       P0 0 ->
       (forall n : nat, Term -> P -> Terms n -> P0 n -> P0 (S n)) ->
       Term -> P.
```

In this case we have two "result" types. `P` is the result of evaluating on `Term`, while `P0` is the result of evaluation on `Terms`.

The disadvantage of this approach is that useful lemmas about `Vectors` must be reproved for `Terms`. Some of these lemmas are quite tricky to prove because of the dependent type. For example, proving `forall x : Terms 0, Tnil = x` is surprisingly difficult.

It turns out that the `Term1` definition would have been adequate. One can explicitly make a sufficient induction principle by using nested `Fixpoint` functions [Marche, 2005].

```
Section RecursionDef.

Variable P : Set.
Variable P0 : nat -> Set.

Variable varcase : nat -> P.
Variable applycase :
  forall f : Functions L, Vector Term1 (arity L (inr _ f)) ->
     P0 (arity L (inr _ f)) -> P.
```

```
Variable nilcase : P0 0.
Variable conscase :
  forall n : nat, Term1 -> P -> Vector Term1 n -> P0 n -> P0 (S n).

Fixpoint Term1_rec_new (t : Term1) : P :=
let fix Terms1_rec (n : nat)(vec : (Vector Term1 n)) {struct vec} :
          P0 n :=
 match vec in (Vector _ n) return (P0 n) with
 | Vnil => nilcase
 | Vcons term m terms =>
    conscase m term (Term1_rec_new term) terms (Terms1_rec m terms)
 end
in
 match t with
 | var1 n => varcase n
 | apply1 f terms => applycase f terms (Terms1_rec _ terms)
 end.

End RecursionDef.
```

I would recommend using something like the Term1 definition with Term1_rec_new in future work.

The definition of Formula was straightforward.

```
Inductive Formula : Set :=
  | equal : Term -> Term -> Formula
  | atomic : forall r : Relations L, Terms (arity L (inl _ r)) ->
               Formula
  | impH : Formula -> Formula -> Formula
  | notH : Formula -> Formula
  | forallH : nat -> Formula -> Formula.
```

I defined the other logical connectives in terms of impH, notH, and forallH.

```
Definition orH (A B : Formula) := impH (notH A) B.
Definition andH (A B : Formula) := notH (orH (notH A) (notH B)).
Definition iffH (A B : Formula) := andH (impH A B) (impH B A).
Definition existH (x : nat) (A : Formula) :=
  notH (forallH x (notH A)).
```

The equals relation is part of the definition of Formula, so it does not occur as a relation symbol in the language. The nat parameter of forallH is the index of the variable being quantified. The Terms parameter occurring in atomic would be replaced with (Vector Term1) if the definition of Term1 were used. The H at the end of the logic connectives, such as impH, stands for "Hilbert" and is used to distinguish them from Coq's connectives.

For example, the formula $\neg\forall \boldsymbol{x}_0. \forall \boldsymbol{x}_1. \boldsymbol{x}_0 = \boldsymbol{x}_1$ is represented by:

```
notH (forallH 0 (forallH 1 (equal (var 0) (var 1))))
```

There are two common alternatives to handle variable binding. One alternative is to use higher order abstract syntax to handle bound variables by giving `forallH` the type `(Term -> Formula) -> Formula`. I would represent the above example as:

```
notH (forallH (fun x : Term =>
        (forallH (fun y : Term => (equal x y)))))
```

This technique would require additional work to disallow "exotic terms" that are created by passing a function into `forallH` that does a case analysis on the term and returning entirely different formulas in different cases. Despeyroux and Hirschowitz [Despeyroux and Hirschowitz, 1994] address this problem by creating a complicated predicate that only valid formulas satisfy. Recently, Chlipala [Chlipala, 2008] has created an elegant solution for using higher order abstract syntax.

Another alternative is to use de Bruijn indices to eliminate named variables. However dealing with free and bound variables with de Bruijn indices can be difficult [McBride and McKinna, 2004a].

Using named variables allowed me to closely follow the standard way of writing formulas in mathematics. Using named variables is also helpful to persuade people of the correctness of the statement of the incompleteness theorem, which refers to the notion of provability and hence formulas.

Renaming bound variables turned out to be a constant source of work during development, because variable names and terms were almost always abstract. In principle the variable names could conflict, so it was constantly necessary to consider this case and deal with it by renaming a bound variable to a fresh one. Perhaps it would have been better to use de Bruijn indices and a deduction system that only deduced closed formulas, or perhaps one of the alternative implementations of variables that I discuss in Section 7.7.

4.3 Definition of `substituteFormula`

I defined the function `substituteFormula` to substitute a term for all occurrences of a free variable inside a given formula. While the definition of `substituteTerm` is simple structural recursion, substitution for formulas is complicated by quantifiers. Suppose we want to substitute the term s for $\boldsymbol{x}_i$ in the formula $\forall \boldsymbol{x}_j.\ \varphi$ and $i \neq j$. Suppose $\boldsymbol{x}_j$ is a free variable of s. If we naïvely perform the substitution, then the occurrences of $\boldsymbol{x}_j$ in s get captured by the quantifier. One common solution to this problem is to disallow substitution for a term s when s is not *substitutable* for x_i in φ. The solution I took was to rename the bound variable in this case:

$$(\forall \boldsymbol{x}_j.\ \varphi)[\boldsymbol{x}_i/s] := \forall \boldsymbol{x}_k.\ (\varphi[\boldsymbol{x}_j/\boldsymbol{x}_k])[\boldsymbol{x}_i/s] \text{ where } k \neq i \text{ and } \boldsymbol{x}_k \text{ is not free in } \varphi \text{ or } s$$

Unfortunately this definition is not structurally recursive. The second substitution operates on the result of the first substitution, which is not structurally smaller than the original formula. Coq will not accept this recursive definition as is; it is necessary to prove the recursion will terminate. I proved that substitution preserves the *depth* of a formula, and that each recursive call operates on a formula of smaller depth.

One of McBride's mantras says, "If my recursion is not structural, I am using the wrong structure" [McBride, 1999, p. 241]. In this case, my recursion is not structural because I am using the wrong recursion. Stoughton shows that it is easier to define substitution that substitutes all variables simultaneously because the recursion is structural [Stoughton, 1988]. If I had used this definition, I could have defined substitution of one variable in terms of it and many of my difficulties would have disappeared.

4.4 Definition of `Prf`

I defined the inductive type `(Prf Gamma phi)` to be the type of proofs of `phi`, from the list of assumptions `Gamma`. Originally `(Prf Gamma phi)` was a member of the `Prop` universe. It seemed natural to consider $\Gamma \vdash \varphi$ as a proposition, but it is important in this work to be able to distinguish between different proofs of the same statement. To get the ability to distinguish between different proofs requires that `(Prf Gamma phi)` be in `Set`.

```
Inductive Prf : Formulas -> Formula -> Set :=
  | AXM : forall A : Formula, Prf (A :: nil) A
  | MP : forall (Axm1 Axm2 : Formulas) (A B : Formula),
      Prf Axm1 (impH A B) -> Prf Axm2 A ->
         Prf (Axm1 ++ Axm2) B
  | GEN : forall (Axm : Formulas) (A : Formula) (v : nat),
      ~ In v (freeVarListFormula L Axm) -> Prf Axm A ->
         Prf Axm (forallH v A)
  | IMP1 : forall A B : Formula, Prf nil (impH A (impH B A))
  | IMP2 : forall A B C : Formula,
      Prf nil (impH (impH A (impH B C))
                  (impH (impH A B) (impH A C)))
  | CP : forall A B : Formula,
      Prf nil (impH (impH (notH A) (notH B)) (impH B A))
  | FA1 : forall (A : Formula) (v : nat) (t : Term),
      Prf nil (impH (forallH v A) (substituteFormula L A v t))
  | FA2 : forall (A : Formula) (v : nat),
      ~ In v (freeVarFormula L A) ->
         Prf nil (impH A (forallH v A))
  | FA3 : forall (A B : Formula) (v : nat),
      Prf nil
```

```
        (impH (forallH v (impH A B))
           (impH (forallH v A) (forallH v B)))
  | EQ1 : Prf nil (equal (var 0) (var 0))
  | EQ2 : Prf nil (impH (equal (var 0) (var 1))
                    (equal (var 1) (var 0)))
  | EQ3 : Prf nil
        (impH (equal (var 0) (var 1))
           (impH (equal (var 1) (var 2)) (equal (var 0) (var 2))))
  | EQ4 : forall R : Relations L, Prf nil (AxmEq4 R)
  | EQ5 : forall f : Functions L, Prf nil (AxmEq5 f).
```

`AxmEq4` and `AxmEq5` are recursive functions that generate the equality axioms for relations and functions. `AxmEq4 R` generates

$$\boldsymbol{x}_0 = \boldsymbol{x}_1 \Rightarrow ... \Rightarrow \boldsymbol{x}_{2n-2} = \boldsymbol{x}_{2n-1} \Rightarrow (R(\boldsymbol{x}_0, \boldsymbol{x}_2, ..., \boldsymbol{x}_{2n-2}) \Leftrightarrow R(\boldsymbol{x}_1, \boldsymbol{x}_3, ..., \boldsymbol{x}_{2n-1}))$$

and `AxmEq5 f` generates

$$\boldsymbol{x}_0 = \boldsymbol{x}_1 \Rightarrow ... \Rightarrow \boldsymbol{x}_{2n-2} = \boldsymbol{x}_{2n-1} \Rightarrow f(\boldsymbol{x}_0, \boldsymbol{x}_2, ..., \boldsymbol{x}_{2n-2}) = f(\boldsymbol{x}_1, \boldsymbol{x}_3, ..., \boldsymbol{x}_{2n-1})$$

I found that replacing ellipses from informal proofs with recursive functions was one of the most difficult tasks. This case was easy, but in general, the informal proof does not contain information on what inductive hypothesis should be used when reasoning about these recursive definitions. Figuring out the correct inductive hypotheses was not always easy.

4.5 Definition of `SysPrf`

There are some problems with the definition of `Prf` given. It requires the list of axioms to be in the correct order for the proof. For example, if we have `Prf Gamma1 (impH phi psi)` and `Prf Gamma2 phi` then we can conclude only `Prf Gamma1++Gamma2 psi`. We cannot conclude `Prf Gamma2++Gamma1 psi` or any other permutation of the axioms. If an axiom is used more than once, it must appear in the list more than once. If an axiom is never used, it must not appear. Also, the number of axioms must be finite because they form a list.

One possible solution would be to add permutation, weakening, and contraction rules for axioms to the inductive definition of `Prf`. However, this would further complicate the definition of `Prf`, which we would like to keep as simple as possible, and it would still not allow the number of axioms to be infinite. To solve this problem, I instead defined `System` to be `Ensemble Formula` and `SysPrf T phi` to be the proposition that the system `T` proves `phi`.

```
Definition System := Ensemble Formula.
Definition mem := Ensembles.In.
Definition SysPrf (T : System) (f : Formula) : Prop :=
  exists Axm : Formulas,
    (exists prf : Prf Axm f,
       (forall g : Formula, In g Axm -> mem _ T g)).
```

The type `Ensemble A` represents subsets of `A` by predicates of type `A -> Prop`. Recall that `a : A` is considered to be a member of `T : Ensemble A` if and only if the type `T a` is inhabited. I also defined `mem` to be `Ensembles.In` so that it does not conflict with `List.In`.

The separation between the notions of `Prf` and `SysPrf` is useful. Each member of `Prf` is a finite value that can easily be encoded by a natural number (see Section 5.1), while `SysPrf` allows one to work with infinite axiom systems.

4.6 The Deduction Theorem

The deduction theorem states that if $\Gamma \cup \{\varphi\} \vdash \psi$ then $\Gamma \vdash \varphi \Rightarrow \psi$. In Coq its formal statement is:

```
Theorem DeductionTheorem :
 forall (T : System) (f g : Formula) (prf : SysPrf (Add _ T g) f),
 SysPrf T (impH g f).
```

There is a choice of whether the side condition for the ∀-generalization rule, `~In v (freeVarListFormula L Axm)`, should be required or not. If this side condition is removed, then the deduction theorem requires a side condition on it. Usually all the formulas in an axiom system are closed, so the side condition on the ∀-generalization is easy to show. I therefore decided to keep the side condition on the ∀-generalization rule.

Originally my proof of the deduction theorem relied on an assumption that the language is decidable, meaning that the function and relation symbols form decidable types.

- `forall x y : Functions L, {x=y} + {x<>y}`
- `forall x y : Relations L, {x=y} + {x<>y}`

The classical proof of the deduction theorem at one point considers whether $\varphi \in F$ or $\varphi \notin F$ for a finite list of axioms F. This disjunction is provable under the assumption that the language is decidable. Because most reasonable languages are decidable, this extra assumption was not much of a burden; however, I regularly used the deduction theorem to prove general logical validities. Each time I did this I knew that there must be a way of proving the theorem without resorting to the deduction theorem, and that these theorems did not require the assumption that the language be decidable. There was good reason for this: the deduction theorem does not require that the language be decidable.

To understand how to remove the assumption that the language is decidable, one needs to first see how it is used in the classical proof. The classical proof works by translating a deduction $\Gamma \cup \{\varphi\} \supseteq F \vdash \psi$ into a new deduction $\Gamma \supseteq F' \vdash \varphi \Rightarrow \psi$, where $\Delta \supseteq G \vdash \theta$ means that θ is proved using the finite set of axioms G, which are a subset of the system Δ. The key place excluded middle is used is when $\Gamma \cup \{\varphi\} \supseteq F \vdash \forall x. \psi$ by the `GEN` rule (forall generalization). In this case, $x \notin \mathrm{FV}(F)$, and $F \vdash \psi$. The classical proof considers two possible cases. In one case, $\varphi \in F$, and therefore $x \notin \mathrm{FV}(\varphi)$. By induction we know that $\Gamma \supseteq F' \vdash \varphi \Rightarrow \psi$. By using the fact that $x \notin \mathrm{FV}(\varphi)$, plus some logical rules, one can conclude that $\Gamma \supseteq F' \vdash \varphi \Rightarrow \forall x. \psi$. In the other case $\varphi \notin F$, and thus $\Gamma \supseteq F \vdash \forall x. \psi$. Then it easily follows that $\Gamma \supseteq F \vdash \varphi \Rightarrow \forall x. \psi$.

Notice the fact that $\varphi \notin F$ is simply used to conclude that $\Gamma \supseteq F$. Therefore, it is sufficient to prove that $\varphi \in F \vee \Gamma \supseteq F$ instead of using excluded middle. This disjunction is easy to prove from the assumption that $\Gamma \cup \{\varphi\} \supseteq F$ by performing induction on F. The decidability of the language is not needed.

This illustrates one technique for removing uses of the excluded middle. One can follow the two cases of the disjunction until one gets to two formulas whose disjunction you can constructively prove. In some sense, my modified proof is simpler than before. The reasoning to get from $\varphi \notin F$ to $\Gamma \supseteq F$ has been removed. On the other hand, the proof is more complicated because an additional argument is needed to prove $\varphi \in F \vee \Gamma \supseteq F$.

Although this technique sounds easy, I must confess that I only noticed it in retrospect. I derived this new proof using my intuition that I ought to be able to use the constructive disjunction hidden inside the union operation to decide between these two cases and I kept fiddling until I got it.

4.7 Substitution Lemmas

After some development, it became clear that there were three key substitution lemmas to be proved:

1. $\Gamma \vdash \varphi[\boldsymbol{x}_i/s] \Leftrightarrow \varphi$ when $\boldsymbol{x}_i \notin \mathrm{FV}(\varphi)$
2. $\Gamma \vdash \varphi[\boldsymbol{x}_i/\boldsymbol{x}_j][\boldsymbol{x}_j/s] \Leftrightarrow \varphi[\boldsymbol{x}_i/s]$ when $\boldsymbol{x}_j \notin \mathrm{FV}(\varphi)\backslash\{\boldsymbol{x}_i\}$
3. $\Gamma \vdash \varphi[\boldsymbol{x}_i/s][\boldsymbol{x}_j/t] \Leftrightarrow \varphi[\boldsymbol{x}_j/t][\boldsymbol{x}_i/s]$ when $i \neq j$, $\boldsymbol{x}_i \notin \mathrm{FV}(t)$, and $\boldsymbol{x}_j \notin \mathrm{FV}(s)$

Ideally I would be able to prove the equality between the formulas; however, the formulas are not equal because applying the different substitution operations may cause bound variables to be renamed in different ways. The result is that the different sides of the equation are only α-equivalent, meaning equivalent up to renaming of bound variables. I never defined an equivalence relation for α-equivalence, so I instead proved the logical equivalence of the formulas. In Coq these lemmas are stated as follows.

```
Lemma subFormulaNil :
 forall (f : Formula) (T : System) (v : nat) (s : Term),
 ~ In v (freeVarFormula L f) ->
 SysPrf T (iffH (substituteFormula L f v s) f).

Lemma subFormulaTrans :
 forall (f : Formula) (T : System) (v1 v2 : nat) (s : Term),
 ~ In v2 (list_remove _ eq_nat_dec v1 (freeVarFormula L f)) ->
 SysPrf T
   (iffH (substituteFormula L
      (substituteFormula L f v1 (var v2)) v2 s)
      (substituteFormula L f v1 s)).

Lemma subFormulaExch :
 forall (f : Formula) (T : System) (v1 v2 : nat) (s1 s2 : Term),
 v1 <> v2 ->
 ~ In v2 (freeVarTerm L s1) ->
 ~ In v1 (freeVarTerm L s2) ->
 SysPrf T
   (iffH (substituteFormula L (substituteFormula L f v1 s1) v2 s2)
      (substituteFormula L (substituteFormula L f v2 s2) v1 s1)).
```

These proofs were the first major challenge that I faced. These theorems are not discussed in Hodel's textbook [Hodel, 1995], so it was necessary for me to develop the proofs myself. Naturally I proceeded by induction on the depth of the formula φ. The cases dealing with formulas of the form $\forall \boldsymbol{x}_j.\,\psi$ were the most difficult.

For example, consider the case of trying to prove the equivalence of $(\forall \boldsymbol{x}_j.\,\psi)[\boldsymbol{x}_i/s]$ and $\forall \boldsymbol{x}_j.\,\psi$ when $\boldsymbol{x}_i \notin \mathrm{FV}(\forall \boldsymbol{x}_j.\,\psi)$. If $i = j$ then the two formulas are identical, so that case is trivial. In the case that $i \neq j$ then there is a k different from i such that $\boldsymbol{x}_k \notin \mathrm{FV}(s)$ and such that $(\forall \boldsymbol{x}_j.\,\psi)[\boldsymbol{x}_i/s] = \forall \boldsymbol{x}_k.(\psi[\boldsymbol{x}_j/\boldsymbol{x}_k][\boldsymbol{x}_i/s])$ (it may be the case that $k = j$). To prove one direction of the implication, consider the following reasoning inside first-order classical logic. Suppose that $\forall \boldsymbol{x}_k.(\psi[\boldsymbol{x}_j/\boldsymbol{x}_k][\boldsymbol{x}_i/s])$ holds. Then one can derive $\psi[\boldsymbol{x}_j/\boldsymbol{x}_k][\boldsymbol{x}_i/s][\boldsymbol{x}_k/\boldsymbol{x}_j]$. By substitution lemma 3, one can derive $\psi[\boldsymbol{x}_j/\boldsymbol{x}_k][\boldsymbol{x}_k/\boldsymbol{x}_j][\boldsymbol{x}_i/s]$. By substitution lemma 1, one can derive $\psi[\boldsymbol{x}_j/\boldsymbol{x}_k][\boldsymbol{x}_k/\boldsymbol{x}_j]$. By substitution lemma 2, one can derive $\psi[\boldsymbol{x}_j/\boldsymbol{x}_j]$ which is equal to ψ. Then, using generalization, one can conclude $\forall \boldsymbol{x}_j.\,\psi$ as required.

This segment of the inductive proof of substitution lemma 1 relied on substitution lemmas 2 and 3. Eventually I discovered that my proofs of substitution lemmas 2 and 3 also relied on substitution lemmas 1, 2, and 3. This meant I had to prove all three substitution lemmas simultaneously by induction in one large lemma.

```
Remark subFormulaNTE :
 forall (f : Formula) (T : System),
 (forall (v : nat) (s : Term),
  ~ In v (freeVarFormula L f) ->
  SysPrf T (iffH (substituteFormula L f v s) f)) /\
 (forall (v1 v2 : nat) (s : Term),
  ~ In v2 (list_remove _ eq_nat_dec v1 (freeVarFormula L f)) ->
  SysPrf T
    (iffH (substituteFormula L
       (substituteFormula L f v1 (var v2)) v2 s)
       (substituteFormula L f v1 s))) /\
 (forall (v1 v2 : nat) (s1 s2 : Term),
  v1 <> v2 ->
  ~ In v2 (freeVarTerm L s1) ->
  ~ In v1 (freeVarTerm L s2) ->
  SysPrf T
    (iffH (substituteFormula L (substituteFormula L f v1 s1) v2 s2)
       (substituteFormula L (substituteFormula L f v2 s2) v1 s1))).
```

My proof of this lemma alone is about 1900 lines long. Admittedly, this was one of my early formal proofs, so it is not very well written. Nonetheless, even if I wrote it again today it would still be a substantial proof. These lemmas are not typically covered in logic courses or logic textbooks because typically the issue of bound variables and renaming is not addressed. However, it is exactly the bookkeeping involved in renaming bound variables and making sure that the side conditions about variables being free in expressions are satisfied that makes proving this lemma so difficult. When creating formal proofs, it is not possible to omit proving theorems just because they do not appear to provide much insight.

The difficulty of this proof strongly suggests that this is the wrong way to handle bound variables and substitution. These lemmas would probably be easier to prove, or may not even have been necessary if I had defined simultaneous substitution as the primitive substitution operation.

4.8 Languages and Theories of Number Theory

So far the theory of first order logic has been abstract over the language. To apply this theory I need some instances of languages. I created two languages. The first language, LNT, is the language of number theory and just has the function symbols Plus, Times, Succ, and Zero with appropriate arities. The second language, LNN, is the language of NN and has the same function symbols as LNT plus one relation symbol, LT, for less than.

I defined two axiom systems NN and PA (Peano arithmetic) for the languages LNN and LNT respectively. NN and PA share six axioms:

1. $\forall \boldsymbol{x}_0.\, \neg\, \boldsymbol{S}\boldsymbol{x}_0 = \boldsymbol{0}$
2. $\forall \boldsymbol{x}_0.\, \forall \boldsymbol{x}_1.\, \boldsymbol{S}\boldsymbol{x}_0 = \boldsymbol{S}\boldsymbol{x}_1 \Rightarrow \boldsymbol{x}_0 = \boldsymbol{x}_1$
3. $\forall \boldsymbol{x}_0.\, \boldsymbol{x}_0 + \boldsymbol{0} = \boldsymbol{x}_0$
4. $\forall \boldsymbol{x}_0.\, \forall \boldsymbol{x}_1.\, \boldsymbol{x}_0 + \boldsymbol{S}\boldsymbol{x}_1 = \boldsymbol{S}(\boldsymbol{x}_0 + \boldsymbol{x}_1)$
5. $\forall \boldsymbol{x}_0.\, \boldsymbol{x}_0 \times \boldsymbol{0} = \boldsymbol{0}$

6. $\forall \boldsymbol{x}_0. \forall \boldsymbol{x}_1. \boldsymbol{x}_0 \times \boldsymbol{S}\boldsymbol{x}_1 = (\boldsymbol{x}_0 \times \boldsymbol{x}_1) + \boldsymbol{x}_0$

`NN` has three additional axioms about less than:

1. $\forall \boldsymbol{x}_0. \neg\, \boldsymbol{x}_0 < \boldsymbol{0}$
2. $\forall \boldsymbol{x}_0. \forall \boldsymbol{x}_1. \boldsymbol{x}_0 < \boldsymbol{S}\boldsymbol{x}_1 \Rightarrow (\boldsymbol{x}_0 = \boldsymbol{x}_1 \vee \boldsymbol{x}_0 < \boldsymbol{x}_1)$
3. $\forall \boldsymbol{x}_0. \forall \boldsymbol{x}_1. \boldsymbol{x}_0 < \boldsymbol{x}_1 \vee \boldsymbol{x}_0 = \boldsymbol{x}_1 \vee \boldsymbol{x}_1 < \boldsymbol{x}_0$

`PA` has an infinite number of induction axioms that follow one schema.

- (schema) $\forall \boldsymbol{x}_{i_1}. \dots \forall \boldsymbol{x}_{i_n}. \varphi[\boldsymbol{x}_j/\boldsymbol{0}] \Rightarrow \forall \boldsymbol{x}_j. (\varphi \Rightarrow \varphi[\boldsymbol{x}_j/\boldsymbol{S}\boldsymbol{x}_j]) \Rightarrow \forall \boldsymbol{x}_j. \varphi$

The $\boldsymbol{x}_{i_1}$, ..., $\boldsymbol{x}_{i_n}$ are the free variables of $\forall \boldsymbol{x}_j. \varphi$. The quantifiers ensure that all the axioms of PA are closed.

Because NN is in a different language than PA, a proof in NN is not a proof in PA. In order to reuse the work done in NN, I created a function called `LNN2LNT_formula` to convert formulas in `LNN` into formulas in `LNT` by replacing occurrences of $t_0 < t_1$ with $(\exists \boldsymbol{x}_2. \boldsymbol{x}_0 + (\boldsymbol{S}\boldsymbol{x}_2) = \boldsymbol{x}_1)[\boldsymbol{x}_0/t_0, \boldsymbol{x}_1/t_1]$, where $\varphi[\boldsymbol{x}_0/t_0, \boldsymbol{x}_1/t_1]$ is the simultaneous substitution of t_0 for $\boldsymbol{x}_0$ and t_1 for $\boldsymbol{x}_1$. I then proved that if NN $\vdash \varphi$ then PA $\vdash$ `LNN2LNT_formula`(φ) by using the induction schema to prove the three axioms about $<$ in NN.

Chapter 5
Representing Functions

The key to proving the incompleteness theorem is to show that one can represent the provability predicate with a first-order formula. A common approach is to develop a primitive recursive function that can verify proofs and then prove that every primitive recursive function is representable by a first-order formula. Because primitive recursive functions can only operate on natural numbers, not formulas or proofs, it is necessary to select a way of encoding data structures as natural numbers.

5.1 Coding

Gödel's original approach was to code a formula as a list of numbers and then code that list using properties from the prime decomposition theorem [Gödel, 1931]. I avoided needing theorems about prime decomposition by instead following Hodel's technique of using the Cantor pairing function [Hodel, 1995]. The Cantor pairing function, `cPair`, is a commonly used bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

$$\texttt{cPair}(a, b) := a + \sum_{i=1}^{a+b} i$$

All my inductive structures were easy to recursively encode. I gave each constructor a unique number and paired that number with the encoding of all its parameters. For example, I defined `codeFormula` as follows, where `codeR` is a coding of the relation symbols for the language:

```
Fixpoint codeFormula (f : Formula) : nat :=
  match f with
  | fol.equal t1 t2 => cPair 0 (cPair (codeTerm t1) (codeTerm t2))
  | fol.impH f1 f2 =>
      cPair 1 (cPair (codeFormula f1) (codeFormula f2))
  | fol.notH f1 => cPair 2 (codeFormula f1)
  | fol.forallH n f1 => cPair 3 (cPair n (codeFormula f1))
  | fol.atomic R ts => cPair (4+(codeR R)) (codeTerms _ ts)
  end.
```

I created coding functions for all my data structures including lists, terms, formulas, and proofs. I proved each coding function was injective.

For each language, a function called `nattoTerm : nat -> Term` transforms a natural number to the closed term representing it. In this thesis I will refer to this function as $\ulcorner\cdot\urcorner$, so $\ulcorner 0 \urcorner = \mathbf{0}$, $\ulcorner 1 \urcorner = \boldsymbol{S}\mathbf{0}$, etc. I will also use $\ulcorner\varphi\urcorner$ for $\ulcorner$`codeFormula` $\varphi\urcorner$ and $\ulcorner t\urcorner$ for $\ulcorner$`codeTerm` $t\urcorner$.

5.2 Primitive Recursive Functions

In this section, I use definitions for extensional equality of functions on the natural numbers. In particular I defined the following:

- `naryFunc` is the type of n-ary functions.
- `naryRel` is the type of n-ary relations.
- `extEqual` is extensional equality of n-ary functions.
- `charFunction` returns the n-ary characteristic function of an n-ary relation.

I defined the type `PrimRec n` for primitive recursive expressions of arity `n` as:

```
Inductive PrimRec : nat -> Set :=
  | succFunc : PrimRec 1
  | zeroFunc : PrimRec 0
  | projFunc : forall n m : nat, m < n -> PrimRec n
  | composeFunc :
     forall (n m : nat) (g : PrimRecs n m) (h : PrimRec m),
        PrimRec n
  | primRecFunc :
     forall (n : nat) (g : PrimRec n) (h : PrimRec (S (S n))),
        PrimRec (S n)
with PrimRecs : nat -> nat -> Set :=
  | PRnil : forall n : nat, PrimRecs n 0
  | PRcons : forall n m : nat,
     PrimRec n -> PrimRecs n m -> PrimRecs n (S m).
```

The type `PrimRec` contains expressions of primitive recursive functions, but are not themselves functions. I defined `evalPrimRec : forall n : nat, PrimRec n -> naryFunc n` to convert expressions into a functions. Rather than working directly with primitive recursive expressions, I worked with particular Coq functions and proved they were extensionally equal to the interpretation of some primitive recursive expressions. This relation is captured by `isPR`.

```
Definition isPR (n : nat) (f : naryFunc n) : Set :=
  {p : PrimRec n | extEqual n (evalPrimRec _ p) f}.
```

Similarly, an n-ary relation is considered primitive recursive if its characteristic function is primitive recursive.

```
Definition isPRrel (n : nat) (R : naryRel n) : Set :=
  isPR n (charFunction n R).
```

5.3 Creating Primitive Recursive Functions

I proved that many functions are primitive recursive, including `plus`, `mult`, `pred`, `minus`, `max`, and `cPair`. I also proved that many relations have characteristic functions that are primitive recursive, such as `ltBool`, `leBool`, and `beq_nat` (equality on the natural numbers). More importantly, I proved several theorems about creating primitive recursive functions out of others. For example, if `f` is a 2-ary primitive recursive function, then `fun a b c : nat => f a b` is a 3-ary primitive recursive function. I also proved that course-of-values recursion is primitive recursive. Another very useful theorem states that a bounded search for the least number satisfying a primitive recursive predicate is primitive recursive. The bounded search is used, for instance, to compute the two projection functions of the Cantor pairing function. Given a code n for a pair of natural numbers, a function called `searchXY` finds the first $c \le n$ such that $n < \sum_{i=1}^{c+1} i$. The first projection, `cPairPi1`, returns $n - \sum_{i=1}^{c} i$. The second projection, `cPairPi2`, returns $c -$ `cPairPi1`(n).

I needed to show that many functions are primitive recursive, including operations on lists, computing free variables, checking if a formula is an instance of Peano arithmetic's induction schema, and more. Of course these functions do not operate on natural numbers, so they themselves cannot be primitive recursive. Instead, I proved that the corresponding operations on the codes of these values are primitive recursive.

It is easy to prove that certain Coq functions are extensionally equivalent to primitive recursive functions when one defines functions from a limited set of primitives. For example, the function `codeApp`, which given the code of two lists of natural numbers returns a code of the concatenation of the two lists, is defined as follows:

```
Definition codeApp : nat -> nat -> nat :=
  evalStrongRec 1
    (fun n Hrecs p1 : nat =>
     switchPR n
       (S (cPair (cPairPi1 (pred n))
             (codeNth (n - S (cPairPi2 (pred n))) Hrecs))) p1).
```

This unusual definition is built out of functions such as `evalStrongRec`, which does course-of-values recursion, and `switchPR`, which behaves like an if statement by returning one of the two last arguments depending on whether the first argument is zero or not. Because I proved that functions built from these kinds of pieces are primitive recursive, it was easy to prove that `codeApp` is primitive recursive.

5.3.1 `codeSubFormula` is Primitive Recursive

I proved that substitution is primitive recursive, meaning that I proved that the corresponding function operating on codes is primitive recursive. The function on codes is called `codeSubFormula`, and I proved it is correct in the following sense:

```
Lemma codeSubFormulaCorrect :
 forall (f : Formula) (v : nat) (s : Term),
  codeSubFormula (codeFormula f) v (codeTerm s) =
  codeFormula (substituteFormula L f v s).
```

The proof that `codeSubFormula` is primitive recursive is very difficult. The problem is again with the need to rebind bound variables. Normally one would attempt to create this primitive recursive function by using course-of-values recursion. Course-of-values recursion requires all recursive calls to have a smaller code than the original call. Renaming a bound variable requires two recursive calls. Recall the definition of substitution in this case:

$$(\forall \boldsymbol{x}_j.\, \varphi)[\boldsymbol{x}_i/s] := \forall \boldsymbol{x}_k.\, (\varphi[\boldsymbol{x}_j/\boldsymbol{x}_k])[\boldsymbol{x}_i/s] \text{ where } k \neq i \text{ and } \boldsymbol{x}_k \text{ is not free in } \varphi \text{ or } s$$

If one is lucky, one might be able to make the inner recursive call (the value of k could cause difficulties), but there is no reason to suspect that the input to the second recursive call, $\varphi[\boldsymbol{x}_j/\boldsymbol{x}_k]$, is going to have a smaller code than the original input, $\forall \boldsymbol{x}_j.\, \varphi$.

If I had used the alternative definition of substitution, where all variables are substituted simultaneously, there would still be problems. The input would include a list of variable and term pairs that represents the substitutions to be performed. In this case a new pair would be added to the list when making the recursive call, so the input to the recursive call could still have a larger code than the input to the original call. Perhaps if I had represented variables with de Bruijn indices, then it would have been easier to prove that substitution is primitive recursive.

Hodel's presentation avoids this difficult proof in two ways. Firstly, he gives only a cursory presentation of the algorithms needed, so no detailed proof that substitution is primitive recursive is given. Secondly, his definition of substitution requires that the term substituting be substitutable into the context of the variable being replaced. This substitutability restriction means that bound variables need never be renamed. This would greatly simplify the proof that substitution is primitive recursive, if it were given.

With my representation it seems that using course-of-values recursion is difficult or impossible. Instead, inspired by Shankar's work [Shankar, 1994], I introduced the notion of a trace of computation. Shankar used a linear list to represent his computation, but I used a tree structure. Think of the trace of computation as a finite tree where the nodes contain the input and output of each recursive call. The subtrees of a node are the traces of the computation of the recursive calls. This tree can be coded as one number. I proved that there is a primitive recursive function called `checkSubFormulaTrace` that verifies if a number represents a trace of the computation of substitution. This function inspects each node of the tree to verify that each child node has the correct inputs based on the inputs of the current node, and it verifies that the output of the current node is correct based on the outputs of the child nodes. Codes for trees are strictly larger than codes for subtrees, so there is no problem using course-of-values recursion on the trace of the computation.

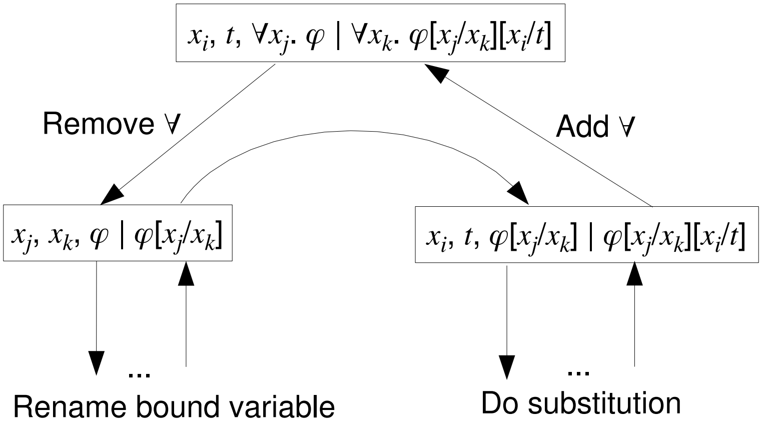

x_i, t, $\forall x_j.\ \varphi$ | $\forall x_k.\ \varphi[x_j/x_k][x_i/t]$
Remove ∀
Add ∀
x_j, x_k, φ | $\varphi[x_j/x_k]$
x_i, t, $\varphi[x_j/x_k]$ | $\varphi[x_j/x_k][x_i/t]$
...
Rename bound variable
...
Do substitution

Figure 5.1. Verifying the trace of the computation of substitution in the case of $\forall y.\varphi$.

Figure 5.1 illustrates the verification of a node in the case of computing $(\forall \boldsymbol{x}_j.\ \varphi)[\boldsymbol{x}_i/t]$ when $\boldsymbol{x}_j \in \mathrm{FV}(t)$, which is the most difficult case. The figure illustrates a node with two child nodes. The rest of the tree is omitted. The function `checkSubFormulaTrace` verifies that the input to the left child is correct by verifying that the variable $\boldsymbol{x}_j$ is the quantified variable from the root input formula, that the formula φ is the body of the root input formula, and that the new variable $\boldsymbol{x}_k$ is computed from the root input formula and the term t in the same way that `substituteFormula` computes it. The function `checkSubFormulaTrace` also verifies that the inputs to the right child are correct by verifying that the variable $\boldsymbol{x}_i$ and the term t are the same as the root input formula and that the formula is the same as the output formula from the left child. Finally, `checkSubFormulaTrace` verifies that the output formula of the root node is correct by verifying that it binds the variable $\boldsymbol{x}_k$ and that the body is the same as the output formula of the right child. Course-of-values recursion verifies the correctness of the left and right subtrees.

The key to proving that substitution $\varphi[\boldsymbol{x}_j/s]$ is primitive recursive is to create a primitive recursive function that computes a bound on how large the code of the trace of computation can be for a given input. I constructed a primitive recursive function for which it is relatively easy to prove that it bounds the size of the code of the trace. This function computes the largest variable index occurring in the input, and adds 3^d where d is the depth of the formula φ. This is an upper bound on the largest variable index that could be created during the computation of substitution. Let C be the max of the code of the variable with this index and the code for the term s. Let C' be the code of the formula φ with every variable (bound or not) replaced by the variable $\boldsymbol{x}_C$. Consider a full binary tree of depth d where each node contains three pseudo-input values of C, C, and C', and a pseudo-output value of C'. I proved that the code of this tree is larger than the code for the trace of the computation of substitution. I proved that computing the code of this large tree is primitive recursive. I created another primitive recursive function that searches for the trace of computation up to this bound. This is composed with another function that extracts the result of substitution from the trace of the computation. Thus `codeSubFormula` is primitive recursive. I then proved that `codeSubFormula` is correct by proving that all of the above computation works (not an easy task).

5.3.2 `checkPrf` is Primitive Recursive

Given a code for a formula and a code for a proof, the function `checkPrf` returns 0 if the proof does not prove the formula; otherwise, it returns one plus the code of the list of axioms used in the proof. I proved that this function is primitive recursive, as well as proving that it is correct in the following sense: for every proof p of $F \vdash \varphi$, $\texttt{checkPrf}(\ulcorner\varphi\urcorner, \ulcorner p\urcorner) = 1 + \ulcorner F\urcorner$; and for all $n, m : \mathbb{N}$ if $\texttt{checkPrf}(n, m) \neq 0$ then there exists φ, F, and some proof p of $F \vdash \varphi$ such that $\ulcorner\varphi\urcorner = n$ and $\ulcorner p\urcorner = m$.

The function `checkPrf` also ensures that all the formulas used in the proof are well formed, which means it checks to make sure that all of the function and relation symbols are part of the language. In principal this check is not needed, because any proof that uses extra symbols in intermediate steps can be transformed into a proof that does not use extra symbols; however it was easier to prove the correctness of `checkPrf` with this restriction.

5.3.3 Efficiency of Primitive Recursive Functions

As one may imagine, the above functions are quite slow at computing. In particular, `codeSubFormula` is even slower than I have suggested above. Instead of computing the depth of φ, I use the code of φ directly, because it serves as a suitable upper bound on the depth of φ. These extremely slow functions are fine to use because all that is important is that there exists some primitive recursive function that could do the computation. The function never actually needs to be executed. All that is needed is the fact that these functions can be represented by some first order formula.

5.4 Representing Primitive Recursive Functions

Informally, an n-ary function f is *representable* in NN if there exists a formula φ such that

1. the free variables of φ are among $\boldsymbol{x}_0, ..., \boldsymbol{x}_n$.
2. for all $a_1, ..., a_n : \mathbb{N}$, $\mathrm{NN} \vdash \varphi[\boldsymbol{x}_1/\ulcorner a_1\urcorner, ..., \boldsymbol{x}_n/\ulcorner a_n\urcorner] \Leftrightarrow \boldsymbol{x}_0 = \ulcorner f(a_1, ..., a_n)\urcorner$.

I proved that every primitive recursive function is representable in NN, or rather that the interpretation of every primitive recursive expression is representable. It is easy to show that the `projFuncs` and the basic functions `zeroFunc` and `succFunc` are representable. Much more difficult is showing that the composition and primitive recursion of representable functions are representable.

Given formulas φ_i that represent n-ary functions for $0 \leq i < m$, and given a formula ψ that represents an m-ary function, the composition can be represented by:

$$\exists \boldsymbol{x}_{n+1}.\dots\exists \boldsymbol{x}_{n+m}.\,\psi[\boldsymbol{x}_1/\boldsymbol{x}_{n+1},\dots,\boldsymbol{x}_m/\boldsymbol{x}_{n+m}] \wedge \varphi_0[\boldsymbol{x}_0/\boldsymbol{x}_{n+1}] \wedge \dots \wedge \varphi_{m-1}[\boldsymbol{x}_0/\boldsymbol{x}_{n+m}]$$

The difficulty in proving that the composition of m n-ary representable functions with one m-ary representable function is representable is proving that it works for all m and n. The proof presented by Hodel [Hodel, 1995] makes heavy use of ellipses and is more like a proof schema in terms of m and n. Of course, one cannot use ellipses in formal proofs. I felt I could prove the composition lemma for any particular m and n, but it was very difficult to create a proof that works for all m and n. The proof must proceed by induction on m and n; however, Hodel's proof [Hodel, 1995] offers no indication of which variable to do induction on first, nor what inductive hypothesis to use. The key insight was to allow more flexibility in the choice of indices for the bound variables. I prove for all $w \geq n$ that

$$\exists \boldsymbol{x}_{w+1}.\dots\exists \boldsymbol{x}_{w+m}.\,\psi[\boldsymbol{x}_1/\boldsymbol{x}_{w+1},\dots,\boldsymbol{x}_m/\boldsymbol{x}_{w+m}] \wedge \varphi_0[\boldsymbol{x}_0/\boldsymbol{x}_{w+1}] \wedge \dots \wedge \varphi_{m-1}[\boldsymbol{x}_0/\boldsymbol{x}_{w+m}]$$

represents the composition. I proceed by induction first on n and then on m. Even with this knowledge, the proof is difficult because n and m are both variables of the dependent type of `naryFunc`. Reasoning about dependent types is quite difficult, because Coq cannot automatically figure out how to generalize the dependent variables in the goal and context to make induction go through. It took some practice before I was able to figure out how to do the correct generalizations myself.

Arguably, Hodel's proof indicates this flexible choice of indices for the bound variables, because he uses the variables $y_1,\dots,y_k$, which are not in his formal syntax for logic. Instead he says, "The letters x, y, and z are used to denote individual variables." [Hodel, 1995, p. 135] This means his notation is ambiguous. When he uses x_n, is he referring to a formal variable for logic or is it a variable standing for a formal variable? It is exactly this sort of subtle ambiguity that makes translating paper proofs into formal proofs difficult. I do not believe Hodel is alone in making this ambiguity. I suspect most presentations of mathematical logic have such ambiguities.

Proving that the primitive recursion of representable functions is representable requires using Gödel's β-function along with the Chinese remainder theorem. Gödel's β-function is a function that codes array indexing. A finite list of numbers $a_0,\dots,a_n$ is coded as a pair of numbers (x, y) and $\beta(x, y, i) = a_i$. The β-function is special because it can be represented directly in terms of plus and times. The Chinese remainder theorem is used to prove that the β-function works.

Primitive recursion is represented by a formula that states the existence of a minimal witness to the computation of primitive recursion. A witness is a pair of numbers which represents a finite list of numbers representing the intermediate steps of the primitive recursion. The elements of this list are accessed using the representation of Gödel's β-function. It is important to use the minimum witness in order to prove that primitive recursion is representable. If the minimum is not used, it is not possible to prove that the formula represents a function that returns a unique value.

The representability of primitive recursion was one of the most difficult developments of my proof. The formula representing primitive recursion is moderately complicated and uses many bound variables and substitution. Once again, using named variables made the work difficult. Throughout my proof, I needed to show side conditions about variables not occurring as free variables. With all the quantifiers used in this formula, proving the side conditions became quite a bit of work. To aid my work I wrote a tactic called `PRsolveFV` to automatically generate proofs of these side conditions. The proofs generated by `PRsolveFV` became quite large. During development of the proof, when I was using Coq 7.3, generating these proofs took several hours. Furthermore, all proof objects were stored in memory. The Coq process took up many hundreds of megabytes of memory, and would cause the operating system to thrash if there was not sufficient physical memory available. During the transition to Coq 8.0, the Coq development team modified my proof, inserting commands to temporarily make `substituteFormula` opaque. This prevents the `substituteFormula` function from being computed and greatly speeds up the matching process used by my tactics by bailing out of pattern matches earlier. The result is that generating the proofs no longer takes hours. Coq also no longer stores all loaded proof objects in memory. Now, on my modern laptop, it takes less than 90 seconds and uses less than 80 megabytes of memory to check the file containing this proof, `PRrepresentable.v`.

Most of the side conditions about variables occur as a result of reasoning symbolically about substitutions in expressions with many quantifiers. Perhaps it would have been worthwhile to build a reflection tactic to compute the results of symbolic substitution. It is unclear how much work that would be compared to the time that would have been saved. It is certain that the resulting proof objects would be much smaller, and type-checking would be faster. Unfortunately I was not aware of this technique at the time.

I took care to make the formulas representing the primitive recursive functions clearly Σ_1 by ensuring that all unbounded quantifiers were existential. For instance, I used existential quantifiers in the representation of the composition of functions when universal quantifiers would have also worked. Making the representation of primitive recursion clearly Σ_1 was more difficult. The minimum witness predicate says that $P(x) \wedge \forall y < x. \neg P(y)$. The predicate $P(x)$ is used both in a negative and a positive context. To make the resulting formula Σ_1, the first instance, $P(x)$, needs to be Σ_1 and the second instance, $P(y)$, needs to be Π_1. Therefore, I made two instances of the predicate used for the minimum witness, `primRecSigmaFormulaHelp`, and `primRecPiFormulaHelp`. I have not proven that the formulas are Σ_1, because it is not needed for the first incompleteness theorem. Such a proof would be used for the second incompleteness theorem (see Section 7.6) and will be straightforward to prove given the work I have already done.

Chapter 6

The Incompleteness Theorem

There are two different statements of the incompleteness theorem. The original statement that Gödel proved in 1931 was that any axiom system of sufficient power that is ω-consistent is incomplete [Gödel, 1931]. In 1936, Rosser strengthened this theorem so that only consistency was required [Rosser, 1936]. I proved both these theorems in Coq. I will only discuss Rosser's version because it is the more general theorem.

6.1 Statement of the Incompleteness Theorem

The incompleteness theorem states that NN is essentially incomplete. This means that for every axiom system T such that

- $\mathrm{NN} \subseteq T$
- `T` can represent its own axioms
- `T` is a decidable set

there exists a sentence φ such that if $T \vdash \varphi$ or $T \vdash \neg\varphi$ then T is inconsistent.

My statement applies only to proofs in `LNN`, the language of NN. This statement does not show the incompleteness of theories that extend the language.

In Coq the theorem is stated as:

```
Theorem Incompleteness
     : forall T : System,
       Included Formula NN T ->
       RepresentsInSelf T ->
       DecidableSet Formula T ->
       exists f : Formula,
         Sentence f /\
         (SysPrf T f \/ SysPrf T (notH f) -> Inconsistent LNN T).
```

A `System` is `Inconsistent` if it proves all formulas.

```
Definition Inconsistent (T : System) :=
   forall f : Formula, SysPrf T f.
```

A `Sentence` is a `Formula` without any free variables.

```
Definition Sentence (f : Formula) :=
   forall v : nat, ~ In v (freeVarFormula LNN f).
```

A `DecidableSet` is an `Ensemble` such that every item either belongs to the `Ensemble` or does not belong to the `Ensemble`. This hypothesis is trivially true in classical logic, but in constructive logic I needed it to construct the inconsistency. Because this hypothesis is true classically, it does not claim that `T` is a computable set. Although a constructive proof of this hypothesis requires `T` to be computable, one cannot use this hypothesis to conclude that there exists a Turing machine that decides `T`.

```
Definition DecidableSet (A : Type)(s : Ensemble A) :=
   forall x : A, mem A s x \/ ~ mem A s x.
```

The `RepresentsInSelf` hypothesis restricts what a system T can be. The statement of essential incompleteness normally requires T to be a computable set. Instead I use the weaker hypothesis that the set T is expressible in the system T.

Given a system T extending NN and another system U along with a formula φ_U with at most one free variable $\boldsymbol{x}_i$, I say φ_U *expresses* the axiom system U in T if the following holds for all formulas ψ:

1. if $\psi \in U$ then $T \vdash \varphi_U[\boldsymbol{x}_i/\ulcorner\psi\urcorner]$
2. if $\psi \notin U$ then $T \vdash \neg\varphi_U[\boldsymbol{x}_i/\ulcorner\psi\urcorner]$

I say U is *expressible* in T if there exists a formula φ_U such that φ_U expresses the axiom system U in T.

In Coq I state that `T` is expressible in `T` as

```
Definition RepresentsInSelf (T : System) :=
exists rep : Formula, exists v : nat,
(forall x : nat, In x (freeVarFormula LNN rep) -> x = v)  /\
(forall f : Formula,
        mem Formula T f ->
        SysPrf T (substituteFormula LNN rep v
                    (natToTerm (codeFormula f)))) /\
(forall f : Formula,
        ~ mem Formula T f ->
        SysPrf T (notH (substituteFormula LNN rep v
                          (natToTerm (codeFormula f))))).
```

This is weaker than requiring that T be a computable set, because any computable set of axioms T is expressible in NN. Since T is an extension of NN, any computable set of axioms T is expressible in T.

By using this weaker hypothesis, I avoid defining what a computable set is. Also, in this form the theorem could be used to prove that any complete and consistent theory of arithmetic cannot define its own axioms. In particular, this could be used to prove Tarski's theorem that the truth predicate is not definable.

6.2 The Fixed Point Theorem

The fixed point theorem states that for every formula φ there is some formula ψ such that

$$\mathrm{NN} \vdash \psi \Leftrightarrow \varphi[\boldsymbol{x}_i / \ulcorner \psi \urcorner]$$

and that the free variables of ψ are the free variables φ with $\boldsymbol{x}_i$ removed.

```
Lemma FixPointLNN :
 forall (A : Formula) (v : nat),
 {B : Formula |
   SysPrf NN
     (iffH B (substituteFormula LNN A v
                (natToTermLNN (codeFormula B)))) /\
   (forall x : nat,
    In x (freeVarFormula LNN B) <->
    In x (list_remove _ eq_nat_dec v (freeVarFormula LNN A)))}.
```

The fixed point theorem allows one to create "self-referential sentences". I used this to create Rosser's sentence which states that for every code of a proof of itself, there is a smaller code of a proof of its negation. The proof of Rosser's incompleteness theorem requires doing a bounded search for a proof, and this requires knowing what is and what is not a proof in the system. For this reason, I require the decidability of the axiom system. Without a decision procedure for the axiom system, I cannot constructively do the search.

6.3 Provability Predicate

Let T and U be axiom systems such that there is some formula φ_U that expresses the axioms of U in T. This can be combined with `checkPrf` to create a formula called `codeSysPrf` such that `codeSysPrf`$[\boldsymbol{x}_0/\ulcorner n \urcorner, \boldsymbol{x}_1/\ulcorner m \urcorner]$ is provable in T if and only if m is the code of a proof in U of the formula coded by n. Another formula `codeSysPf` can be derived from `codeSysPrf` by adding an existential quantifier for $\boldsymbol{x}_1$ so that `codeSysPf`$[\boldsymbol{x}_0/\ulcorner n \urcorner]$ is provable in T if there exists a proof in U of a formula coded by n.

The formulas `codeSysPrf`, `codeSysPf`, etc. are not derived from primitive recursive functions so that the incompleteness theorem applies to axiom systems that may not have a primitive recursive characteristic function.

6.4 Essential Incompleteness of NN

All the pieces are in place for proving the essential incompleteness of NN. Suppose T is a system extending NN that satisfies the requirements of the incompleteness theorem. The axioms of T can be expressed in itself, so there exists a formula `codeSysPrf` for checking proofs of T in T. The fixed point theorem is applied to $\forall \boldsymbol{x}_1.\, \texttt{codeSysPrf} \Rightarrow \exists \boldsymbol{x}_2.\, \boldsymbol{x}_2 < \boldsymbol{x}_1 \wedge \texttt{codeSysPrfNot}[\boldsymbol{x}_1/\boldsymbol{x}_2]$ with the free variable $\boldsymbol{x}_0$ to create Rosser's sentence. The `codeSysPrfNot` formula is derived from the `codeSysPrf` formula such that $\texttt{codeSysPrfNot}[\boldsymbol{x}_0/\ulcorner n \urcorner, \boldsymbol{x}_1/\ulcorner m \urcorner]$ is provable in T if and only if m is the code of a proof in U of the negation of the formula coded by n. Finally, I can prove that if T proves the Rosser sentence or if T proves the negation of the Rosser sentence, then T is inconsistent.

Hodel [Hodel, 1995] adds an extra axiom to NN, producing a system called NN$^+$, in order to prove the Gödel-Rosser incompleteness theorem:

- $\forall \boldsymbol{x}_0.\, \forall \boldsymbol{x}_1.\, (\boldsymbol{x}_0 = \boldsymbol{x}_1 \vee \boldsymbol{x}_0 < \boldsymbol{x}_1) \Rightarrow \boldsymbol{x}_0 < \boldsymbol{S}\boldsymbol{x}_1$

Leo Harrington pointed out to me that this extra axiom is not necessary. Hodel uses this axiom in only one place, to prove $(\boldsymbol{x}_0 = \ulcorner n \urcorner \vee \boldsymbol{x}_0 < \ulcorner n \urcorner) \Rightarrow \boldsymbol{x}_0 < \boldsymbol{S}\ulcorner n \urcorner$ for all n. However, it is possible to prove this in NN already. Thus I proved the essential incompleteness theorem for NN instead of NN$^+$.

6.5 Essential Incompleteness of Peano Arithmetic

Peano arithmetic (PA) is not an extension of NN for two reasons. First, the induction scheme of PA replaces some of the axioms of NN. Requiring that T include all the axioms of NN for the incompleteness theorem was a small error on my part. Instead I should have required that T prove all the axioms of NN. Fortunately this slightly stronger theorem should be an easy corollary of my existing statement of the incompleteness theorem. The second reason is that Peano arithmetic uses the language of number theory, which is in a different language than that of NN; the incompleteness theorem for NN only applies to systems with the language `LNN`.

The language of number theory is a subset of the language of NN, so it is possible to inject formulas from the language of number theory into the language of NN. Let T' be the union of NN with the axioms of T injected into the language of NN. It is easy to prove that if T proves a formula φ, then T' proves φ', where φ' is the formula φ injected into the language of NN. I also proved that if T' proves a formula ψ, then T proves ψ', where ψ' is the formula ψ translated back into the language of number theory according to the translation given in Section 4.8. The `codeSysPrf` function and associated lemmas had only been proven valid for extensions of NN. I used this translation mechanism back and forth between T and T' to allow me to reuse the theorems about `codeSysPrf`. With this I was able to take the proof of the Gödel-Rosser theorem for NN and modify it to prove the essential incompleteness of Peano arithmetic.

The most obvious application of the essential incompleteness of Peano arithmetic is to apply it to PA itself. Peano arithmetic is a decidable theory that can represent its own axioms. It is trivially an extension of itself. I constructed a primitive recursive function to check if a number is the code of an instance of Peano arithmetic's induction schema (see Section 4.8). It was then easy to create a primitive recursive function to check to see if a number codes an axiom of PA. Because all primitive recursive functions are representable, I could create a predicate that represents the axioms of PA in NN, and, by translation, in PA also. Finally, by Rosser's incompleteness theorem, I proved that if PA is complete then it is inconsistent. However, it is possible to go further still.

6.6 Model Theory

Coq is sufficiently powerful to prove the consistency of Peano arithmetic. This allowed me to prove the incompleteness of Peano arithmetic—there (constructively) exists a sentence φ such that neither $\mathrm{PA} \vdash \varphi$ nor $\mathrm{PA} \vdash \neg\varphi$.

```
Theorem PAIncomplete :
 exists f : Formula,
   Sentence f /\ ~(SysPrf PA f \/ SysPrf PA (notH f)).
```

I proved the consistency of Peano arithmetic by proving that the natural numbers form a model. A model is a structure that contains a type, along with interpretations of the function and relations symbols.

```
Record Model (L:Language) : Type := model
  {U : Set;
   func : forall f : Functions L, naryFunc U (arity L (inr _ f));
   rel : forall r : Relations L, naryRel U (arity L (inl _ r))}.
```

Formulas can be interpreted with respect to a model plus a valuation function that assigns values to free variables occurring in the formula. A system has a model if all the interpretations of all the formulas of the system hold. I proved that any system that has a model is consistent.

One notable difficulty is that Coq's logic is constructive while the internal logic is classical. Therefore, it is difficult to interpret the proofs of the internal logic as proofs in Coq. Instead of directly interpreting the formulas, I use a negative translation of the formulas. In particular, I use Kuroda's negative translation [Troelstra and Schwichtenberg, 1996, p. 42]. This translation of a formula places two negations inside each universal quantifier, and two negations outside the entire formula. For example, I interpret $\boldsymbol{\forall x_0. \forall x_1. (S x_0 = S x_1 \Rightarrow x_0 = x_1)}$ as

$$\neg\neg\forall x{:}\, \mathbb{N}. \neg\neg\forall y{:}\, \mathbb{N}. \neg\neg(\mathsf{S}(x) = \mathsf{S}(y) \Rightarrow x = y).$$

It was then easy for me to prove that the interpreted formula will always hold if the original formula is provable from a system that has a model. From this I proved that a system cannot both have a model and be inconsistent.

```
Lemma ModelConsistent
     : forall (L : Language) (M : Model L) (T : System L)
         (value : nat -> U L M),
       (forall f : Formula L,
        mem (Formula L) T f ->
        interpFormula L M value (nnTranslate L f)) ->
       Consistent L T.
```

There is an alternative to the negative translation if the interpretation of all relation symbols (including equality) are stable. Recall that a relation is stable if $\neg\neg R(x_0, ..., x_{n-1}) \Rightarrow R(x_0, ..., x_{n-1})$. In this case one can directly interpret proofs and formulas inside Coq. The only tricky point is proving that the contrapositive axiom, CP, holds for all formulas. With the assumption that atomic formulas are stable, one can prove CP holds for atomic formulas. One can then prove that it holds for all formulas by induction on the syntax of formulas. This is possible because the primitive connectives of my formulas are $(\forall)$, $(\Rightarrow)$, and $(\neg)$. All these connectives have the same interpretation in constructive logic as they do in classical logic. Only the logical connectives $(\exists)$ and $(\vee)$ have different interpretations in constructive logic.

Equality over the natural numbers is provably stable in constructive mathematics, so this direct interpretation technique could have been used. However, I was unaware of this direct interpretation at the time I proved the ModelConsistent theorem. In the future, this direct interpretation could be implemented, allowing one to directly translate theorems from the inner logic into theorems in Coq. Currently equality is always interpreted as Leibniz equality. Because Coq has no quotient types, in the future it would be better to allow equality to be interpreted as any equivalence relation, and require the function and relation interpretations to be well defined with respect to the equivalence relation.

Chapter 7
Remarks

7.1 Trusting the Proof

If one accepts the correctness of the proof checker, Coq, then one only needs to verify that I correctly stated the incompleteness theorem. One needs to check the definitions of all the expressions used in the statement and verify that no axioms have been assumed. In fact, the proof depends on the `Ensembles` library which declares an axiom of extensionality for `Ensembles`, but one can verify that I never use this axiom. Fortunately, one does not need to check anything about coding or primitive recursive functions because these are not used in the statement of the incompleteness theorem.

7.2 Creating a Constructive Proof

Hodel did not constrain himself to giving a constructive presentation of the incompleteness theorem. However, creating a constructive proof from his presentation was not difficult. Gödel even noted in his original paper that his argument was intuitionistically proven [Gödel, 1931]. In my development, there were only two notable spots where using constructive logic was an issue.

One spot was the proof of the deduction theorem (see Section 4.6) where I originally used an assumption that language was decidable in order to prove $\varphi = \psi \vee \varphi \neq \psi$ for any two formulas φ and ψ. However, as I noted in that section, I was later able to modify this proof to remove even this innocent assumption.

The second spot was in my proof that Peano arithmetic is consistent (see Section 6.6). Technically, this is not even part of the incompleteness theorem. The difficulty was interpreting the classical deductions of Peano arithmetic in Coq's constructive metalogic. However, as I noted in that section, using a double negation interpretation works and allows one to conclude that Peano arithmetic is consistent from the fact that Coq's natural numbers form a model of the axioms.

Even though my meta-system, Coq, is constructive, I still used classical logic for the internal system. I did this because no one expects a constructive system to be complete, so proving the incompleteness theorem for constructive systems did not seem worthwhile.

7.3 Extracting the Sentence

Because my proof is constructive, it is possible, in principle, to compute the sentence that makes Peano arithmetic incomplete. This was not done for two reasons. The first reason is that the existential statement lives in Coq's `Prop` universe, and Coq only extracts from its `Set` universe. This was an error on my part. Fixing it to use Coq's `Set` existential quantifier should be fairly easy. The second reason is that the sentence contains a closed term of the code of most of itself. This code is a large number and it is written in unary notation. This would likely make the sentence far too large to be actually printed. Perhaps there is some way of convincing Coq to compute the Gödel sentence by writing the codes in standard decimal notation.

7.4 Robinson's System Q

The proof of essential incompleteness is usually carried out for Robinson's system Q. I instead followed Hodel's development [Hodel, 1995] and used NN. System Q is PA with the induction schema replaced with

$$\forall \boldsymbol{x}_0.\, \exists \boldsymbol{x}_1.\, (\boldsymbol{x}_0 = \mathbf{0} \vee \boldsymbol{x}_0 = \boldsymbol{S}\boldsymbol{x}_1).$$

All of NN axioms are Π_1, whereas Q has the above Π_2 axiom. Both axiom systems are finite.

Neither system is strictly weaker than the other, so it would not be possible to use the essential incompleteness of one to get the essential incompleteness of the other; however, both NN and Q are sufficiently powerful to prove a small number of needed lemmas, and afterward only these lemmas are used. If one abstracts my proof at these lemmas, it would then be easy to prove the essential incompleteness of both Q and NN.

7.5 Comparisons with Shankar's 1986 Proof

It is worth comparing this formalization of the incompleteness theorem with Shankar's 1986 proof in the Boyer-Moore theorem prover. The development of my proof is similar to Shankar's development. The time frame is even similar; Shankar spent eighteen months developing his proof, and I spent sixteen months (2002-03/2003-06-13) developing mine. It is unclear what this similar development time indicates. Perhaps the development has some intrinsic difficulty that proof systems have not removed in the last fifteen years. A fast computer does not help me much because the computer spends most of its time idle as I create my proof. On the other hand, perhaps there has been some improvement. I do not know how much experience Shankar had with the Boyer-Moore proof system, but this was my first major proof in a proof system. I only started using Coq a few months prior, and the most significant work I had previously done was to prove there are not a finite number of primes. I managed to do a more difficult version of the incompleteness theorem in more generality in a shorter period of time.

The most notable difference is between the proof systems. In Coq, the user is expected to input the proof, in the form of a proof script, and Coq will check the correctness of the proof. In the Boyer-Moore theorem prover, the user states a series of lemmas and the system generates the proofs. However, using the Boyer-Moore proof system requires feeding it a "well-chosen sequence of lemmas" [Shankar, 1994, p. xii]. Still, it seems the information being fed into the two systems is similar.

There are some notable semantic differences between Shankar's statement of incompleteness and mine. His theorem only states that finite extensions of Z2, hereditarily finite set theory, are incomplete, whereas my theorem states that even infinite extensions of NN are incomplete as long as they are self-representable. Also, Shankar's internal logic allows axioms to define new relation or function symbols as long as they come with the required proofs of admissibility. Such extensions are conservative over Z2, but no computer verified proof of this fact is given. My internal logic does not allow new symbols. Finally, I prove the essential incompleteness of NN. NN is a weaker system than Z2, and uses the language of arithmetic. Without any set structures, the proof is somewhat more difficult and requires using Gödel's β-function.

One of Shankar's goals when creating his proof was to use a proof system without modifications. Unfortunately he was not able to meet that goal; he ended up making some improvements to the Boyer-Moore theorem prover. My proof was developed in Coq without any modifications.

7.6 Gödel's Second Incompleteness Theorem

The second incompleteness theorem states that if T is a system extending PA (or a somewhat weaker system) such that T satisfies the requirements of the first incompleteness theorem and $T \vdash \mathsf{Con}_T$, then T is inconsistent. The formula Con_T is some reasonable sentence stating the consistency of T, such as $\neg\mathsf{Prov}_T(\ulcorner\neg(\forall \boldsymbol{x}_0.\, \boldsymbol{x}_0 = \boldsymbol{x}_0)\urcorner)$, where Prov_T is the provability predicate `codeSysPf` for T.

If I had created a formal proof in Peano arithmetic of Gödel's incompleteness theorem, I would have $\vdash_{\mathsf{PA}}$ "Gödel's first incompleteness theorem". This formal proof could then be mechanically transformed to create another formal proof in PA that $\vdash_{\mathsf{PA}}$ ($\mathsf{PA} \vdash$ "Gödel's first incompleteness theorem"). The second incompleteness theorem follows from this by a short argument. Unfortunately, I have only shown that $\vdash_{\mathsf{Coq}}$ "Gödel's first incompleteness theorem", so I cannot use the above argument to create a proof of the second incompleteness theorem.

Still, this work can be used as a basis for formalizing the second incompleteness theorem. The approach would be to prove the Hilbert-Bernays-Löb derivability conditions:

1. if $T \vdash \varphi$ then $T \vdash \mathsf{Prov}_T(\ulcorner\varphi\urcorner)$
2. $T \vdash \mathsf{Prov}_T(\ulcorner\varphi\urcorner) \Rightarrow \mathsf{Prov}_T(\ulcorner\mathsf{Prov}_T(\ulcorner\varphi\urcorner)\urcorner)$
3. $T \vdash \mathsf{Prov}_T(\ulcorner\varphi \Rightarrow \psi\urcorner) \Rightarrow \mathsf{Prov}_T(\ulcorner\varphi\urcorner) \Rightarrow \mathsf{Prov}_T(\ulcorner\psi\urcorner)$

The second incompleteness theorem follows easily from Gödel's first incompleteness theorem and these three conditions [Shoenfield, 1967]. I proved the second incompleteness theorem in Coq from the assumption that `codeSysPf` satisfies these three conditions. The first condition is basically the same as one of the correctness conditions for `codeSysPf` used in the first incompleteness theorem, so it was easy to prove. All that remains is to prove the last two conditions.

The second condition will be the most difficult to prove. It is usually proved by first proving that for every Σ_1 sentence φ, $\mathsf{PA} \vdash \varphi \Rightarrow \mathsf{Prov}_{\mathsf{PA}}(\ulcorner\varphi\urcorner)$. Because I made sure that all primitive recursive functions are representable by a Σ_1 formula, the second derivability condition would be a direct corollary.

7.7 Proof Development

I did a lot of cut and paste work when developing my proof. In Coq it is hard to reuse part of an existing proof because there are no tools for helping one to abstract part of a proof. For example, when doing two branches of a case analysis, one may find oneself doing the exact same proof (or similar proofs) in each case. However, it is difficult to abstract out this proof by hand because it means writing out the full statement of the lemma and generalizing the terms. It is so much easier to just cut and paste the proof script and fix the names of the hypothesis. This same problem is faced by programmers in many programming languages.

When developing a formal proof in Coq, I did a lot of moving back and forth over the proof script. Often one wants to back up to adjust the inductive hypothesis, or assert a lemma. Because some tactics automatically generate hypothesis names based on the context, the hypothesis may sometimes get new names when moving back and forth. This means that any reference one has to the old hypothesis names are no longer valid and need to be changed, which creates a lot of work. A strong discipline of avoiding automatically generated names by naming hypotheses oneself can alleviate this problem.

My formal proof follows a typical modern presentation of the incompleteness theorem. I used a Hilbert-style deduction system for classical logic. I used named variables for both free and bound variables in formulas. I defined substitution as substitution of one variable. I used primitive recursive functions to program `checkPrf`. The only unusual step I took in my proof was to follow Hodel's idea of using the Cantor pairing function to encode data structures as numbers, rather than using Gödel's encoding that requires the fundamental theorem of arithmetic. Looking back I feel that all these choices were bad choices (except the idea of using the Cantor pairing function). If I were to do this project again I would use a sequent calculus or perhaps a natural deduction system to get a constructive logic. I would use some other way of dealing with variable binding. I am inclined to try a novel approach such as using the nested datatype representation [Bird and Meertens, 1998][Hirschowitz and Maggesi, 2006] (with simultaneous substitution), or perhaps I would use a nominal approach to binding [Pitts, 2006]. I would consider using partial recursive functions instead of primitive recursive functions. These functions can be interpreted in Coq by functions that return a coinductive container of the result type [Capretta, 2005]. As a beginner at Coq, I could not have been expected to know all these alternatives before I started, and, in fact, many of these techniques have developed since I completed my proof.

I have shown that one can, with enough effort, create formal proofs that follow standard mathematical arguments. However, following a standard mathematical argument seems to be a poor way to create formal proofs. Instead it is probably better to innovate new approaches so that the proof becomes easier to formalize and/or the theorem becomes more general. For example, dealing with bound and free variables is typically considered a solved problem by logicians; however, it is currently an active field of research for formal mathematics and computer science [Aydemir et al., 2005]. I suspect that when mathematicians omit tedious proofs (such as results about variables and substitution) it is a sign that they are taking the wrong approach to a problem, or perhaps using the wrong structures.

7.8 Statistics

My proof [O'Connor, 2005c], excluding standard libraries and the library for Pocklington's criterion [Caprotti and Oostdijk, 2001], consists of 50 source files, 7 223 lines of specifications, 38 469 lines of proof, and 1 288 796 total characters. The size of the gzipped tarball (`gzip -9`) of all the source files is 149 698 bytes, which is an estimate of the information content of my proof.

Part II

Exact Real Arithmetic

Chapter 8
Introduction

Several mathematical theorems rely on numerical computation of real number values in their proofs. One example is the disproof of the Mertens conjecture [Odlyzko and teRiele, 1985]. The Mertens conjecture, which implies the Riemann hypothesis, claims that the partial sums of the Möbius function are bounded in absolute value by the square root function. Odlyzko and te Riele disproved the Mertens conjecture. Their disproof involves computing the first hundred decimal digits of the first two thousand zeros of the Riemann zeta function.

Another example is Hales's famous proof of the Kepler conjecture [Hales, 2002]. This proof involves verifying thousands of nonlinear inequalities over the real numbers. These inequalities were verified by special purpose software. Unfortunately, the peer review process has failed to fully accept the proof as correct. After four years, the twelve referees could only claim to be 99% certain of the correctness of the proof [Hales, 2006]. In response, Hales has started the *Flyspeck* project [Hales, 2006], a project to create a computer-verified proof of the Kepler conjecture. A full computer-verified proof will verify both the mathematical and computational parts of the proof together.

Many other proofs rely on real number computation. All of these proofs require a verified implementation of real number arithmetic before these proofs can be verified by a computer. By verified implementation, I mean a software verified library of continuous functions on $\mathbb{R}$ where computations can be performed to arbitrary precision. One such library has been created by Cruz-Filipe as a product of his software verified constructive proof of the fundamental theorem of calculus [Cruz-Filipe, 2003]. Because Cruz-Filipe's proof is constructive, the functions in his library can be evaluated. However, this construction was not designed with execution in mind, so unfortunately, evaluation is not practical [Cruz-Filipe and Spitters, 2003]. Therefore, if one wants to handle the problems encountered in the aforementioned mathematical proofs, a new library is needed.

This part develops a new constructive implementation of real numbers via a new characterization of metric spaces (Chapter 9). A generic method of completing metric spaces is developed and satisfies the properties of a monad (Chapter 10). This generic monadic interface is simple and elegant to use, and at the same time practical enough to run the computations needed for proofs. This work contributes not only to the *Flyspeck* project, but to any project requiring a verified library of functions for exact real arithmetic.

Three different complete metric spaces are developed:

- Chapter 11 develops the real numbers as the completion of the rational numbers. This chapter is based on my publication, “Certified Exact Transcendental Real Number Computation in Coq” [O’Connor, 2008a].
- Chapter 12 develops integrable functions as the completion of rational step functions. This chapter is based on a paper by Bas Spitters and myself entitled, “A computer verified, monadic, functional implementation of the integral” [O’Connor and Spitters, 2009].
- Chapter 13 develops compact sets as the completion of finite sets. This chapter is based on my publication, “A Computer Verified Theory of Compact Sets” [O’Connor, 2008b].

The purpose of this work is to formally verify the proofs presented in this part using a proof assistant. To this end, I have included many theorems with detailed proofs in order to help ease the formalization process. Those readers who are only interested in the data structures and operations may wish to skip over the details of the proofs. All of the theorems in this part have been verified by the Coq proof assistant except where otherwise noted.

8.1 Notation

The strictly positive rationals are denoted by $\mathbb{Q}^+$, while the non-negative rationals are denoted by $\mathbb{Q}^{0+}$. The strictly positive rationals with positive infinity are denoted by $\mathbb{Q}^+_\infty$. Similar notation is used for reals and natural numbers. The injection ι: $\mathbb{Q}^+ \Rightarrow \mathbb{Q}^+_\infty$ and some other similar injections will be omitted and their use will be implicit.

I use the notation $f^{(n)}(x)$ to express the operation f iterated n-times, while $f^n(x)$ means $(f(x))^n$. However, I will follow tradition and use f^{-1} to indicate the inverse function of f, despite the fact that $f^{(-1)}$ would actually be more appropriate.

I will sometimes use the notation $a^{\perp b}$ for $\mathsf{min}\ (a, b)$ and $a_{\top b}$ for $\mathsf{max}\ (a, b)$, particularly when the primary focus is on the a expression. I will use the notation $a^{\perp c}_{\top b}$ for $\mathsf{min}\ (\mathsf{max}\ (a, b), c)$. In order to avoid confusion, this “clamp” notation will only be used in contexts where $b \leq c$ since $\mathsf{min}\ \ (\mathsf{max}\ (a, b), c) = \mathsf{max}\ (\mathsf{min}\ (a, c), b)$ in this case.

The notation $\lceil a \rceil$ and $\lfloor a \rfloor$ is used to denote $\mathsf{ceiling}(a)$ and $\mathsf{floor}(a)$ respectively. The notation $\lfloor a \rceil$ is used to denote $\mathsf{round}(a)$ which rounds a to the nearest integer. I do not state which integer $\frac{1}{2}$ is rounded to; the proofs only rely on the fact that $\lfloor a \rceil$ is an integer such that $|a - \lfloor a \rceil| \leq \frac{1}{2}$.

Generally speaking, I will use different styles of variables for different types in accordance with the following table.

Variables	Types
a, b, c	rational numbers or points in an incomplete metric space
n, m	natural numbers
x, y, z	real numbers or points in a complete metric space
f, g, h	functions or function-like values
X, Y, Z	types and type-like values, especially for metric spaces
$\varepsilon, \delta, \gamma$	values of type $\mathbb{Q}^+$ or $\mathbb{Q}^+_\infty$
$\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$	predicates or other set-like values
$\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \ldots$	type constructors, especially for functors and monads

Table 8.1. A guide to variable name usage

However, I may violate this convention when necessary for clarity.

8.2 Rational Numbers

The rational numbers $\mathbb{Q}$ form the foundation for my metric space development. They play a particularly crucial role in my development of the real numbers (see Chapter 11). It is worth taking a detailed look at how they can be represented.

There are two obvious ways to represent $\mathbb{Q}$. The first way to define $\mathbb{Q}$ is as a setoid on $\mathbb{Z} \times \mathbb{N}^+$ where

$$(n_1, d_1) \asymp_{\mathbb{Q}} (n_2, d_2) := n_1\, d_2 =_{\mathbb{Z}} n_2\, d_1.$$

The pair (n, d) represents the rational number $\frac{n}{d}$. The second way to define $\mathbb{Q}$ is as a dependent pair $\exists p{:}\, \mathbb{Z} \times \mathbb{N}^+.\, \mathsf{coprime}(\pi_1(p), \pi_2(p))$. In this case, one can use regular equality because $\mathsf{coprime}$ is a classical predicate, meaning that $\mathsf{coprime}(x, y)$ has at most one proof (see Section 2.2.10). In either case, a binary representation for $\mathbb{Z}$ and $\mathbb{N}$ is used to allow for efficient arithmetic operations.

There are other ways of representing $\mathbb{Q}$. Most notably, $\mathbb{Q}$ can be defined using a continued fractions representation [Niqui, 2004]. Using this representation, there is no need to use a setoid relation nor is there a need to select canonical elements using a dependent pair. However, there are some efficiency issues with the continued fraction representation, so I did not choose to use it for my work.

The Coq standard library represents $\mathbb{Q}$ using the setoid representation. The problem with the dependent pair representation is that every operation must reduce the numerator and denominator to lowest terms before returning it. This reduction is effectively a GCD computation, and doing it after every operation can become expensive. However, never reducing the numerator and denominator can eventually lead to computations involving very large integers, which slows down evaluation. The standard library does not do any reduction after any of the standard operations. The library instead proves a function `Qred` that the user can call to reduce the pair to lowest terms. This leaves it to the user to decide when to reduce and when not to reduce.

I made very little use of `Qred` in my implementation. In Section 11.5.1, I define a compress function that reduces the size of numerators and denominators used during real number computation, thus avoiding running a GCD computation. However, I do not apply the compress function after my operations. Similar to the Coq standard library, I leave it up to the user to decide when to call compress to do the reduction.

I found that representing $\mathbb{Q}^+$ as a setoid on $\mathbb{N}^+ \times \mathbb{N}^+$ worked better than representing it as the dependent pair $\exists q\colon \mathbb{Q}. 0 < q$. I defined an implicit injection ι: $\mathbb{Q}^+ \Rightarrow \mathbb{Q}$, which makes it easy to use positive rationals in contexts requiring a rational number.

I will keep the representation of $\mathbb{Q}$ and $\mathbb{Q}^+$ abstract for the discussion within this thesis. I will use the equals symbol on rational numbers ($=_\mathbb{Q}$) with the understanding that one might use a setoid equality for one's representation instead.

I believe my work could be redeveloped using dyadic rational numbers (rational number whose denominator is a power of 2) instead of $\mathbb{Q}$. This might be more efficient at the cost of making the reciprocal operation (Section 11.3) and some other operations more complicated to define.

8.3 Regular Functions of Rationals

Abstractly, real numbers can be defined as any complete, Archimedean, ordered field; however, in order to show that such a structure exists, it is necessary to produce a model of the real numbers. One common model of the constructive real numbers consists of equivalence classes of Cauchy sequences of rational numbers with a constructive modulus of convergence [Geuvers and Niqui, 2002].

Bishop and Bridges [Bishop and Bridges, 1985] define the real numbers to be regular sequences of rational numbers with an equivalence relation.

Definition 8.1. A sequence x_n is *regular* if

$$\forall m n\colon \mathbb{N}^+. |x_m - x_n| \leq \frac{1}{m} + \frac{1}{n}.$$

Two regular sequences x and y are equivalent ($x \asymp y$) if

$$\forall n\colon \mathbb{N}^+. |x_n - y_n| \leq \frac{2}{n}.$$

Under this equivalence relation, regular sequences are isomorphic to Cauchy sequences. Every regular sequence is a Cauchy sequence, and every Cauchy sequence can be transformed into a regular sequence by using its modulus of convergence to select a suitable subsequence.

One can think of a regular sequence as a function that approximates a real number by taking a positive number n to a rational number x_n that is within $\frac{1}{n}$ of the real number the sequence represents. Instead of using such coarse-grained approximations, one can generalize the concept of regular sequences to regular functions.

Definition 8.2. A function $x\colon \mathbb{Q}^+ \Rightarrow \mathbb{Q}$ is *regular* if

$$\forall \varepsilon_1\, \varepsilon_2.\, |x(\varepsilon_1) - x(\varepsilon_2)| \leq \varepsilon_1 + \varepsilon_2,$$

and two regular functions x and y are equivalent if

$$\forall \varepsilon.\, |x(\varepsilon) - y(\varepsilon)| \leq 2\,\varepsilon.$$

Regular functions are isomorphic to regular sequences. Any regular function can be transformed into the regular sequence $\lambda n\colon \mathbb{N}^+ . x\left(\frac{1}{n}\right)$, and any regular sequence can be transformed into a regular function that maps ε to some x_n such that $\frac{1}{n} \leq \varepsilon$.

One can think of a regular function as a function that approximates a real number by taking a positive rational number ε to a rational number that is within ε of the real number the function represents. Regular functions allow more fine-grained approximations than regular sequences allow. This fine control can prevent unnecessary over-approximation when doing calculations.

Regular functions, like regular sequences, can be used not only to construct the reals, but to complete any metric space.

Chapter 9
Metric Spaces

A traditional definition of a metric space is a set X with a distance function $d\colon X \Rightarrow X \Rightarrow \mathbb{R}^{0+}_{\infty}$ satisfying certain properties [Burago et al., 2001]; however, the purpose of this part is to define $\mathbb{R}$ as the completion of a metric space. This task requires a definition of metric space that does not presuppose the existence of $\mathbb{R}$. Given a traditional metric space, one can define a distance relation $\boldsymbol{B}^X_\varepsilon(a,\ b) := d(a,\ b) \leq \varepsilon$ where $\varepsilon\colon \mathbb{Q}^+$. This distance relation does not depend on $\mathbb{R}$, but it still completely characterizes the metric space.

Definition 9.1. Given a tuple $\left(X, \asymp, \boldsymbol{B}^X\right)$, where $\asymp$ is an equivalence relation on X and $\boldsymbol{B}^X : \mathbb{Q}^+ \Rightarrow X \Rightarrow X \Rightarrow \star$ is a relation respecting the equivalence relation on X, meaning that

$$a_0 \asymp a_1 \Rightarrow b_0 \asymp b_1 \Rightarrow \left(\boldsymbol{B}^X_\varepsilon(a_0, b_0) \Leftrightarrow \boldsymbol{B}^X_\varepsilon(a_1, b_1)\right),$$

we call $\left(X, \asymp, \boldsymbol{B}^X\right)$ a *metric space* if the following properties hold:

1. $\forall a \varepsilon.\, \boldsymbol{B}^X_\varepsilon(a, a)$
2. $\forall a b \varepsilon.\, \boldsymbol{B}^X_\varepsilon(a, b) \Rightarrow \boldsymbol{B}^X_\varepsilon(b, a)$
3. $\forall a b c \varepsilon_1 \varepsilon_2.\, \boldsymbol{B}^X_{\varepsilon_1}(a, b) \Rightarrow \boldsymbol{B}^X_{\varepsilon_2}(b, c) \Rightarrow \boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}(a, c)$ (*triangle law*)
4. $\forall a b \varepsilon.\, (\forall \delta.\, \boldsymbol{B}^X_{\varepsilon+\delta}(a, b)) \Rightarrow \boldsymbol{B}^X_\varepsilon(a, b)$
5. $\forall a b.\, (\forall \varepsilon.\, \boldsymbol{B}^X_\varepsilon(a, b)) \Rightarrow a \asymp b$

I use the symbol $\boldsymbol{B}$ for the distance relation because if $\boldsymbol{B}^X_\varepsilon(a)$ is considered as a predicate, then the set of points satisfying $\boldsymbol{B}^X_\varepsilon(a)$ are exactly the points that are within ε of a. Therefore, $\boldsymbol{B}^X_\varepsilon(a)$ can be understood as a ball of radius ε around a.

Axioms 1 and 2 state that the distance relationship is reflexive and symmetric. Axiom 3 is a form of the triangle inequality. Axiom 4 states that $\boldsymbol{B}^X_\varepsilon(a)$ is a closed ball. I use closed balls because they are usually classical propositions, so they can be ignored during evaluation (see Section 2.3.2). For some metric spaces, such as the real numbers, open balls are defined with existential quantifiers, and their use would lead to unnecessary computation under strict evaluation [Cruz-Filipe and Spitters, 2003], which is the evaluation strategy used by Coq's `vm_compute` command (see Section 11.7). Axiom 5 states the identity of indiscernibles.

Given this definition of a metric space, the traditional metric can be recovered (classically) by defining $d(a, b) := \mathsf{inf}\left\{\varepsilon \mid \boldsymbol{B}^X_\varepsilon(a, b)\right\}$—the infimum of the empty set is taken to be ∞. One can easily show that my definition of a metric space is classically equivalent to the traditional definition.

My definition of a metric space is more general than the definition given by Bishop and Bridges [Bishop and Bridges, 1985]. Their definition requires that the distance between any two points can be computed to any given precision. Given my definition of a metric space, such a computation may not be possible. For example, if X is a metric space, then the space of functions $E \Rightarrow X$ can be given a metric where $\boldsymbol{B}^{E\Rightarrow X}_\varepsilon(f,\ g) := \forall a.\ \boldsymbol{B}^X_\varepsilon(f(a),\ g(a))$ (see Section 10.2). In general, the distance between two functions is uncomputable under this metric, and Bishop and Bridges would not consider this a metric space.

Notation 9.2. *Sometimes an extended distance relation $\breve{\boldsymbol{B}}^X : \mathbb{Q}^+_\infty \Rightarrow X \Rightarrow X \Rightarrow \star$ will be used where $\breve{\boldsymbol{B}}^X_\infty(a, b) := \top$ and for $\varepsilon{:}\, \mathbb{Q}^+$, $\breve{\boldsymbol{B}}^X_\varepsilon(a, b) := \boldsymbol{B}^X_\varepsilon(a, b)$.*

It is useful to note that if two points are within ε_1 of each other, then they are also within ε_2 of each other whenever $\varepsilon_1 < \varepsilon_2$.

Theorem 9.3. *For any ε_1 and ε_2 such that $\varepsilon_1 < \varepsilon_2$, if $\boldsymbol{B}^X_{\varepsilon_1}(a, b)$ holds then $\boldsymbol{B}^X_{\varepsilon_2}(a, b)$ holds.*

Proof. $\boldsymbol{B}^X_{\varepsilon_2-\varepsilon_1}(b,\ b)$ holds by reflexivity. Therefore, $\boldsymbol{B}^X_{\varepsilon_2}(a,\ b)$ holds by the triangle law. □

9.1 Uniform Continuity

Definition 9.4. Given two metric spaces $\left(X, \asymp, \boldsymbol{B}^X\right)$ and $\left(Y, \asymp, \boldsymbol{B}^Y\right)$, a function $f{:}\, X \Rightarrow Y$ is *uniformly continuous with modulus* $\mu{:}\, \mathbb{Q}^+ \Rightarrow \mathbb{Q}^+_\infty$ if

$$\forall a b \varepsilon.\, \breve{\boldsymbol{B}}^X_{\mu(\varepsilon)}(a, b) \Rightarrow \boldsymbol{B}^Y_\varepsilon(f(a), f(b)).$$

A function $f{:}\, X \Rightarrow Y$ is *uniformly continuous* if there is some μ such that f is uniformly continuous with modulus μ.

Notation 9.5. *When a function f is uniformly continuous, I will write $f{:}\ X \to Y$ using a single-bar arrow. This indicates that f is not just a function, but it is a dependent tuple*

$$\exists (f, \mu){:}\, (X \Rightarrow Y) \times \left(\mathbb{Q}^+ \Rightarrow \mathbb{Q}^+_\infty\right).\forall a b \varepsilon.\, \breve{\boldsymbol{B}}^X_{\mu(\varepsilon)}(a, b) \Rightarrow \boldsymbol{B}^Y_\varepsilon(f(a), f(b))$$

consisting of a function, its modulus of continuity, and a proof that it is uniformly continuous.

This structure is a morphism in the category of metric spaces with uniformly continuous functions between them. When $f\colon X \to Y$, I will leave the projections implicit and write $f(x)$ instead of $\pi_1(\pi_1(f))(x)$. (The coercion mechanism in Coq [The Coq Development Team, 2004] allows the projection function to be inferred in the same way in the formal proofs.)

This definition of uniform continuity is the same as the usual definition except that the relation between ε and δ is instead explicitly given by μ_f.

Notation 9.6. *Sometimes an extended function $\breve{\mu}\colon \mathbb{Q}^+_\infty \Rightarrow \mathbb{Q}^+_\infty$ will be used where $\breve{\mu}(\infty) := \infty$ and for $\varepsilon\colon \mathbb{Q}^+$, $\breve{\mu}(\varepsilon) := \mu(\varepsilon)$.*

Theorem 9.7. *The identity function* $\mathbf{I}$ *is uniformly continuous with modulus* $\mathbf{I}$*, and if $f\colon X \to Y$ and $g\colon Y \to Z$ are both uniformly continuous, then the composition $g \circ f\colon X \to Z$ is uniformly continuous with modulus $\breve{\mu_f} \circ \mu_g$.*

Proof. Easy. □

9.2 Prelength Spaces

In a metric space, just because two points a and b are within l of each other does not mean that there exist curves of length approaching l that connect these points. A metric space that has such curves is known as a length space [Burago et al., 2001]. Since the notion of curves presupposes the real numbers, I introduce the weaker notion of a prelength space.

Definition 9.8. A *prelength space* is a metric space such that

$$\forall a b \varepsilon \delta_1 \delta_2 . \varepsilon < \delta_1 + \delta_2 \Rightarrow \boldsymbol{B}^X_\varepsilon(a,b) \Rightarrow \exists c.\, \boldsymbol{B}^X_{\delta_1}(a,c) \wedge \boldsymbol{B}^X_{\delta_2}(c,b).$$

This property implies that if two points a and b are within l of each other, then there is a path of points connecting a to b with consecutive points within δ of each other whose total length is less than $l + \varepsilon$.

Theorem 9.9. *In a prelength space, for any a, b, ε, $\delta_0, \ldots, \delta_{n-1}$ such that $\boldsymbol{B}^X_\varepsilon(a,b)$ holds and $\varepsilon < \delta_0 + \ldots + \delta_{n-1}$, there exist c_0, ..., c_n such that $c_0 = a$, $c_n = b$, and $\forall i < n.\, \boldsymbol{B}^X_{\delta_i}(c_i, c_{i+1})$ holds.*

Prelength spaces are quite common. For example, all normed vector spaces are prelength spaces. All the metric spaces considered in this thesis are prelength spaces. A prelength space is also known as a metric space with approximate midpoints.

Theorem 9.10. $\left(\mathbb{Q},=,\boldsymbol{B}^{\mathbb{Q}}\right)$ *is a prelength space where* $\boldsymbol{B}^{\mathbb{Q}}_{\varepsilon}(a,b) := |a-b| \leq \varepsilon$.

9.3 Classification of Metric Spaces

There is a hierarchy of classes that metrics can belong to. The strongest class of metrics is the decidable metrics.

Definition 9.11. A *decidable metric* is a distance relation $\boldsymbol{B}^X$ such that

$$\forall a b \varepsilon. \boldsymbol{B}^X_{\varepsilon}(a,b) \vee \neg \boldsymbol{B}^X_{\varepsilon}(a,b).$$

The constructive disjunction here means that a constructive proof contains an algorithm for computing whether two points are within ε of each other or not. The metric on $\mathbb{Q}$ has this property; however, this disjunction cannot be constructed for the metric on $\mathbb{R}$.

The next strongest class of metrics is what I call the located metrics.

Definition 9.12. A *located metric* is a distance relation $\boldsymbol{B}^X$ such that

$$\forall a b \varepsilon \delta. \delta < \varepsilon \Rightarrow \boldsymbol{B}^X_{\varepsilon}(a,b) \vee \neg \boldsymbol{B}^X_{\delta}(a,b).$$

This is similar to being decidable, but there is a little extra wiggle room. If a and b are between δ and ε far apart, then the algorithm has the option of returning either a proof of $\boldsymbol{B}^X_{\varepsilon}(a, b)$ or $\neg \boldsymbol{B}^X_{\delta}(a, b)$. This extra flexibility will allow us to prove that $\mathbb{R}$ is a located metric. For some metrics even this disjunction cannot be constructed. In general, the located metric property cannot be constructed for the standard sup-metric on functions between metric spaces (see Section 10.2).

Theorem 9.13. *Every decidable metric is also a located metric.*

Proof. This follows from Theorem 9.3. □

The weakest class of metrics in this hierarchy is the stable metrics.

Definition 9.14. A *stable metric* is a distance relation $\boldsymbol{B}^X$ such that

$$\forall a b \varepsilon. \neg\neg \boldsymbol{B}^X_{\varepsilon}(a,b) \Rightarrow \boldsymbol{B}^X_{\varepsilon}(a,b).$$

Theorem 9.15. *Every located metric is a stable metric.*

Proof. Suppose that $\boldsymbol{B}^X$ is a located metric. Assume that $\neg\neg\boldsymbol{B}^X_\varepsilon(a, b)$ holds. Let δ be arbitrary. Because $\boldsymbol{B}^X$ is a located metric, either $\boldsymbol{B}^X_{\varepsilon+\delta}(a, b)$ holds, or $\neg\boldsymbol{B}^X_\varepsilon(a, b)$ holds. However, it is impossible that $\neg\boldsymbol{B}^X_\varepsilon(a, b)$ holds by our assumption. So $\boldsymbol{B}^X_{\varepsilon+\delta}(a, b)$ holds. Since δ was arbitrary, $\boldsymbol{B}^X_\varepsilon(a, b)$ holds as required. □

Note that when X has a stable metric, the equivalence relation between points $(a \asymp b)$ is stable, meaning that $\forall a b. \neg\neg(a \asymp b) \Rightarrow a \asymp b$. This follows from the fact that $a \asymp b \Leftrightarrow \forall \varepsilon. \boldsymbol{B}^X_\varepsilon(a, b)$.

Although we will discuss the possibility of metrics that are not constructively stable in Section 13.5, it appears that metric spaces used in practice are stable. The only place I require that a metric be stable is in Chapter 13, so, for the sake of generality, I will not assume that my metric spaces are stable in other chapters. Despite this generality, I believe one should consider making stability an additional axiom of metric spaces.

9.4 Completion of a Metric Space

One can define regular functions over any metric space X by using the distance relation.

Definition 9.16. A function $x : \mathbb{Q}^+_\infty \Rightarrow X$ is *regular* if

$$\forall \varepsilon_1 \varepsilon_2 : \mathbb{Q}^+. \boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}(x(\varepsilon_1), x(\varepsilon_2)).$$

Two regular functions x and y are equivalent $(x \asymp y)$ if

$$\forall \varepsilon_1 \varepsilon_2 : \mathbb{Q}^+. \boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}(x(\varepsilon_1), y(\varepsilon_2)).$$

Notice how this definition of equivalence means that $x : \mathbb{Q}^+_\infty \Rightarrow X$ is a regular function if and only if $x \asymp x$.

Call the type of regular functions over X the *completion* of X or $\mathfrak{C}(X)$.

Definition 9.17. $\mathfrak{C}(X) := \exists x : \mathbb{Q}^+_\infty \Rightarrow X. \forall \varepsilon_1 \varepsilon_2 : \mathbb{Q}^+. \boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}(x(\varepsilon_1), x(\varepsilon_2)).$

Define the distance relation on $\mathfrak{C}(X)$ as follows.

Definition 9.18. $\boldsymbol{B}^{\mathfrak{C}(X)}_\varepsilon(x, y) := \forall \delta_1 \delta_2 : \mathbb{Q}^+. \boldsymbol{B}^X_{\delta_1+\varepsilon+\delta_2}(x(\delta_1), y(\delta_2)).$

Theorem 9.19. *The distance relation $\boldsymbol{B}^{\mathfrak{C}(X)}$ respects the equivalence relation on $\mathfrak{C}(X)$.*

Proof. Given x_0, x_1, y_0, y_1: $\mathfrak{C}(X)$, suppose that $x_0 \asymp x_1$, $y_0 \asymp y_1$, and $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x_0, y_0)$ hold. Let δ_1, δ_2, γ: $\mathbb{Q}^+$ be arbitrary. We have that $\boldsymbol{B}^{X}_{\delta_1+\frac{\gamma}{4}}\big(x_1(\delta_1),\ x_0\big(\frac{\gamma}{4}\big)\big)$ and $\boldsymbol{B}^{X}_{\frac{\gamma}{4}+\delta_2}\big(y_0\big(\frac{\gamma}{4}\big),\ y_1(\delta_2)\big)$ hold. We also have that $\boldsymbol{B}^{X}_{\frac{\gamma}{2}+\varepsilon}\big(x_0\big(\frac{\gamma}{4}\big),\ y_0\big(\frac{\gamma}{4}\big)\big)$ holds. Therefore, $\boldsymbol{B}^{X}_{\gamma+\delta_1+\varepsilon+\delta_2}(x_1(\delta_1),\ y_1(\delta_2))$ holds. Because γ was arbitrary, we get that $\boldsymbol{B}^{X}_{\delta_1+\varepsilon+\delta_2}(x_1(\delta_1), y_1(\delta_2))$ holds as required to show that $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x_1, y_1)$ holds.

By the symmetry of $\asymp$, we also get that $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x_1,\ y_1) \Rightarrow \boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x_0,\ y_0)$ as required. □

The following theorem shows that the completion of a metric space is a metric space.

Theorem 9.20. *If $\big(X, \asymp, \boldsymbol{B}^X\big)$ is a metric space, then $\Big(\mathfrak{C}(X), \asymp, \boldsymbol{B}^{\mathfrak{C}(X)}\Big)$ is a metric space.*

Proof.

1. Consider arbitrary δ_1 and δ_2. By Theorem 9.3, $\boldsymbol{B}^{X}_{\delta_1+\delta_2+\varepsilon}(x(\delta_1),\ x(\delta_2))$ holds. Therefore, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}$ is reflexive.
2. $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}$ is symmetric because $\boldsymbol{B}^{X}_{\varepsilon}$ is symmetric.
3. Suppose that $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon_1}(x,\ y)$ and $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon_2}(y,\ z)$ hold. Let γ be arbitrary. Then $\boldsymbol{B}^{X}_{\delta_1+\varepsilon_1+\frac{\gamma}{2}}\big(x(\delta_1),\ \ y\big(\frac{\gamma}{2}\big)\big)$ and $\boldsymbol{B}^{X}_{\frac{\gamma}{2}+\varepsilon_2+\delta_2}\big(y\big(\frac{\gamma}{2}\big),\ \ z(\delta_2)\big)$ hold. Therefore, $\boldsymbol{B}^{X}_{\delta_1+\varepsilon_1+\varepsilon_2+\delta_2+\gamma}(x(\delta_1),\ y(\delta_2))$ holds. Since γ was arbitrary, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon_1+\varepsilon_2}(x,\ y)$ holds.
4. Given ε, x, and y, suppose that $\forall\gamma.\ \boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon+\gamma}(x,\ y)$ holds. Therefore, $\forall\gamma\delta_1\ \delta_2$, $\boldsymbol{B}^{X}_{\delta_1+\varepsilon+\delta_2+\gamma}(x(\delta_1),\ y(\delta_2))$ holds. Then $\forall\delta_1\ \delta_2.\ \boldsymbol{B}^{X}_{\delta_1+\varepsilon+\delta_2}(x(\delta_1),\ y(\delta_2))$ holds, which is the same as $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x, y)$.
5. Given x and y, suppose that $\forall\varepsilon.\ \boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x,\ y)$ holds. For any ε, δ_1, and δ_2, $\boldsymbol{B}^{X}_{\delta_1+\varepsilon+\delta_2}(x(\delta_1),\ y(\delta_2))$ holds. Then $\boldsymbol{B}^{X}_{\delta_1+\delta_2}(x(\delta_1),\ y(\delta_2))$ holds because ε was arbitrary. Therefore, $x \asymp y$.

□

The following are useful theorems about the distance relation on regular functions.

Theorem 9.21. *If $\boldsymbol{B}^{X}_{\varepsilon}(x(\varepsilon_1), y(\varepsilon_2))$ holds, then $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon_1+\varepsilon+\varepsilon_2}(x, y)$ holds.*

Proof. Let δ_1 and δ_2 be arbitrary. $\boldsymbol{B}^{X}_{\delta_1+\varepsilon_1}(x(\delta_1),\ x(\varepsilon_1))$ and $\boldsymbol{B}^{X}_{\varepsilon_2+\delta_2}(y(\varepsilon_2),\ y(\delta_2))$ hold. Therefore, $\boldsymbol{B}^{X}_{\delta_1+\varepsilon_1+\varepsilon+\varepsilon_2+\delta_2}(x(\delta_1), y(\delta_2))$ holds. □

Corollary 9.22. *If $\boldsymbol{B}^X_{\varepsilon_0}(x(\varepsilon_1), b)$ holds, then $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon_0+\varepsilon_1}\left(x, \hat{b}\right)$ holds where $\hat{b}$ is the regular function $\hat{b} := \lambda\varepsilon.b$.*

Proof. By Theorem 9.21 $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon_0+\varepsilon_1+\varepsilon_2}\left(x,\ \hat{b}\right)$ holds for every ε_2. Therefore, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon_0+\varepsilon_1}\left(x, \hat{b}\right)$ holds. □

The following theorem shows that the distance relation, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x,\ y)$ is equivalent to $\forall\delta.\, \boldsymbol{B}^X_{\varepsilon+2\delta}(x(\delta), y(\delta))$.

Theorem 9.23. *For any ε, if $\forall\delta.\, \boldsymbol{B}^X_{\varepsilon+2\delta}(x(\delta), y(\delta))$ holds, then $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x, y)$ holds.*

Proof. For any δ, $\boldsymbol{B}^X_{\varepsilon+\frac{\delta}{2}}\left(x\left(\frac{\delta}{4}\right),\ y\left(\frac{\delta}{4}\right)\right)$ holds. By Theorem 9.21, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon+\delta}(x,\ y)$ holds. Since δ was arbitrary, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x, y)$ holds. □

The following theorem shows that the completion of a prelength space is a prelength space.

Theorem 9.24. *If $\left(X, \asymp, \boldsymbol{B}^X\right)$ is a prelength space, then $\left(\mathfrak{C}(X), \asymp, \boldsymbol{B}^{\mathfrak{C}(X)}\right)$ is a prelength space.*

Proof. By Theorem 9.20, all that remains is to show that the prelength space property is preserved. Suppose that $\varepsilon < \delta_1 + \delta_2$, and $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x,\ y)$ holds. Let $\gamma := \mathsf{min}\ \left(\frac{\delta_1}{2},\ \frac{\delta_2}{2},\ \frac{\delta_1+\delta_2-\varepsilon}{5}\right)$. $\boldsymbol{B}^X_{\varepsilon+2\gamma}(x(\gamma),\ y(\gamma))$ holds. There is some c such that $\boldsymbol{B}^X_{\delta_1-\gamma}(x(\gamma), c)$ and $\boldsymbol{B}^X_{\delta_2-\gamma}(c,\ y(\gamma))$ hold. So $\boldsymbol{B}^{\mathfrak{C}(X)}_{\delta_1}(x, \hat{c})$ and $\boldsymbol{B}^{\mathfrak{C}(X)}_{\delta_2}(\hat{c},\ y)$ holds by Corollary 9.22 where $\hat{c} := \lambda\varepsilon.c$. □

The completion operation preserves the located metric and stable metric properties.

Theorem 9.25. *If X is a located metric, then $\mathfrak{C}(X)$ is a located metric.*

Proof. Let $xy\colon \mathfrak{C}(X)$ be arbitrary, and let $\delta\varepsilon\colon \mathbb{Q}^+$ be such that $\delta < \varepsilon$. We need to show that $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(x,\ y) \vee \neg\boldsymbol{B}^{\mathfrak{C}(X)}_{\delta}(x,\ y)$. Let $\gamma := \frac{\varepsilon-\delta}{5}$. Because X is located, then either $\boldsymbol{B}^X_{\delta+3\gamma}(x(\gamma), y(\gamma))$ holds or $\neg\boldsymbol{B}^X_{\delta+2\gamma}(x(\gamma), y(\gamma))$ holds.

Suppose that $\boldsymbol{B}^X_{\delta+3\gamma}(x(\gamma),\ y(\gamma))$ holds. Then $\boldsymbol{B}^{\mathfrak{C}(X)}_{\delta+5\gamma}(x,\ y)$ holds by Theorem 9.21, which is the same as $\boldsymbol{B}^X_{\varepsilon}(x, y)$ as required.

Suppose that $\neg\boldsymbol{B}^X_{\delta+2\gamma}(x(\gamma),\ y(\gamma))$ holds. Then, by Theorem 9.21, $\neg\boldsymbol{B}^{\mathfrak{C}(X)}_{\delta}(x,\ y)$ holds as required. □

Theorem 9.26. *If X is a stable metric, then $\mathfrak{C}(X)$ is a stable metric.*

We cannot expect to prove that completion operation preserves decidable metrics. The most obvious example is that $\mathbb{Q}$ is a decidable metric, while it is known that one cannot construct the decidable metric property for $\mathfrak{C}(\mathbb{Q})$ (which is $\mathbb{R}$).

Typically the completion of a space X is denoted by $\bar{X}$. Because I wish to emphasize the monadic property of the completion, I use Fraktur $\mathfrak{C}$.

9.5 Remarks

9.5.1 Infinite Distance

Often the definition of a metric space requires the distance between two points always to be finite. Because I will want to consider the space of all uniformly continuous functions between two metric spaces as a metric space in Section 10.2, I use the more liberal definition.

9.5.2 Dedekind Cuts

Instead of using the distance relation $\boldsymbol{B}^X$ as defined above, another possibility would be to use a relation of type $X \Rightarrow X \Rightarrow \mathbb{Q}^+ \Rightarrow \star$. This is just a rearrangement of the parameters to $\boldsymbol{B}^X$, so it does not appear to make a difference; however, you can also view it as a function of two parameters that returns a unary predicate on $\mathbb{Q}^+$. Axiom 4 and Theorem 9.3 imply that this predicate forms a closed uppercut of $\mathbb{Q}^+$, thus the result of the function is a Dedekind cut. Define $\mathbb{D}_\infty^{0+}$ to be the non-negative Dedekind extended reals, or more specifically, the dependent record type of closed upper cuts of $\mathbb{Q}^+$, including the empty cut (meaning ∞) and the full cut (meaning 0). One could then define a metric space by a metric function $\mathbf{d}^X$ of type $X \Rightarrow X \Rightarrow \mathbb{D}_\infty^{0+}$ (and adjust the axioms of a metric space appropriately). Only a small amount of theory about the Dedekind reals is needed, addition but not multiplication. Cauchy reals ($\mathbb{R}$) would then be defined as a metric space whose distance function returns a non-negative Dedekind real.

I attempted this approach. I tried to define equality and addition on $\mathbb{D}_\infty^{0+}$ to show that $\left(\mathbb{D}_\infty^{0+}, \asymp_\mathbb{D}, +_\mathbb{D}\right)$ is a monoid. This would have allowed me to reuse existing theorems about monoids that have been developed in the C-CoRN library. Unfortunately this does not work in Coq 8.0 because the fact that $\mathbb{D}_\infty^{0+}$ is a predicate forces $\mathbb{D}_\infty^{0+}$ into a higher universe level than is allowed by the C-CoRN library [CoRN, 2008]. Coq 8.1 allows more flexibility in universe levels for instances of inductive types. This could help alleviate some of the universe issues; however, I still fear that trying to use C-CoRN monoids for predicates will never work. To avoid universe level issues, I use the distance relation $\boldsymbol{B}^X$ as given in Definition 9.1, and I do not use Dedekind reals at all.

9.5.3 Identity of Indiscernibles

The distance relation can be used to define equality because

$$\left(\forall\varepsilon.\, \boldsymbol{B}^X_\varepsilon(a,b)\right) \Leftrightarrow a \asymp b$$

holds. If the type X does not come with an equivalence relation, then an equivalence relation can be defined from the distance relation. This would make Axiom 5 more of a definition than an axiom.

Usually a type X comes equipped with its own equivalence relation used in other theorems. This is the case for $\mathbb{Q}$. Therefore I use the existing equivalence relation in order to be compatible with those theorems. Otherwise I would end up constantly swapping between two equivalent equivalence relations.

9.5.4 Modulus of Continuity

My definition of modulus of continuity maps ε to δ, while the usual mathematical definition of modulus of continuity maps δ to ε. The inverse of the usual definition bounds my definition of the modulus of continuity. My definition of the modulus of continuity is the standard one for constructive analysis, and is essentially the same as Bishop and Bridges's definition [Bishop and Bridges, 1985].

9.5.5 Prelength Space

A prelength space is a weaker notion than a length space. For example, the rationals are not a length space because no two points can be connected by a continuous curve; however, the rationals are a prelength space. I chose the name *prelength space* because of an analogy with precompact spaces. When a precompact space is completed it becomes a compact space; when a prelength space is completed it becomes a length space.

A possible alternative definition for prelength space would be to use the classical existential quantifier:

$$\forall a\, b\, \varepsilon\, \delta_1\, \delta_2. \varepsilon < \delta_1 + \delta_2 \Rightarrow \boldsymbol{B}^X_\varepsilon(a,b) \Rightarrow \tilde{\exists} c.\, \boldsymbol{B}^X_{\delta_1}(a,c) \wedge \boldsymbol{B}^X_{\delta_2}(c,b)$$

The witness c is never used in any construction in my work. However, I will stick with the constructive existential for this part. If one uses the classical quantifier, one may need to add the extra assumption that the metric space has a stable metric to some of the theorems.

In the Coq development, I used the `Prop` based existential quantifier. This special quantifier has the same constructive introduction rules that the constructive existential quantifier has, but the elimination rule is restricted to be more like a classical quantifier. The dependent pair is never allowed to be eliminated when constructing a value of a `Set` kind (see Section 2.3.2). It can only be eliminated when constructing other `Prop`-kinded values (such as proofs of $\boldsymbol{B}^X_\varepsilon(a,b)$).

9.5.6 Notation for Projections of Dependent Records

Recall that metric spaces (Definition 9.1) and uniformly continuous functions (Definition 9.4) are represented by dependent record types. The distance relation $\boldsymbol{B}$ is actually a projection function that takes a metric space X and returns the distance relation $\boldsymbol{B}^X$. I won't distinguish between the type $\mathbb{Q}$ and the metric space $\left(\mathbb{Q}, =, \boldsymbol{B}^{\mathbb{Q}}\right)$; I will denote both as $\mathbb{Q}$. In fact, because I declared the dependent record `Q_as_MetricSpace` as a `Canonical Structure` [The Coq Development Team, 2004], Coq is also able to infer the correct metric space from the type `Q`. In particular, I will write the extraction of the distance relationship from a metric space as $\boldsymbol{B}^{\left(\mathbb{Q}, =, \boldsymbol{B}^{\mathbb{Q}}\right)}$ as simply $\boldsymbol{B}^{\mathbb{Q}}$. This means that the notation $\boldsymbol{B}^{\mathbb{Q}}$ may either denote the actual distance relationship on $\mathbb{Q}$ as defined in Theorem 9.10, or it may denote the projection function for the distance relationship applied to the metric space $\left(\mathbb{Q}, =, \boldsymbol{B}^{\mathbb{Q}}\right)$. This ambiguity in my notation poses no problem because $\boldsymbol{B}^{\left(\mathbb{Q}, =, \boldsymbol{B}^{\mathbb{Q}}\right)}$ reduces to $\boldsymbol{B}^{\mathbb{Q}}$.

Similar notation will be used for the modulus of continuity, μ_f, and other dependent records.

Chapter 10
Completion Is a Monad

The definition of a monad comes from category theory, but one does not need to understand the categorical definition of monads in order to use them. Given a type operation $\mathfrak{M}$, a type of morphisms, and three functions, $\mathsf{unit} : X \to \mathfrak{M}(X)$, $\mathsf{map} : (X \to Y) \Rightarrow (\mathfrak{M}(X) \to \mathfrak{M}(Y))$, and $\mathsf{join} : \mathfrak{M}(\mathfrak{M}(X)) \to \mathfrak{M}(X)$ that satisfy the seven laws given subsequently in Section 10.1, the dependent record $(\mathfrak{M}, \mathsf{unit}, \mathsf{map}, \mathsf{join})$ is a *monad*. The completion monad, $\mathfrak{C}$, operates on metric spaces with uniformly continuous functions. Notice that, although map is required to take and return uniformly continuous functions (see Notation 9.5), it is not required to be uniformly continuous itself in order to form a monad. Later we will see that map is in fact uniformly continuous (Theorem 10.31).

The unit function is the injection of X into $\mathfrak{C}(X)$.

Definition 10.1. $\mathsf{unit}(a) := \lambda\varepsilon.a$.

Notation 10.2. *I will sometimes denote* $\mathsf{unit}(a)$ *by* $\hat{a}$.

Theorem 10.3. *For any* $a\colon X$, $\mathsf{unit}(a)$ *is a regular function, and* unit *is uniformly continuous with modulus* $\mathbf{I}$.

The following two theorems show that the metric is preserved under this injection, and it is sound to think of $x(\varepsilon)$ as an approximation of x within ε.

Theorem 10.4. *For any* a, b, *and* ε, $\boldsymbol{B}^{X}_{\varepsilon}(a,\, b)$ *holds if and only if* $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}\Big(\hat{a},\, \hat{b}\Big)$ *holds.*

Theorem 10.5. *For any* ε, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}\Big(x, \widehat{x(\varepsilon)}\Big)$ *holds.*

Proof. Let δ_1 and δ_2 be arbitrary. Because x is a regular function, $\boldsymbol{B}^{X}_{\delta_1+\varepsilon}(x(\delta_1), x(\varepsilon))$ holds, but this is the same as $\boldsymbol{B}^{X}_{\delta_1+\varepsilon}\Big(x(\delta_1),\ \widehat{x(\varepsilon)}(\delta_2)\Big)$. By Theorem 9.3, $\boldsymbol{B}^{X}_{\delta_1+\varepsilon+\delta_2}\Big(x(\delta_1), \widehat{x(\varepsilon)}(\delta_2)\Big)$ holds as required. □

The function $\mathsf{join} : \mathfrak{C}(\mathfrak{C}(X)) \to \mathfrak{C}(X)$ comes from the proof that the completion of a metric space is complete.

Definition 10.6. $\mathsf{join}(x) := \lambda\varepsilon.x\left(\frac{\varepsilon}{2}\right)\left(\frac{\varepsilon}{2}\right)$.

Theorem 10.7. *For any* $x : \mathfrak{C}(\mathfrak{C}(X))$, $\mathsf{join}(x)$ *is a regular function.*

Proof. Let ε_1 and ε_2 be arbitrary. Because x is a regular function, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\frac{\varepsilon_1}{2}+\frac{\varepsilon_2}{2}}\left(x\left(\frac{\varepsilon_1}{2}\right), x\left(\frac{\varepsilon_2}{2}\right)\right)$ holds. By Theorem 10.5, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\frac{\varepsilon_i}{2}}\left(x\left(\frac{\varepsilon_i}{2}\right), \widehat{x\left(\frac{\varepsilon_i}{2}\right)\left(\frac{\varepsilon_i}{2}\right)}\right)$ holds, so all together $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon_1+\varepsilon_2}\left(\widehat{\mathsf{join}(x)(\varepsilon_1)}, \widehat{\mathsf{join}(x)(\varepsilon_2)}\right)$ holds. By Theorem 10.4, $\boldsymbol{B}^{X}_{\varepsilon_1+\varepsilon_2}(\mathsf{join}(x)(\varepsilon_1), \mathsf{join}(x)(\varepsilon_2))$ holds, so $\mathsf{join}(x)$ is a regular function. □

Theorem 10.8. *The function* join *is uniformly continuous with modulus* $\mathbf{I}$.

Proof. Let ε and δ be arbitrary, and suppose that x_1 and x_2 are such that $\boldsymbol{B}_{\varepsilon}(x_1, x_2)$ holds. We know that $\boldsymbol{B}^{\mathfrak{C}(X)}_{\frac{\delta}{2}}\left(x_i\left(\frac{\delta}{2}\right), \widehat{x_i\left(\frac{\delta}{2}\right)\left(\frac{\delta}{2}\right)}\right)$ holds by Theorem 10.5 and hence $\boldsymbol{B}^{\mathfrak{C}(\mathfrak{C}(X))}_{\frac{\delta}{2}}\left(\widehat{x_i\left(\frac{\delta}{2}\right)}, \widehat{\widehat{x_i\left(\frac{\delta}{2}\right)\left(\frac{\delta}{2}\right)}}\right)$ holds by Theorem 10.4. Also $\boldsymbol{B}^{\mathfrak{C}(\mathfrak{C}(X))}_{\frac{\delta}{2}}\left(x_i, \widehat{x_i\left(\frac{\delta}{2}\right)}\right)$ holds by Theorem 10.5. Together this means that $\boldsymbol{B}^{\mathfrak{C}(\mathfrak{C}(X))}_{\delta}\left(x_i, \widehat{\widehat{x_i\left(\frac{\delta}{2}\right)\left(\frac{\delta}{2}\right)}}\right)$ holds. Therefore, $\boldsymbol{B}^{\mathfrak{C}(\mathfrak{C}(X))}_{\varepsilon+2\delta}\left(\widehat{\widehat{\mathsf{join}(x_1)(\delta)}}, \widehat{\widehat{\mathsf{join}(x_2)(\delta)}}\right)$ holds. By two applications of Theorem 10.4, $\boldsymbol{B}^{X}_{\varepsilon+2\delta}(\mathsf{join}(x_1)(\delta), \mathsf{join}(x_2)(\delta))$ holds. By Theorem 9.23, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\varepsilon}(\mathsf{join}(x_1), \mathsf{join}(x_2))$ holds as required. □

The function $\mathsf{map} : (X \rightarrow Y) \Rightarrow (\mathfrak{C}(X) \rightarrow \mathfrak{C}(Y))$ lifts uniformly continuous functions on the base spaces to uniformly continuous functions on the completed spaces.

Definition 10.9. $\mathsf{map}(f)(x) := \lambda\varepsilon. f\left(x\left(\frac{\breve{\mu}_f(\varepsilon)}{2}\right)\right)$.

Notation 10.10. *I will sometimes write* $\mathsf{map}(f)$ *as* $\bar{f}$.

Theorem 10.11. *If* $f : X \rightarrow Y$ *and* $x : \mathfrak{C}(X)$ *and* δ *and* ε *are such that* $\delta \leq \frac{\mu_f(\varepsilon)}{2}$, *then* $\boldsymbol{B}^{Y}_{\varepsilon}\left(f(x(\delta)), \bar{f}\,(x)(\varepsilon)\right)$.

Proof. Because x is a regular function, $\breve{\boldsymbol{B}}^{X}_{\mu_f(\varepsilon)}\left(x(\delta), x\left(\frac{\mu_f(\varepsilon)}{2}\right)\right)$ holds by Theorem 9.3. Because f is uniformly continuous, $\boldsymbol{B}^{Y}_{\varepsilon}\left(f(x(\delta)), f\left(x\left(\frac{\mu_f(\varepsilon)}{2}\right)\right)\right)$ holds as required. □

Theorem 10.12. *If* $f : X \rightarrow Y$ *and* $x : \mathfrak{C}(X)$, *then* $\bar{f}\,(x)$ *is a regular function.*

Proof. Consider arbitrary ε_1 and ε_2. Let $\delta_i := \mu_f(\varepsilon_i)$. Without loss of generality, assume that $\delta_1 \leq \delta_2$. By Theorem 10.11, $\boldsymbol{B}^Y_{\varepsilon_2}\Big(f\Big(x\Big(\frac{\delta_1}{2}\Big)\Big), \bar{f}\,(x)(\varepsilon_2)\Big)$ hold. By Theorem 9.3, $\boldsymbol{B}^Y_{\varepsilon_1+\varepsilon_2}\big(\bar{f}\,(x)(\varepsilon_1), \bar{f}\,(x)(\varepsilon_2)\big)$ holds as required. □

Theorem 10.13. *The function $\bar{f}$ is uniformly continuous with modulus $\lambda\varepsilon.\frac{\mu_f(\varepsilon)}{2}$.*

Proof. Let x, y, and ε be such that $\breve{\boldsymbol{B}}^{\mathfrak{C}(X)}_{\frac{\mu_f(\varepsilon)}{2}}(x, y)$ holds. Let $\delta := \frac{\mu_f(\varepsilon)}{4}$. Consider arbitrary ε_1 and ε_2. Let $\delta_i := \frac{\mu_f(\varepsilon_i)}{2}$. Let $\delta_i' :=$ min (δ, δ_i). Because x is a regular function, $\breve{\boldsymbol{B}}^X_{\frac{\mu_f(\varepsilon_1)}{2}+\delta_1'}\Big(x\Big(\frac{\mu_f(\varepsilon_1)}{2}\Big), x\big(\delta_1'\big)\Big)$ holds. Therefore, $\breve{\boldsymbol{B}}^X_{\mu_f(\varepsilon_1)}\Big(x\Big(\frac{\mu_f(\varepsilon_1)}{2}\Big), x\big(\delta_1'\big)\Big)$ holds. Hence, $\boldsymbol{B}^Y_{\varepsilon_1}\big(\bar{f}\,(x)(\varepsilon_1),\ f\big(x\big(\delta_1'\big)\big)\big)$ holds because f is uniformly continuous. Similarly, $\boldsymbol{B}^Y_{\varepsilon_2}\big(f(y(\delta_2')), \bar{f}\,(y)(\varepsilon_2)\big)$ holds.

By our hypothesis, $\breve{\boldsymbol{B}}^X_{\delta_1'+\frac{\mu_f(\varepsilon)}{2}+\delta_2'}\big(x\big(\delta_1'\big),\ y\big(\delta_2'\big)\big)$ holds, and hence $\breve{\boldsymbol{B}}^X_{\mu_f(\varepsilon)}\big(x\big(\delta_1'\big), y\big(\delta_2'\big)\big)$ holds. Again, because f is uniformly continuous, $\boldsymbol{B}^Y_{\varepsilon}\big(f\big(x\big(\delta_1'\big)\big), f\big(y\big(\delta_2'\big)\big)\big)$ holds. Therefore, $\boldsymbol{B}^Y_{\varepsilon_1+\varepsilon+\varepsilon_2}\big(\bar{f}\,(x)(\varepsilon_1), \bar{f}\,(y)(\varepsilon_2)\big)$ holds as required. □

There is an alternate definition for map that is more efficient.

Definition 10.14. $\mathsf{map}'(f)(x) := f \circ x \circ \breve{\mu_f}$.

However, there is a catch. The functions map and map′ are only equivalent when X is a prelength space. Consider a case when X is not a prelength space. Let X be the space $\{-1\} \cup \{n^{-1} \mid n\colon \mathbb{N}^+\}$ and Y be the space $\{-2\} \cup \{n^{-1} \mid n\colon \mathbb{N}^+\}$ with the induced metrics. The function $f\colon X \to Y$ sending -1 to -2 and n^{-1} to n^{-1} is uniformly continuous with modulus $\lambda\varepsilon.\,\varepsilon^{\perp 1}$. Consider the regular function x that sends ε to -1 when $1 \leq \varepsilon$ and sends ε to some $n^{-1} \leq \varepsilon$ when $\varepsilon < 1$. In this case, $\mathsf{map}'(f)(x)$ is not a regular function.

Theorem 10.15. *Given $f\colon X \to Y$ where X is a prelength space and Y is a metric space, consider arbitrary a, b, $\varepsilon_0, \ldots, \varepsilon_{n-1}$. Let $\delta_i := \mu_f(\varepsilon_i)$, $\delta := \delta_0 + \ldots + \delta_{n-1}$, and $\varepsilon := \varepsilon_0 + \ldots + \varepsilon_{n-1}$. If $\breve{\boldsymbol{B}}^X_\delta(a, b)$ holds, then $\boldsymbol{B}^Y_\varepsilon(f(a), f(b))$ holds.*

Proof. Let ε_n be arbitrary, and let $\delta_n := \mu_f(\varepsilon_n)$. By Theorem 9.9, there are $c_0, \ldots, c_{n+1}$ such that $\breve{\boldsymbol{B}}^X_{\delta_i}(c_i, c_{i+1})$ holds, $a = c_0$, and $b = c_{n+1}$. By uniform continuity, $\boldsymbol{B}^Y_{\varepsilon_i}(f(c_i),\ f(c_{i+1}))$ holds. Thus $\boldsymbol{B}^Y_{\varepsilon+\varepsilon_n}(f(a),\ f(b))$ holds. Since ε_n was arbitrary, $\boldsymbol{B}^Y_\varepsilon(f(a), f(b))$ holds as required. □

Theorem 10.16. *If $f\colon X \to Y$ where X is a prelength space and Y is a metric space and $x\colon \mathfrak{C}(X)$, then $\mathsf{map}'(f)(x)$ is a regular function.*

Proof. Consider arbitrary ε_1 and ε_2. Let $\delta_i := \mu_f(\varepsilon_i)$. Because x is a regular function, $\breve{\boldsymbol{B}}^X_{\delta_1+\delta_2}(x(\delta_1),\ x(\delta_2))$ holds. By Theorem 10.15, $\boldsymbol{B}^Y_{\varepsilon_1+\varepsilon_2}(f(x(\delta_1)),\ f(x(\delta_2)))$ holds, which is the same as $\boldsymbol{B}^Y_{\varepsilon_1+\varepsilon_2}(\mathsf{map}'(f)(\varepsilon_1), \mathsf{map}'(f)(\varepsilon_2))$ as required. □

Theorem 10.17. *The function* $\mathsf{map}'(f)$ *is uniformly continuous with modulus* μ_f.

Proof. Suppose that $\breve{\boldsymbol{B}}^{\mathfrak{C}(X)}_{\mu_f(\varepsilon)}(x, y)$ holds. Let $\delta := \mu_f(\varepsilon)$. Consider arbitrary ε_1 and ε_2. Let $\delta_i := \mu_f(\varepsilon_i)$. Thus $\breve{\boldsymbol{B}}^X_{\delta_1+\delta+\delta_2}(x(\delta_1), \; y(\delta_2))$ holds. By Theorem 10.15, $\boldsymbol{B}^Y_{\varepsilon_1+\varepsilon+\varepsilon_2}(f(x(\delta_1)), \; f(y(\delta_2)))$ holds. Therefore, $\boldsymbol{B}^Y_\varepsilon(\mathsf{map}'(f)(x), \; \mathsf{map}'(f)(y))$ holds. □

Theorem 10.18. *For any* $f\colon X \to Y$ *and* $x\colon X$, *where* X *is a prelength space,* $\mathsf{map}'(f)(x) \asymp \mathsf{map}(f)(x)$.

Proof. Let ε_1 and ε_2 be arbitrary. Let $\delta_i := \mu_f(\varepsilon_i)$. Because x is a regular function, $\breve{\boldsymbol{B}}^X_{\delta_1+\frac{\delta_2}{2}}\Big(x(\delta_1), x\Big(\frac{\delta_2}{2}\Big)\Big)$ holds and hence $\breve{\boldsymbol{B}}^X_{\delta_1+\delta_2}\Big(x(\delta_1), x\Big(\frac{\delta_2}{2}\Big)\Big)$ holds. By Theorem 10.15, $\boldsymbol{B}^Y_{\varepsilon_1+\varepsilon_2}\Big(f(x(\delta_1)), \; f\Big(x\Big(\frac{\delta_2}{2}\Big)\Big)\Big)$ holds which is the same as $\boldsymbol{B}^Y_{\varepsilon_1+\varepsilon_2}(\mathsf{map}'(f)(x)(\varepsilon_1), \mathsf{map}(f)(x)(\varepsilon_2))$ as required. □

Notice that map' not only avoids dividing the modulus by 2 in its definition, it also avoids dividing by 2 in the modulus of continuity of $\bar{f}$. The metric spaces we typically work with are prelength spaces, so in practice we use map' instead of map. In fact, this entire project was developed using map' first. It was only after reading “Real numbers and other completions” [Richman, 2008] that I learned how to write map without requiring the assumption that X is a prelength space.

The function $\mathsf{bind} : (X \to \mathfrak{C}(Y)) \Rightarrow (\mathfrak{C}(X) \to \mathfrak{C}(Y))$ is defined in terms of join and map in the usual way.

Definition 10.19. $\mathsf{bind}(f) := \mathsf{join} \circ \mathsf{map}(f)$.

Notation 10.20. *I will sometimes denote* $\mathsf{bind}(f)$ *as* $\check{f}$.

Similarly there is an equivalent bind'.

Definition 10.21. $\mathsf{bind}'(f) := \mathsf{join} \circ \mathsf{map}'(f)$.

10.1 Monad Laws

To prove that completion forms a monad, we need to verify that the seven monad laws [Wadler, 1995] hold.

Theorem 10.22. $\bar{\mathbf{I}} \asymp \mathbf{I}$.

Proof. Let $x\colon \mathfrak{C}(X)$ be arbitrary. Because x is a regular function, $\boldsymbol{B}^X_{\frac{\varepsilon_1}{2}+\varepsilon_2}\big(x\big(\frac{\varepsilon_1}{2}\big), x(\varepsilon_2)\big)$ holds. Hence $\boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}\big(\bar{\mathbf{I}}(x)(\varepsilon_1), \mathbf{I}(x)(\varepsilon_2)\big)$ holds and $\bar{\mathbf{I}}(x) \asymp \mathbf{I}(x)$ as required. □

Theorem 10.23. $\forall g\colon X \to Y . \forall f\colon Y \to Z .\, \overline{f \circ g} \asymp \bar{f} \circ \bar{g}$.

Proof. Let ε_1, ε_2, and $x\colon \mathfrak{C}(X)$ be arbitrary. Let $\delta_i := \mu_f(\varepsilon_i)$. Let $\gamma := \mathsf{min}\left(\breve{\mu_g}(\delta_1), \breve{\mu_g}\left(\frac{\delta_2}{2}\right)\right)$. By Theorem 10.11, $\breve{\boldsymbol{B}}^{Y}_{\frac{\delta_2}{2}}\left(g\left(x\left(\frac{\gamma}{2}\right)\right),\ \bar{g}\,(x)\left(\frac{\delta_2}{2}\right)\right)$ holds and hence $\breve{\boldsymbol{B}}^{Y}_{\delta_2}\left(g\left(x\left(\frac{\gamma}{2}\right)\right),\ \bar{g}\,(x)\left(\frac{\delta_2}{2}\right)\right)$ holds. From this we know that $\boldsymbol{B}^{Z}_{\varepsilon_2}\left(f\left(g\left(x\left(\frac{\gamma}{2}\right)\right)\right), \bar{f} \circ \bar{g}\,(x)(\varepsilon_2)\right)$ holds.

Again, by Theorem 10.11, $\boldsymbol{B}^{Z}_{\varepsilon_1}\left((f \circ g)\left(x\left(\frac{\gamma}{2}\right)\right), \overline{f \circ g}\,(x)(\varepsilon_1)\right)$ holds. Therefore, by the triangle law, $\boldsymbol{B}^{Z}_{\varepsilon_1+\varepsilon_2}\left(\overline{f \circ g}\,(x)(\varepsilon_1), \bar{f} \circ \bar{g}\,(x)(\varepsilon_2)\right)$ as required. □

Theorem 10.24. $\forall f\colon X \to Y .\, \bar{f} \circ \mathsf{unit} \asymp \mathsf{unit} \circ f$.

Proof. Let ε and $x\colon X$ be arbitrary. We have $\bar{f}\,(\hat{x})(\varepsilon) = f(x)$ and $\widehat{f(x)}(\varepsilon) = f(x)$, so $\bar{f}\,(\hat{x}) \asymp \widehat{f(x)}$ holds as required. □

Theorem 10.25. $\forall f\colon X \to Y .\, \bar{f} \circ \mathsf{join} \asymp \mathsf{join} \circ \mathsf{map}(\bar{f}\,)$.

Proof. Let ε_1, ε_2, and $x\colon \mathfrak{C}(\mathfrak{C}(X))$ be arbitrary. Let $\delta_1 := \mu_f(\varepsilon_1)$, $\delta_2 := \mu_f\left(\frac{\varepsilon_2}{2}\right)$, and $\delta := \min\left(\frac{\delta_1}{4}, \frac{\delta_2}{8}\right)$. By Theorem 10.5, $\breve{\boldsymbol{B}}^{\mathfrak{C}(\mathfrak{C}(X))}_{\frac{\delta_1}{4}}\left(\widehat{x\left(\frac{\delta_1}{4}\right)}, x\right)$ holds. By Theorem 10.4 and Theorem 10.5, $\breve{\boldsymbol{B}}^{\mathfrak{C}(\mathfrak{C}(X))}_{\frac{\delta_1}{4}}\left(\widehat{\widehat{x\left(\frac{\delta_1}{4}\right)\left(\frac{\delta_1}{4}\right)}},\ \widehat{x\left(\frac{\delta_1}{4}\right)}\right)$ holds. Therefore $\breve{\boldsymbol{B}}^{\mathfrak{C}(\mathfrak{C}(X))}_{\frac{\delta_1}{2}}\left(\widehat{\widehat{x\left(\frac{\delta_1}{4}\right)\left(\frac{\delta_1}{4}\right)}}, x\right)$ holds. Because $\delta \leq \frac{\delta_1}{4}$, $\breve{\boldsymbol{B}}^{\mathfrak{C}(\mathfrak{C}(X))}_{\frac{\delta_1}{2}}\left(x, \widehat{\widehat{x(\delta)(\delta)}}\right)$ holds by similar reasoning. This means that $\breve{\boldsymbol{B}}^{X}_{\delta_1}\left(x\left(\frac{\delta_1}{4}\right)\left(\frac{\delta_1}{4}\right),\ x(\delta)(\delta)\right)$ holds by the triangle law and Theorem 10.4. Because f is uniformly continuous, $\boldsymbol{B}^{Y}_{\varepsilon_1}\left(f\left(x\left(\frac{\delta_1}{4}\right)\left(\frac{\delta_1}{4}\right)\right), f(x(\delta)(\delta))\right)$ holds.

Because $\delta \leq \frac{\delta_2}{8}$, we can conclude that $\breve{\boldsymbol{B}}^{\mathfrak{C}(\mathfrak{C}(X))}_{\frac{\delta_2}{4}}\left(\widehat{\widehat{x(\delta)(\delta)}}, x\right)$ holds by reasoning similar to the above. We can also conclude that $\breve{\boldsymbol{B}}^{\mathfrak{C}(\mathfrak{C}(X))}_{\frac{3\delta_2}{4}}\left(x, \widehat{\widehat{x\left(\frac{\delta_2}{4}\right)\left(\frac{\delta_2}{2}\right)}}\right)$ holds by the same kind of argument. Again, by the triangle law and Theorem 10.4, $\breve{\boldsymbol{B}}^{X}_{\delta_2}\left(x(\delta)(\delta),\ x\left(\frac{\delta_2}{4}\right)\left(\frac{\delta_2}{2}\right)\right)$ holds. Because f is uniformly continuous, $\boldsymbol{B}^{Y}_{\frac{\varepsilon_2}{2}}\left(f(x(\delta)(\delta)),\ f\left(x\left(\frac{\delta_2}{4}\right)\left(\frac{\delta_2}{2}\right)\right)\right)$ holds, and hence $\boldsymbol{B}^{Y}_{\varepsilon_2}\left(f(x(\delta)(\delta)), f\left(x\left(\frac{\delta_2}{4}\right)\left(\frac{\delta_2}{2}\right)\right)\right)$ holds.

By putting these two results together, we get $\boldsymbol{B}^{Y}_{\varepsilon_1+\varepsilon_2}\left(f\left(x\left(\frac{\delta_1}{4}\right)\left(\frac{\delta_1}{4}\right)\right), f\left(x\left(\frac{\delta_2}{4}\right)\left(\frac{\delta_2}{2}\right)\right)\right)$, which is exactly $\boldsymbol{B}^{Y}_{\varepsilon_1+\varepsilon_2}\left(\bar{f}\,(\mathsf{join}(x))(\varepsilon_1), \mathsf{join}(\mathsf{map}(\bar{f}\,)(x))(\varepsilon_2)\right)$ as required. □

Theorem 10.26. $\mathsf{join} \circ \mathsf{unit} \asymp \mathbf{I}$.

Proof. Let ε_1, ε_2, and x: $\mathfrak{C}(X)$. be arbitrary. Since x is a regular function, $\boldsymbol{B}^X_{\frac{\varepsilon_1}{2}+\varepsilon_2}\big(x\big(\frac{\varepsilon_1}{2}\big),\ x(\varepsilon_2)\big)$ holds and therefore $\boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}\big(x\big(\frac{\varepsilon_1}{2}\big),\ x(\varepsilon_2)\big)$ holds. But this is exactly $\boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}(\mathsf{join}(\hat{x})(\varepsilon_1), \mathbf{I}(x)(\varepsilon_2))$ as required. □

Theorem 10.27. $\mathsf{join} \circ \overline{\mathsf{unit}} \asymp \mathbf{I}$.

Proof. Let ε_1, ε_2, and x: $\mathfrak{C}(X)$. be arbitrary. Since x is a regular function, $\boldsymbol{B}^X_{\frac{\varepsilon_1}{4}+\varepsilon_2}\big(x\big(\frac{\varepsilon_1}{4}\big),\ x(\varepsilon_2)\big)$ holds and therefore $\boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}\big(x\big(\frac{\varepsilon_1}{4}\big),\ x(\varepsilon_2)\big)$ holds. But this is exactly $\boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}(\mathsf{join}(\overline{\mathsf{unit}}(x))(\varepsilon_1), \mathbf{I}(x)(\varepsilon_2))$ as required. □

Theorem 10.28. $\mathsf{join} \circ \overline{\mathsf{join}} \asymp \mathsf{join} \circ \mathsf{join}$.

Proof. Let ε_1, ε_2, and x: $\mathfrak{C}(\mathfrak{C}(\mathfrak{C}(X)))$ be arbitrary. Since x is a regular function, $\boldsymbol{B}^{\mathfrak{C}(\mathfrak{C}(X))}_{\frac{\varepsilon_1}{4}+\frac{\varepsilon_2}{4}}\big(x\big(\frac{\varepsilon_1}{4}\big),\ x\big(\frac{\varepsilon_2}{4}\big)\big)$ holds. This means that $\boldsymbol{B}^{\mathfrak{C}(X)}_{\frac{\varepsilon_1}{2}+\frac{\varepsilon_2}{2}}\big(x\big(\frac{\varepsilon_1}{4}\big)\big(\frac{\varepsilon_1}{4}\big),\ x\big(\frac{\varepsilon_2}{4}\big)\big(\frac{\varepsilon_2}{4}\big)\big)$ holds, which in turn means $\boldsymbol{B}^X_{\frac{3\varepsilon_1}{4}+\varepsilon_2}\big(x\big(\frac{\varepsilon_1}{4}\big)\big(\frac{\varepsilon_1}{4}\big)\big(\frac{\varepsilon_1}{4}\big), x\big(\frac{\varepsilon_2}{4}\big)\big(\frac{\varepsilon_2}{4}\big)\big(\frac{\varepsilon_2}{2}\big)\big)$ holds. Therefore $\boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}\big(x\big(\frac{\varepsilon_1}{4}\big)\big(\frac{\varepsilon_1}{4}\big)\big(\frac{\varepsilon_1}{4}\big),\ x\big(\frac{\varepsilon_2}{4}\big)\big(\frac{\varepsilon_2}{4}\big)\big(\frac{\varepsilon_2}{2}\big)\big)$ holds, but this is exactly $\boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}(\mathsf{join}(\overline{\mathsf{join}}(x))(\varepsilon_1), \mathsf{join}(\mathsf{join}(x))(\varepsilon_2))$ as required. □

Twice completing a metric space is isomorphic to completing it once, $\mathfrak{C}(\mathfrak{C}(X)) \leftrightarrow \mathfrak{C}(X)$. A monad with this property is called an *idempotent monad*. Theorem 10.26 gives half of this isomorphism. The following theorem (which is not a monad law) completes this isomorphism.

Theorem 10.29. $\mathsf{unit} \circ \mathsf{join} \asymp \mathbf{I}$.

Proof. Let ε_1, ε_2, δ_1, and δ_2 be arbitrary. Let x: $\mathfrak{C}(\mathfrak{C}(X))$ be arbitrary. Because x is a regular function, $\boldsymbol{B}^X_{\delta_1+\varepsilon_2+\delta_2}\Big(x\Big(\frac{\delta_1}{2}\Big)\Big(\frac{\delta_1}{2}\Big),\quad x(\varepsilon_2)(\delta_2)\Big)$ holds. Hence $\boldsymbol{B}^X_{\delta_1+\varepsilon_1+\varepsilon_2+\delta_2}(\mathsf{join}(x)(\delta_1),\ x(\varepsilon_2)(\delta_2))$ holds. Therefore $\boldsymbol{B}^X_{\delta_1+\varepsilon_1+\varepsilon_2+\delta_2}\Big(\widehat{\mathsf{join}(x)}(\varepsilon_1)(\delta_1),$ $x(\varepsilon_2)(\delta_2)\Big)$ holds, which means that $\mathsf{unit}(\mathsf{join}(x)) \asymp x$ as required. □

10.2 Lifting Binary Functions

The function map can be used to lift unary functions from $X \to Y$ to $\mathfrak{C}(X) \to \mathfrak{C}(Y)$; however, one also wants to lift multi-argument functions such as binary functions. One can lift curried binary functions on base spaces to curried functions on completed spaces, but one first needs to consider the space of uniformly continuous functions as a metric space. This is done using the supremum metric.

Theorem 10.30. *For any two metric spaces* $(X, \asymp, \boldsymbol{B}^X)$ *and* $(Y, \asymp, \boldsymbol{B}^Y)$*, the space of uniformly continuous functions from* X *to* Y*,* $(X \to Y, \asymp, \boldsymbol{B}^{X \to Y})$*, is a metric space where* $\boldsymbol{B}^{X \to Y}_{\varepsilon}(f, g) := \forall a, \boldsymbol{B}^Y_{\varepsilon}(f(a), g(a))$ *and with the equivalence relation* $f \asymp g := \forall a.\, f(a) \asymp g(a)$*.*

Proof. All the properties of $\boldsymbol{B}^{X \to Y}$follow directly from the same properties for $\boldsymbol{B}^Y$. □

Unfortunately, the space of uniformly continuous functions between two prelength spaces is not necessarily a prelength space. Consider the space of curves on the unit circle, $[0,1] \to S^1$. Consider two curves connecting two points where one goes clockwise and the other goes counter-clockwise. If $[0,1] \to S^1$ were a prelength space, then there would be a curve about halfway between these two curves; however, there is no such curve.

Fortunately, one can still lift curried functions using the more efficient map′, because the theorem that the result of map′ is a regular function, Theorem 10.16, requires only that the *domain* be a prelength space. With curried functions (say $X \to (X \to X)$), it is the *range* that may not be a prelength space. The domain is still the prelength space X, so Theorem 10.16 applies.

A monad $\mathfrak{M}$ is a *strong monad* when there is a morphism strength : $X \times \mathfrak{M}(Y) \to \mathfrak{M}(X \times Y)$ satisfying particular laws [Moggi, 1989]. In Section 13.1, we will see that metric spaces are Cartesian, meaning that we can take the cross product of two metric spaces. We will also see that $\mathfrak{C}$ is a symmetric monoidal monad, and therefore, $\mathfrak{C}$ is a (commutative) strong monad. If metric spaces formed a Cartesian closed category, then strength could be used to prove that map is a morphism [Kock, 1972]. Unfortunately, our category of metric spaces is not closed because evaluation is not uniformly continuous (see Section 10.3.2). Therefore, we must prove directly that map is uniformly continuous. This is known as *functorial strength.*

Theorem 10.31. *Let* X *and* Y *be metric spaces. Then* map : $(X \to Y) \to (\mathfrak{C}(X) \to \mathfrak{C}(Y))$ *is uniformly continuous with modulus* $\mathbf{I}$.

Proof. Let ε be arbitrary. Suppose that f_1 $f_2 : X \to Y$ are such that $\boldsymbol{B}^{X \to Y}_{\varepsilon}(f_1, f_2)$. Let ε_0 and $x \colon \mathfrak{C}(X)$ be arbitrary. Let $\delta_0 :=$ min $\left(\mu_{\overline{f_1}}(\frac{\varepsilon_0}{2}), \mu_{\overline{f_2}}(\frac{\varepsilon_0}{2})\right)$ and $a := x(\delta_0)$. We know that $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\frac{\varepsilon_0}{2}}\left(\bar{f_i}(x), \bar{f_i}(\hat{a})\right)$ holds by Theorem 10.5 and uniform continuity. By Theorem 10.24, $\bar{f_i}(\hat{a}) \asymp \widehat{f_i(a)}$. By Theorem 10.4, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\varepsilon}\left(\widehat{f_1(a)}, \widehat{f_2(a)}\right)$ holds, so all together $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\varepsilon+\varepsilon_0}\left(\bar{f}_1(x), \bar{f}_2(x)\right)$. Since ε_0 and x were arbitrary, $\boldsymbol{B}^{\mathfrak{C}(X) \to \mathfrak{C}(Y)}_{\varepsilon}\left(\bar{f}_1, \bar{f}_2\right)$ holds as required. □

Corollary 10.32. *Let* X *be a prelength space and* Y *a metric space. Then* map′ : $(X \to Y) \to (\mathfrak{C}(X) \to \mathfrak{C}(Y))$ *is uniformly continuous with modulus* $\mathbf{I}$.

We also want to construct the function $\mathsf{ap} : \mathfrak{C}(X \to Y) \to \mathfrak{C}(X) \to \mathfrak{C}(Y)$, which one can think of as a function that takes the limit of uniformly continuous functions (see also Section 12.1.3). Again, if we were in a Cartesian closed category, then we would be able to define ap using map, but because the category of metric spaces is not Cartesian closed, we define ap directly.

Definition 10.33. $\mathsf{ap}(f)(x) := \lambda\varepsilon.\, \mathsf{map}\big(f\big(\tfrac{\varepsilon}{2}\big)\big)(x)\big(\tfrac{\varepsilon}{2}\big)$.

Theorem 10.34. *For any $f\colon \mathfrak{C}(X \to Y)$ and $x\colon \mathfrak{C}(X)$, $\mathsf{ap}(f)(x)$ is a regular function.*

Proof. Let ε_1 and ε_2 be arbitrary. Let $f_i := f\big(\frac{\varepsilon_i}{2}\big)$. Because f is regular, $\boldsymbol{B}^{X\to Y}_{\frac{\varepsilon_1}{2}+\frac{\varepsilon_2}{2}}(f_1,\, f_2)$ holds. Let $y_i := \bar{f_i}\,(x)$. By Theorem 10.31, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\frac{\varepsilon_1}{2}+\frac{\varepsilon_2}{2}}(y_1,\, y_2)$ holds. By Theorem 10.5, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\frac{\varepsilon_i}{2}}\Big(\widehat{y_i\big(\frac{\varepsilon_i}{2}\big)},\ y_i\Big)$ holds. All together $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\varepsilon_1+\varepsilon_2}\Big(\widehat{y_1\big(\frac{\varepsilon_1}{2}\big)},\ \widehat{y_2\big(\frac{\varepsilon_2}{2}\big)}\Big)$ holds. By Theorem 10.4, $\boldsymbol{B}^{Y}_{\varepsilon_1+\varepsilon_2}\big(y_1\big(\frac{\varepsilon_1}{2}\big), y_2\big(\frac{\varepsilon_2}{2}\big)\big)$ holds as required. □

Theorem 10.35. *For any $f\colon \mathfrak{C}(X \to Y)$, $x\colon \mathfrak{C}(X)$, and ε, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\varepsilon}\big(\mathsf{ap}(f)(x), \overline{f(\varepsilon)}(x)\big)$ holds.*

Proof. Let ε_1 and ε_2 be arbitrary. Let $f_1 := f\big(\frac{\varepsilon_1}{2}\big)$ and $f_2 := f(\varepsilon)$. Because f is regular, $\boldsymbol{B}^{X\to Y}_{\frac{\varepsilon_1}{2}+\varepsilon}(f_1,\, f_2)$ holds. By Theorem 10.31, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\frac{\varepsilon_1}{2}+\varepsilon}\big(\bar{f_1}(x),\, \bar{f_2}(x)\big)$ holds. By Theorem 10.5, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\frac{\varepsilon_1}{2}}\Big(\widehat{\bar{f_1}(x)\big(\frac{\varepsilon_1}{2}\big)},\ \bar{f_1}(x)\Big)$ and $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\varepsilon_2}\Big(\bar{f_2}(x),\ \widehat{\bar{f_2}(x)(\varepsilon_2)}\Big)$ hold. All together $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\varepsilon_1+\varepsilon+\varepsilon_2}\Big(\widehat{\bar{f_1}(x)\big(\frac{\varepsilon_1}{2}\big)},\ \widehat{\bar{f_2}(x)(\varepsilon_2)}\Big)$ holds, and by Theorem 10.4, $\boldsymbol{B}^{Y}_{\varepsilon_1+\varepsilon+\varepsilon_2}\big(\bar{f_1}(x)\big(\frac{\varepsilon_1}{2}\big),\ \bar{f_2}(x)(\varepsilon_2)\big)$ holds. Therefore, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\varepsilon}\big(\mathsf{ap}(f)(x),\ \bar{f_2}(x)\big)$ holds as required. □

Theorem 10.36. *For any $f\colon \mathfrak{C}(X \to Y)$, $\mathsf{ap}(f)$ is uniformly continuous with modulus $\lambda\varepsilon.\, \frac{\mu_{f(\frac{\varepsilon}{3})}(\frac{\varepsilon}{3})}{2}$.*

Proof. Let ε be arbitrary, and let $\delta := \mu_{\overline{f(\frac{\varepsilon}{3})}}\big(\frac{\varepsilon}{3}\big)$. Suppose that x_1 and x_2 are such that $\breve{\boldsymbol{B}}^{\mathfrak{C}(X)}_{\delta}(x_1,\, x_2)$ holds. Then, by Theorem 10.13, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\frac{\varepsilon}{3}}\Big(\overline{f\big(\frac{\varepsilon}{3}\big)}(x_1),\ \overline{f\big(\frac{\varepsilon}{3}\big)}(x_2)\Big)$ holds. By Theorem 10.35, $\boldsymbol{B}^{\mathfrak{C}(X)}_{\frac{\varepsilon}{3}}\Big(\mathsf{ap}(f)(x_i),\ \overline{f\big(\frac{\varepsilon}{3}\big)}(x_i)\Big)$ holds. Together this means that $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\varepsilon}(\mathsf{ap}(f)(x_1), \mathsf{ap}(f)(x_2))$ holds as required. □

Theorem 10.37. *The function ap is uniformly continuous with modulus $\mathbf{I}$.*

Proof. Let ε be arbitrary. Suppose that f_1 and f_2 are such that $\boldsymbol{B}^{\mathfrak{C}(X\to Y)}_{\varepsilon}(f_1,\ f_2)$ holds. Let x: $\mathfrak{C}(X)$ and δ be arbitrary. Because f_1 and f_2 are regular functions, $\boldsymbol{B}^{X\to Y}_{\frac{\delta}{2}+\varepsilon}\Big(f_1\big(\frac{\delta}{4}\big),\ f_2\big(\frac{\delta}{4}\big)\Big)$ holds. Therefore, by Theorem 10.31, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\frac{\delta}{2}+\varepsilon}\Big(\overline{f_1\big(\frac{\delta}{4}\big)}(x),\ \overline{f_2\big(\frac{\delta}{4}\big)}(x)\Big)$ holds. By Theorem 10.35, $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\frac{\delta}{4}}\Big(\mathsf{ap}(f_i)(x),\ \overline{f_i\big(\frac{\delta}{4}\big)}(x)\Big)$ holds. Together this yields $\boldsymbol{B}^{\mathfrak{C}(Y)}_{\delta+\varepsilon}(\mathsf{ap}(f_1)(x),\ \ \mathsf{ap}(f_2)(x))$, and since δ and x were arbitrary, $\boldsymbol{B}_{\varepsilon}(\mathsf{ap}(f_1), \mathsf{ap}(f_2))$ holds as required. □

Theorem 10.38. *For any* $f\colon X\to Y$, $\mathsf{ap}\Big(\hat{f}\Big)\asymp\bar{f}$.

Proof. Let ε_1, ε_2, and x: $\mathfrak{C}(X)$ be arbitrary. Because $\bar{f}\,(x)$ is a regular function, $\boldsymbol{B}^{X}_{\frac{\varepsilon_1}{2}+\varepsilon_2}\big(\bar{f}\,(x)\big(\frac{\varepsilon_1}{2}\big),\ \bar{f}\,(x)(\varepsilon_2)\big)$ holds and hence $\boldsymbol{B}^{X}_{\varepsilon_1+\varepsilon_2}\big(\bar{f}\,(x)\big(\frac{\varepsilon_1}{2}\big),\ \bar{f}\,(x)(\varepsilon_2)\big)$ holds. Because $\mathsf{ap}\Big(\hat{f}\Big)(x)(\varepsilon_1) = \bar{f}\,(x)\big(\frac{\varepsilon_1}{2}\big)$, we have $\boldsymbol{B}^{X}_{\varepsilon_1+\varepsilon_2}\Big(\mathsf{ap}\Big(\hat{f}\Big)(x)(\varepsilon_1),\ \bar{f}\,(x)(\varepsilon_2)\Big)$ as required. □

By using ap, we can define a $\mathsf{map2}$ function that lifts a curried function f: $X\to Y\to Z$ to $\mathsf{map2}(f)\colon\mathfrak{C}(X)\to\mathfrak{C}(Y)\to\mathfrak{C}(Z)$.

Definition 10.39. $\mathsf{map2}(f) := \mathsf{ap}\circ\mathsf{map}(f)$.

Other functions can be made to lift n-ary functions by repeated use of ap and map.

We also define equivalent versions of ap and $\mathsf{map2}$ that use map', which work when the appropriate domains are prelength spaces.

Definition 10.40. $\mathsf{ap}'(f)(x) := \lambda\varepsilon.\,\mathsf{map}'\big(f\big(\frac{\varepsilon}{2}\big)\big)(x)\big(\frac{\varepsilon}{2}\big)$.

Definition 10.41. $\mathsf{map2}'(f) := \mathrm{ap}'\circ\mathsf{map}'(f)$.

The same theorems that hold for ap also hold for ap' with the exception that the modulus of continuity for $\mathsf{ap}'(f)$ can be improved.

Theorem 10.42. *For any* $f\colon\mathfrak{C}(X\to Y)$, $\mathsf{ap}'(f)$ *is uniformly continuous with modulus* $\lambda\varepsilon.\,\mu_{f(\frac{\varepsilon}{3})}\big(\frac{\varepsilon}{3}\big)$.

Proof. This proof is similar to the proof of Theorem 10.36. The improved modulus of continuity is a consequence of improved modulus of continuity for map' given by Theorem 10.17. □

10.3 Remarks

10.3.1 Curried Uniformly Continuous Functions

There is an advantage to using curried functions over using uncurried functions. Consider some function $f\colon X \to (Y \to Z)$. The result of applying f to some value a: X is itself a uniformly continuous function $f(a)\colon Y \to Z$. The advantage is that the modulus of continuity of $f(a)$ is allowed to depend on the value of a. This means that when $\mathsf{map2}(f)(x)(y)$: $\mathfrak{C}(Z)$ is approximated to within ε, the computation will first approximate x: $\mathfrak{C}(X)$ to within $\mu_f\left(\frac{\varepsilon}{2}\right)$, then y: $\mathfrak{C}(X)$ will be approximated by using a modulus that depends on the approximation of x. The extra information given by the approximation of x can potentially be used to save work by reducing accuracy required from y. If f were uncurried, $X \times Y \to Z$, then one modulus of continuity for f would need to apply uniformly for all $(x, y)\colon X \times Y$.

This advantage does not work symmetrically in the two parameters of a uniformly continuous function. When the value $\mathsf{map2}(f)(x)(y)$: $\mathfrak{C}(Z)$ is approximated, the approximation for x: $\mathfrak{C}(X)$ cannot depend on the value of the approximation of y: $\mathfrak{C}(Y)$ because of the way that the parameters are ordered. Therefore, one should consider the order of the parameters of uniformly continuous functions in order to have optimal efficiency.

I believe we could define $\mathsf{curry} : (X \times Y \to Z) \Rightarrow X \to Y \to Z$. However, I do not believe that we can define $\mathsf{uncurry} : (X \to Y \to Z) \Rightarrow X \times Y \to Z$ because there is no way to construct a uniform modulus of continuity from the X-indexed set of uniformly continuous functions. Since I work with curried functions directly, I did not pursue the development of curry.

10.3.2 Cartesian but not Closed

Theorem 9.7 showed that metric spaces form a category. In Section 13.1, we will show that this category is Cartesian, meaning that we can take the cross product of two metric spaces. However, this category is not closed because we cannot construct a uniformly continuous $\mathsf{eval} : X \times (X \to Y) \to Y$ function. The problem is that we cannot find a modulus of continuity for eval that works uniformly for all inputs. In particular it has to work for all $f\colon X \to Y$. Suppose μ_{eval} did exist. Then given a and b such that $\boldsymbol{B}^X_{\mu_{\mathsf{eval}}(\varepsilon)}(a, b)$ holds, then $\boldsymbol{B}^Y_\varepsilon(\mathsf{eval}(a,\ f), \mathsf{eval}(b,\ f))$ would hold. But this would be equivalent to $\boldsymbol{B}^Y_\varepsilon(f(a),\ f(b))$, which means that μ_{eval} would be a modulus of continuity for f. However, f was arbitrary. We do not have one modulus of continuity that works for every function, so this is impossible. A similar argument shows that there is no uniformly continuous $\mathsf{compose} : (Y \to Z) \times (X \to Y) \to X \to Z$ function, so the category is not an enriched category over itself either.

If we worked with a restricted set of uniformly continuous functions that all shared one global modulus, for example, the set of uniformly continuous non-expansive maps, then we would have a Cartesian closed category. In this case, eval would be uniformly continuous with the global modulus, and any function in the simply typed lambda calculus could be interpreted in this category [Lambek, 1980].

10.3.3 Errors Found During Formalization

I completed paper proofs of the previous theorems before formalizing them in Coq. I published these paper proofs in Mathematical Structures in Computer Science [O'Connor, 2007]. Because of this preparation, the process of computer formalization was straightforward. Still, a few errors were uncovered during formalization. These errors have been corrected in this thesis.

Most errors found were very minor. For example, my original proof of Theorem 9.24 failed to consider the case when γ is too big. Problems with epsilons being too large were easy to fix. Coq did catch one more substantial error in my original proof of Theorem 10.8 with the error message.

```
Error: Impossible to unify
 "ball (m:=Complete (Complete X)) ((1 # 2) * d1)
    (Cunit (Cunit (x ((1 # 2) * d1)%Qpos) ((1 # 2) * d1)%Qpos))
    (Cunit (x ((1 # 2) * d1)%Qpos))" with
 "ball (m:=Complete (Complete X)) ((1 # 2) * d1)
    (Cunit (Cunit (x ((1 # 2) * d1)%Qpos ((1 # 2) * d1)%Qpos)))
    (Cunit (x ((1 # 2) * d1)%Qpos))"
```

It may take you a moment to find the difference between these two expressions. I had bitten off a bit more than I could chew when I tried to conclude $\boldsymbol{B}_{\delta}^{\mathfrak{C}(\mathfrak{C}(X))}\left(x_i, \widehat{\widehat{x_i\left(\frac{\delta}{2}\right)\left(\frac{\delta}{2}\right)}}\right)$ directly. Although it does hold, it took one more step in reasoning than I had originally figured.

I also found one "error" in my Coq proof during the final preparation of this chapter. Of course, it was not a logical error because Coq would not allow that. Instead it was an efficiency error. When creating ap′, I had neglected to give it its better modulus of continuity found in Theorem 10.42, and instead had given it the same modulus found in Theorem 10.36. This "error" has been corrected in the Coq code.

Chapter 11
Real Numbers

Using the completion monad, it is now easy to define the *real numbers*.

Definition 11.1. $\mathbb{R} := \mathfrak{C}(\mathbb{Q})$.

Functions operating on the real numbers can be created by using map' or bind' to lift uniformly continuous functions operating on the rationals. This is the advantage of using the monad functions. It is often easier to define a function on the rationals than on the reals, because it is possible to decide for two rational numbers a and b which of $a < b$, $a = b$, or $a > b$ hold. The real numbers do not have this property.

Theorem 11.2.

- *For all* $a:\mathbb{Q}$*, the functions* $\lambda b:\mathbb{Q}.a+b$ *and* $\lambda b:\mathbb{Q}.-b$ *are both uniformly continuous with moduli* $\mathbf{I}$.
- *For all* $a:\mathbb{Q}$*,* $\lambda b:\mathbb{Q}.\,b^{\perp a}$ *is uniformly continuous with modulus* $\mathbf{I}$.
- *For all* $a:\mathbb{Q}$*,* $\lambda b:\mathbb{Q}.\,b_{\top a}$ *is uniformly continuous with modulus* $\mathbf{I}$.
- *The function* $\lambda b:\mathbb{Q}.|b|$ *is uniformly continuous with modulus* $\mathbf{I}$.
- *For all* $a:\mathbb{Q}$*,* $\lambda b:\mathbb{Q}.a\ b$ *is uniformly continuous with modulus* $\lambda\varepsilon.\ \frac{\varepsilon}{|a|}$ *(or* $\lambda\varepsilon.\infty$ *in the case when* $a = 0$*).*

By using map', all of these unary functions can be lifted to functions on $\mathbb{R}$. For the moment, one of the arguments to the binary functions $+$, $\cdot$, min and max must be rational. For example, for any $a:\mathbb{Q}$, the unary function $\mathsf{map}'\,(\lambda b:\mathbb{Q}.a+b):\mathbb{R}\to\mathbb{R}$ is the function that translates by the rational number a.

At this point, one can see why regular functions work better than regular sequences. For multiplication by the constant $a:\mathbb{Q}$, the modulus of continuity is $\lambda\varepsilon.\ \frac{\varepsilon}{|a|}$. This means that when $\mathsf{map}'(\lambda b.a\ b)(x)$ is approximated by ε, then x is approximated by $\frac{\varepsilon}{|a|}$, which is precisely the minimum approximation needed from x. If one used regular sequences, then one would need to round $\frac{\varepsilon}{|a|}$ down to the nearest number of the form $\frac{1}{n}$ before proceeding. Extra work may be done because x is over-approximated. For a deep computation, all these over-approximations could multiply causing significant excess work.

11.1 Binary Functions of Real Numbers

One can use $\mathsf{map2}'$ to lift addition to operate on $\mathbb{R}$.

Theorem 11.3. *The function $\lambda a\colon \mathbb{Q}.\lambda b\colon \mathbb{Q}.a+b$ is uniformly continuous from $\mathbb{Q}$ to $\mathbb{Q}\to\mathbb{Q}$ with modulus $\mathbf{I}$.*

The definition of addition on $\mathbb{R}$ is

Definition 11.4. $x+y := \mathsf{map2}'(\lambda a\colon \mathbb{Q}.\lambda b\colon \mathbb{Q}.a+b)(x,y)$.

The definition of multiplication on $\mathbb{R}$ is a little more complicated than for addition.

Theorem 11.5. *For any $c\colon \mathbb{Q}^+$, the function $\lambda a.\lambda b.a\,\underset{\top -c}{\overset{\perp c}{b}}$ is uniformly continuous from $\mathbb{Q}$ to $\mathbb{Q}\to\mathbb{Q}$ with modulus $\lambda\varepsilon.\frac{\varepsilon}{c}$.*

Proof. Let a_0, a_1: $\mathbb{Q}$ be such that $\boldsymbol{B}^{\mathbb{Q}}_{\frac{\varepsilon}{c}}(a_0,\, a_1)$ holds. This means $c\;|a_0 - a_1| \leq \varepsilon$ holds. Let b: $\mathbb{Q}$ be arbitrary. Because $\left|\underset{\top -c}{\overset{\perp c}{b}}\right| \leq c$ holds, we have that $\left|\underset{\top -c}{\overset{\perp c}{b}}\right| |a_0 - a_1\,| \leq \varepsilon$ holds. Therefore, $\left|a_0\,\underset{\top -c}{\overset{\perp c}{b}} - a_1\,\underset{\top -c}{\overset{\perp c}{b}}\right| \leq \varepsilon$ holds, which is the same as $\boldsymbol{B}^{\mathbb{Q}}_{\varepsilon}\!\left(a_0\,\underset{\top -c}{\overset{\perp c}{b}}, a_1\,\underset{\top -c}{\overset{\perp c}{b}}\right)$. Thus $\boldsymbol{B}^{\mathbb{Q}\to\mathbb{Q}}_{\varepsilon}\!\left(\lambda b.a_0\,\underset{\top -c}{\overset{\perp c}{b}}, \lambda b.a_1\,\underset{\top -c}{\overset{\perp c}{b}}\right)$ holds as required. □

The definition of multiplication for $x, y\colon \mathbb{R}$ is

Definition 11.6. $x\,y := \mathsf{map2}'\!\left(\lambda a.\lambda b.a\,\underset{\top -c}{\overset{\perp c}{b}}\right)(x,y)$, where $c := |y(1)|+1$.

Because multiplication is not uniformly continuous, some closed interval containing y must be found. This requirement is reflected in the fact that c is used in the definition of the modulus of continuity in Theorem 11.5.

This definition of multiplication is asymmetric because only y is bound by a closed interval. The typical definitions of multiplication in exact real arithmetic implementations find closed intervals for both x and y. My definition is the natural definition of multiplication that one gets when one uses monad operations. It saves one from the unnecessary work of finding a closed interval for x. Also notice that when approximating $x\;y$, how well y needs to be approximated depends on the value of the approximation of x. These two considerations mean that one should try to multiply two real numbers in the best order in order to allow the approximation of $x\,y$ to be as efficient as possible.

11.2 Inequalities

Inequality is defined in terms of non-negativity and is analogous to the definition that Bishop and Bridges [Bishop and Bridges, 1985] use.

Definition 11.7. $x \in \mathbb{R}^{0+} := \forall \varepsilon \colon \mathbb{Q}^{+}. -\varepsilon \leq_{\mathbb{Q}} x(\varepsilon)$.

Definition 11.8. $x \leq_{\mathbb{R}} y := y - x \in \mathbb{R}^{0+}$.

Strict inequalities are defined in terms of positivity. The obvious definition of positivity is dual to the definition of non-negativity, $\exists \varepsilon \colon \mathbb{Q}^{+}. \varepsilon <_{\mathbb{Q}} x(\varepsilon)$. This would be analogous to the definition given by Bishop and Bridges [Bishop and Bridges, 1985]. However, I have found an alternative definition to be better.

Definition 11.9. $x \in \mathbb{R}^{+} := \exists \varepsilon \colon \mathbb{Q}^{+}. \hat{\varepsilon} \leq_{\mathbb{R}} x$.

Definition 11.10. $x <_{\mathbb{R}} y := y - x \in \mathbb{R}^{+}$.

To compute the reciprocal (see Section 11.3) or the logarithm (see Section 11.4) of a real number x, one needs a rational number strictly between 0 and x. This is exactly what the witness in my definition provides. To find the same value using the definition from Bishop and Bridges [Bishop and Bridges, 1985], one needs to compute $x(\varepsilon) - \varepsilon$, which is a potentially expensive computation.

11.3 Reciprocal

Not all functions that one wishes to represent are uniformly continuous. One example of a non-uniformly continuous function is the reciprocal function, $\lambda x. x^{-1}$. Bishop and Bridges [Bishop and Bridges, 1985] define a continuous function as a function that is uniformly continuous on every compact interval in its domain. The general idea is, for each x, to find a domain containing x that the function is uniformly continuous on, and clamp the input to that domain.

Theorem 11.11. *For every $a \colon \mathbb{Q}^{+}$, $\lambda x. \left(\underset{\top a}{x} \right)^{-1}$ and $\lambda x. \left(\overset{\bot -a}{x} \right)^{-1}$ are uniformly continuous from $\mathbb{Q}$ to $\mathbb{Q}$, and both functions have a modulus of continuity of $\lambda \varepsilon. \varepsilon\, a^{2}$.*

One can now define x^{-1} for any $x \colon \mathbb{R}$ such that $(x < 0) \vee (0 < x)$.

Definition 11.12. Suppose that $0 < x$. Then there exists some $a \colon \mathbb{Q}$ such that $0 < \hat{a} \leq x$. In this case, define the reciprocal as

$$x^{-1} := \mathsf{map}'\left(\lambda b.\left(b_{\top a}\right)^{-1}\right)(x).$$

Similarly if $x < 0$, then there exists some $a\colon \mathbb{Q}$ such that $x \leq \hat{a} < 0$. In this case, define the reciprocal as

$$x^{-1} := \mathsf{map}'\left(\lambda b.\left(b^{\perp a}\right)^{-1}\right)(x).$$

It is important to note that the result is independent of the choice of a.

Theorem 11.13. *If* $0 < \hat{a_0} \leq x$ *and* $0 < \hat{a_1} \leq x$*, then*

$$\mathsf{map}'\left(\lambda b.\left(b_{\top a_0}\right)^{-1}\right)(x) \asymp \mathsf{map}'\left(\lambda b.\left(b_{\top a_1}\right)^{-1}\right)(x).$$

Notice that the larger the value of a, the larger the modulus of continuity is. The larger the modulus of continuity is, the more efficient the lifted function is, because less work is done when approximating the input. Therefore, it is helpful to find as large an a as is reasonable when proving that x is positive (or negative).

11.4 Power Series

Other elementary functions can be defined in terms of power series. The limit of a convergent sequence can be constructed if the modulus of convergence is known. It is particularly easy to compute the limit of an alternating decreasing series.

Theorem 11.14. *Suppose that* $\lim\limits_{n\to\infty} a_n = 0$*,* a_n *is alternating, and* $\forall\, i.\ |a_{i+1}| < |a_i|$.[11.1] *Let* $x(\varepsilon) := \sum\limits_{\{\, i \,|\, \varepsilon < |a_i| \,\}} a_i$*. Then* x *is a regular function.*

Definition 11.15. Given the conditions of Theorem 11.14, define $\sum_{i=0}^{\infty} a_i := x$.

We can also easily compute the limit of a sub-geometric series.

Theorem 11.16. *Given any* $c\colon \mathbb{Q}$ *such that* $0 \leq c < 1$*, suppose that* a_n *is a series such that* $\forall i.\ |a_{i+1}| \leq c\,|a_i|$*. Let* $x(\varepsilon) := \sum\limits_{\left\{ i \,\middle|\, \varepsilon < \frac{|a_i|}{1-c} \right\}} a_i$*. Then* x *is a regular function.*

[11.1] *The definition of* $\lim\limits_{n\to\infty} a_n = l$ *is* $\forall \varepsilon\colon \mathbb{Q}^{+}.\exists n.\forall m \geq n.\ |a_m - l| \leq \varepsilon$.

Definition 11.17. Given the conditions of Theorem 11.16, define $\sum_{i=0}^{\infty} a_i := x$.

The power series of several elementary functions are alternating and decreasing on small rational inputs.

Definition 11.18.

- For $-1 < a \leq 0$, $\mathsf{exp}_{\mathbb{Q}}(a) := \sum_{i=0}^{\infty} \frac{a^i}{i!}$.
- For $-1 < a < 1$, $\mathsf{sin}_{\mathbb{Q}}(a) := \sum_{i=0}^{\infty} (-1)^i \frac{a^{2i+1}}{(2i+1)!}$.
- For $-1 < a < 1$, $\mathsf{tan}_{\mathbb{Q}}^{-1}(a) := \sum_{i=0}^{\infty} (-1)^i \frac{a^{2i+1}}{2i+1}$.

These functions can be extended to all rationals in their domains by repeated applications of the following formulas to reduce input into the above domains.

$$\begin{aligned}
\mathsf{exp}_{\mathbb{Q}}(a) &\asymp \mathsf{exp}_{\mathbb{Q}}^{2}\Big(\frac{a}{2}\Big) \\
\mathsf{exp}_{\mathbb{Q}}(a) &\asymp \mathsf{exp}_{\mathbb{Q}}^{-1}(-a) \\
\mathsf{sin}_{\mathbb{Q}}(a) &\asymp 3\,\mathsf{sin}_{\mathbb{Q}}\Big(\frac{a}{3}\Big) - 4\,\mathsf{sin}_{\mathbb{Q}}^{3}\Big(\frac{a}{3}\Big) \\
\mathsf{tan}_{\mathbb{Q}}^{-1}(a) &\asymp -\mathsf{tan}_{\mathbb{Q}}^{-1}(-a) \\
\mathsf{tan}_{\mathbb{Q}}^{-1}(a) &\asymp \frac{\pi}{2} - \mathsf{tan}_{\mathbb{Q}}^{-1}(a^{-1}) \qquad \text{(for } 0 < a\text{)} \\
\mathsf{tan}_{\mathbb{Q}}^{-1}(a) &\asymp \frac{\pi}{4} + \mathsf{tan}_{\mathbb{Q}}^{-1}\Big(\frac{a-1}{a+1}\Big) \qquad \text{(for } 0 < a\text{)}
\end{aligned}$$

The real number π can be defined in terms of $\mathsf{tan}_{\mathbb{Q}}^{-1}$ in several ways. Below is a definition that uses particularly small inputs to $\mathsf{tan}_{\mathbb{Q}}^{-1}$ [Williams, 2002].

Definition 11.19.

$\pi := 176\,\mathsf{tan}_{\mathbb{Q}}^{-1}\Big(\frac{1}{57}\Big) + 28\,\mathsf{tan}_{\mathbb{Q}}^{-1}\Big(\frac{1}{239}\Big) - 48\,\mathsf{tan}_{\mathbb{Q}}^{-1}\Big(\frac{1}{682}\Big) + 96\,\mathsf{tan}_{\mathbb{Q}}^{-1}\Big(\frac{1}{12943}\Big)$.

The power series for tanh^{-1} is sub-geometric.

Definition 11.20. For $-1 < a < 1$, $\mathsf{tanh}_{\mathbb{Q}}^{-1}(a) := \sum_{i=0}^{\infty} \frac{a^{2i+1}}{2i+1}$.

The function $\mathsf{cos}_{\mathbb{Q}}$ can be easily defined in terms of $\mathsf{sin}_{\mathbb{Q}}$, and $\mathsf{ln}_{\mathbb{Q}}$ can be defined in terms of $\mathsf{tanh}_{\mathbb{Q}}^{-1}$.

$$\begin{aligned}
\mathsf{cos}_{\mathbb{Q}}(a) &:= 1 - 2\,\mathsf{sin}_{\mathbb{Q}}^{2}\Big(\frac{a}{2}\Big) \\
\mathsf{ln}_{\mathbb{Q}}\Big(\frac{n}{d}\Big) &:= 2\,\mathsf{tanh}_{\mathbb{Q}}^{-1}\Big(\frac{n-d}{n+d}\Big)
\end{aligned}$$

In order to improve efficiency, the input for ln is reduced to the interval $\Big[\frac{1}{2}, 2\Big]$ by repeated applications of the following formula:

$$\mathsf{ln}_{\mathbb{Q}}(a) \asymp \mathsf{ln}_{\mathbb{Q}}\Big(\frac{a}{2}\Big) + \mathsf{ln}_{\mathbb{Q}}(2)$$

Theorem 11.21. *The functions* $\mathsf{sin}_{\mathbb{Q}}$*,* $\mathsf{cos}_{\mathbb{Q}}$*, and* $\mathsf{tan}_{\mathbb{Q}}^{-1}$ *are uniformly continuous on* $\mathbb{Q}$ *with moduli* $\mathbf{I}$.

Theorem 11.22. *For every* $a\colon \mathbb{Z}$*,* $\lambda b\colon \mathbb{Q}.\ \mathsf{exp}_{\mathbb{Q}}\left(\overset{\perp^{a}}{b}\right)$ *is uniformly continuous with modulus* $\lambda\varepsilon.\varepsilon\, 2^a$ *if* $a \leq 0$ *and with modulus* $\lambda\varepsilon.\varepsilon\, 3^a$ *if* $0 \leq a$.

Theorem 11.23. *For every* $a\colon \mathbb{Q}^+$*,* $\lambda b\colon \mathbb{Q}.\ \mathsf{ln}_{\mathbb{Q}}\left(\underset{\top a}{b}\right)$ *is uniformly continuous with modulus* $\lambda\varepsilon.\varepsilon\, a$.

These functions can all be lifted, using bind', to functions on $\mathbb{R}$ or whatever their domains are. The non-uniformly continuous functions exp and ln are defined the same way that multiplication and the reciprocal were defined.

Definition 11.24. $\mathsf{exp}(x) := \mathsf{bind}'\left(\lambda b\colon \mathbb{Q}.\, \mathsf{exp}_{\mathbb{Q}}\left(\overset{\perp^{c}}{b}\right)\right)(x)$ where $c := \lceil x(1) + 1 \rceil$.

Definition 11.25. Suppose that $0 < x$. Then there exists some $a\colon \mathbb{Q}$ such that $0 < \hat{a} \leq x$.

$$\mathsf{ln}(x) := \mathsf{bind}'\left(\lambda b.\, \mathsf{ln}_{\mathbb{Q}}\left(\underset{\top a}{b}\right)\right)(x).$$

Other elementary functions could be defined in terms of the elementary functions given above, but I have not pursued this yet.

11.4.1 Summing Series

One of the more challenging aspects of the formalization was computing the infinite series in an efficient manner. I needed to convince Coq that the procedure of accumulating terms until the error becomes sufficiently small will terminate. To do this, I provided Coq with an upper bound on the number of terms that would be required. I tried two different methods to accomplish this.

The first method computes an upper bound on the number of terms needed as a Peano natural number. The problem is that the call-by-value evaluation scheme used by Coq's virtual machine would first compute this value before computing the series. This upper bound is potentially extremely large, it is encoded in unary, and only a few terms may actually be needed in the computation. My solution to this problem was to create a lazy natural number data type using the standard trick of placing a function from the unit type inside the constructor to delay evaluation [Mitchell, 2003].

```
Inductive LazyNat : Set :=
| LazyO : LazyNat
| LazyS : (unit -> LazyNat) -> LazyNat.
```

Figure 11.1. Inductive definition of lazy natural numbers for Coq.

When a lazy natural number parameter is passed to a function under the call-by-value evaluation scheme, the parameter will be evaluated to weak head normal form. This means that if the number is of the form `LazyS` *`f`*, then the *`f`* term will be reduced to `fun () =>` *`m`*, but the body *`m`* will not be evaluated. If *`m`* has a very large normal form, then this delay will prevent the term from being expanded. Only when *`f`* is forced, by evaluating *`f`* `()`, will the body be evaluated to the next constructor. If *`f`* `()` is never evaluated, then the work of fully evaluating the number is avoided.

By using this delaying and forcing technique, only the number of constructors needed for the recursion are evaluated. Care is needed to only force evaluation when necessary in order to avoid accidentally normalizing the entire (extremely large) lazy natural number.

A second method, suggested by Benjamin Grégoire [Grégoire, 2007], is to compute the number of terms needed as a binary number. This prevents the term from becoming too big. It is possible to do recursion over the binary natural numbers such that two recursive calls are made with the output of one recursive call being threaded through the other. In this way, up to 2^n recursive calls can be made even though only n constructors are provided by the witness of termination.

In the simplified example below, think of `F` as a function that we would take the fixed point of if we had general recursion available to us. The function `iterate_pos n` is `n` iterations of `F`. Continuation passing style is used to thread the recursive calls.

```
Variable A R : Type.
Variable F : (A -> R) -> A -> R.

Fixpoint iterate_pos (n:positive) (cont: A -> R) : A -> R :=
match n with
| xH => F cont
| xO n’ => iterate_pos n’ (fun a => iterate_pos n’ cont a)
| xI n’ => F (fun a => (iterate_pos n’
                          (fun a => iterate_pos n’ cont a)) a)
end.
```

Figure 11.2. The Coq function `iterate_pos` iterates `F` a given number of times.

The η-expansions of the continuations in the above definition are important, otherwise the virtual machine would compute the value of the `iterate_pos n’ cont` calls before reducing `F`. This is important because the idea is that `F rec a` may not utilize its recursive call, `rec`, depending on the value of `a`.

If Coq's virtual machine supported lazy evaluation, none of these tricks would be needed. A standard Peano number could be used as the witness of termination, and it would only be evaluated as far as the number of recursive calls need.

11.4.2 π

A common definition of π is $4\,\mathtt{tan}^{-1}(1)$. This is an inefficient way of computing π because the series for $\mathtt{tan}^{-1}(1)$ converges very slowly. My definition of π (Definition 11.19) can easily be shown to be equivalent to $4\ \mathtt{tan}^{-1}(1)$ by repeated application of the *arctangent sum law*:

$$\text{if } a, b \in]-1, 1[\text{ then } \mathtt{tan}^{-1}(a) + \mathtt{tan}^{-1}(b) \asymp \mathtt{tan}^{-1}\left(\frac{a+b}{1-a\,b}\right)$$

To apply the arctangent sum law, one needs to repeatedly verify that a and b lie in $]-1, 1[$. To prove this, I wrote a Coq function to iterate the function $f(b) := \frac{a+b}{1-a\,b}$, and at each step verify that the result is in the interval $]-1,\ 1[$. This function, called `ArcTan_multiple`, has type

$$\forall a \colon \mathbb{Q}. -1 < a < 1 \Rightarrow \forall n. \top \vee \left(n\,\mathtt{tan}^{-1}(x) \asymp \mathtt{tan}^{-1}\left(f^{(n)}(0)\right)\right).$$

It is easy to build a function of the above type that just proves $\top$ in all cases, but `ArcTan_multiple` tries to prove the non-trivial result if it can.

To apply this lemma, I use a technique called reflection [Barendregt and Geuvers, 2001]. The idea is to evaluate the `ArcTan_multiple`(a, r, n) into head normal form. This will yield either `left`(q) or `right`(p). If `right`(p) is returned then p is the proof we want.

My first attempt at building a tactic to implement this did not work well. I used Coq's `eval hnf` command to reduce my expression to head normal form. However, this command repeatedly calls `simpl` to expose a constructor instead of using the evaluation mechanism directly. The problem was that `simpl` does extra reductions that are not necessary to get head normal form, and using `eval hnf` was too time consuming.

Instead, I built a generic reflection lemma, called `reflect_right`, to assist in applying the `ArcTan_multiple` function:

$$\forall z \colon A \vee B. (\text{if } z \text{ then } \bot \text{ else } \top) \Rightarrow B$$

This simple lemma does case analysis on z. If z contains a proof of A, it returns a proof of $\bot \Rightarrow B$. If z contains a proof of B, it returns a proof of $\top \Rightarrow B$. To prove $n\,\mathtt{tan}^{-1}(a) \asymp \mathtt{tan}^{-1}(f^{(n)}(0))$ for the example where $a := \frac{1}{57}$ and $n := 176$, one applies `reflect_right` composed with `ArcTan_multiple` to reduce the goal to

$$\mathtt{if\ (ArcTan\_multiple}\ \frac{1}{57}\ *\ \mathtt{176)\ then}\ \bot\ \mathtt{else}\ \top,$$

where $*$ is the trivial proof of $-1 < \frac{1}{57} < 1$. One then normalizes this expression to either $\top$, if `ArcTan_multiple` succeeds in finding a proof, or $\bot$, if it fails.

11.5 Improving Efficiency

11.5.1 Compression

Without intervention, the numerators and denominators of rational numbers occurring in real number computations become too large for practical computation. To help prevent this, I defined a compression operation for real numbers:

$$\mathsf{compress}(x)(\varepsilon) := \mathsf{approx}_{\mathbb{Q}}\left(x\left(\frac{1}{d}\right), d\right)$$

where $d\colon \mathbb{Z}$ is the smallest integer such that $\frac{1}{d} < \frac{\varepsilon}{2}$ and $\mathsf{approx}_{\mathbb{Q}}(a, d)$ returns some rational number within $\frac{1}{d}$ of a. The idea is that $\mathsf{approx}_{\mathbb{Q}}(a, d)$ quickly computes a rational number close to a but having a smaller numerator and denominator. In my implementation, I return $\frac{n}{d}$, where n is chosen appropriately so that $\frac{n}{d}$ is within $\frac{1}{d}$ of a.

In any rational interval $[p, q]$ there is a unique simplest rational number $\frac{n}{d}$ where simplicity is measured by $|n| + |d|$. However, finding the simplest rational number is roughly equivalent to a GCD computation. This is too expensive to compute, so I use the faster approximation method given above.

The $\mathsf{compress}$ function is equivalent to the identity function on $\mathbb{R}$.

Theorem 11.26. *For any $x\colon \mathbb{R}$, $\mathsf{compress}(x) \asymp x$.*

Generous use of $\mathsf{compress}$ usually increases the efficiency of real number computation. I recommend that you pass every function's input through $\mathsf{compress}$ first unless you have reason not to.

There is one extra property that I will require from $\mathsf{approx}_{\mathbb{Q}}$ in Section 11.5.2. The approximation of a rational number will never drop below an integer.

Theorem 11.27. *For any $z\colon \mathbb{Z}$ and $a\colon \mathbb{Q}$, if $z \leq a$ then $\forall d. z \leq \mathsf{approx}_{\mathbb{Q}}(a, d)$.*

11.5.2 Square Root

Although one can define $\sqrt{x}$ as $\mathsf{exp}\left(\frac{1}{2}\,\mathsf{ln}(x)\right)$, the square root function is common enough to warrant its own more efficient definition. If $a\colon \mathbb{Q}$ is such that $1 \leq a < 4$, one can compute $\sqrt{a}$ by taking the limit of Newton approximations. Let the first approximation be $b_0 := \frac{a+1}{2}$, and define successive approximations by $b_{i+1} := \mathsf{approx}_{\mathbb{Q}}(\frac{a + b_i^2}{2\,b_i}, 2^{2^{i+1}+1})$. The first approximation has an error of at most $\frac{1}{2}$. Each successive approximation squares the error.

Theorem 11.28. *For any $a\colon \mathbb{Q}$, such that $1 \le a < 4$, let $x(\varepsilon) := b_n$, where n is the first natural number such that $2^{-2^n} \le \varepsilon$. Then x is a regular function.*

Proof. Firstly, it suffices to show that for all n, $\left(b_n - 2^{-2^n}\right)^2 \le a \le \left(b_n + 2^{-2^n}\right)^2$ and $1 \le b_n$ hold. To see this, suppose $\left(b_n - 2^{-2^n}\right)^2 \le a \le \left(b_n + 2^{-2^n}\right)^2$ and $\left(b_m - 2^{-2^m}\right)^2 \le a \le \left(b_m + 2^{-2^m}\right)^2$ hold as well as $1 \le b_n$ and $1 \le b_m$. From this we see that $0 \le \left(b_m + 2^{-2^m}\right)^2 - \left(b_n - 2^{-2^n}\right)^2$ holds. We have that $1 \le b_m + 2^{-2^m} + b_n - 2^{-2^n}$ holds. Therefore, $0 \le b_m + 2^{-2^m} - b_n + 2^{-2^n}$ and hence $b_n - b_m \le 2^{-2^n} + 2^{-2^m}$ holds. Similarly, $b_m - b_n \le 2^{-2^m} + 2^{-2^n}$ holds. Together this means $\boldsymbol{B}^{\mathbb{Q}}_{2^{-2^n}+2^{-2^m}}(b_n, b_m)$ holds.

Consider the case where $n = 0$. In this case $\left(b_0 - 2^{-1}\right)^2 = \frac{a^2}{4}$. Because $a < 4$, we can conclude that $\frac{a^2}{4} \le a$. We also have that $\left(b_0 + 2^{-1}\right)^2 = \frac{a^2}{4} + a + 1$, and clearly $a \le \frac{a^2}{4} + a + 1$. Finally note that $1 \le a$, so $1 \le b_0$.

Suppose that $n = m + 1$. Let $\delta := 2^{-2^m}$ and $c := \frac{a + b_m^2}{2\, b_m}$. Assume by induction that $(b_m - \delta)^2 \le a \le (b_m + \delta)^2$ and $1 \le b_m$. We know $0 \le \left(b_m^2 - a\right)^2$ holds and hence $0 \le \left(b_m^2 + a\right)^2 - 4\, a\, b_m^2$ holds. From this we see that $a\,(2\, b_m)^2 \le \left(a + b_m^2\right)^2$ holds. Therefore $a \le c^2$ and hence $a \le \left(c + \frac{\delta^2}{2}\right)^2$ hold. Because $1 \le b_m$, we know that $a + b_m^2 - b_m\, \delta^2 \le a + b_m^2 - \delta^2$. Because $\delta^2 \le 1 \le b_m$, we have $0 \le a + b_m \left(b_m - \delta^2\right)$ and hence $\left(a + b_m^2 - b_m\, \delta^2\right)^2 \le \left(a + b_m^2 - \delta^2\right)^2$ holds. We also know that $0 \le \left(a - (b_m - \delta)^2\right)\left((b_m + \delta)^2 - a\right)$, so $\left(a + b_m^2 - \delta^2\right)^2 \le a\ \ (2\, b_m)^2$ holds. Together we have $\left(a + b_m^2 - b_m\, \delta^2\right)^2 \le a\,(2\, b_m)^2$ and hence $\left(c - \frac{\delta^2}{2}\right)^2 \le a$ holds.

Now we need to show that $1 \le b_{m+1}$ holds. We know $1 \le c$ because $1 \le a$. Therefore, by Theorem 11.27, $1 \le b_{m+1}$.

Because $\boldsymbol{B}^{\mathbb{Q}}_{\frac{\delta^2}{2}}(c, b_{m+1})$ holds and both $1 \le c$ and $1 \le b_{m+1}$ hold, we can can conclude that $\left(b_{m+1} - \delta^2\right)^2 \le a \le \left(b_{m+1} + \delta^2\right)^2$ holds from the fact $\left(c - \delta^2\right)^2 \le a \le \left(c + \delta^2\right)^2$ holds as shown above. □

For $0 < a$ that falls outside the interval $[1,\ 4[$, one can find some $m\colon \mathbb{Z}$ such that $1 \le 4^m\, a < 4$. Therefore, one can define the square root function on the rationals as

$$\begin{array}{rcll} \sqrt{0} & := & 0 & \\ \text{for } 1 \le a < 4,\ \sqrt{a} & := & x & \text{(where } x \text{ is defined as in Theorem 11.28)} \\ \text{otherwise, } \sqrt{a} & := & \dfrac{\sqrt{4^m\, a}}{2^m} & \text{(for the } m\colon \mathbb{Z} \text{ such that } 1 \le 4^m\, a < 4\text{).} \end{array}$$

Theorem 11.29. *The function $\lambda a.\ \sqrt{a_{\top 0}}$ is uniformly continuous with modulus $\lambda\varepsilon.\varepsilon^2$.*

By using bind′, the square root function can be lifted to the domain $\mathbb{R}$. Notice that the square root of any negative number will be 0. If one wishes, one could add a dependent parameter to the square root that requires a proof that the input is non-negative (see Definition 11.7). Unlike a proof of positivity, a proof of non-negativity contains no constructive information, so a dependent parameter is not required to compute the square root.

11.5.3 More Efficient Polynomials

Consider the problem of computing $4\,x\,(1-x)$ for an arbitrary real number x. The variable x occurs twice in the expression, so two different approximations for x will be computed (in general) to two different precisions. There is potential for some savings here. One could instead approximate x only once, to the higher degree of accuracy, and use the higher degree of accuracy for both approximations of x. If approximating x is an expensive operation, and x occurs several times, then one could potentially save a lot of work.

The above situation occurs whenever a subexpression occurs multiple times in a given expression. The solution seems as simple as computing the highest precision required for all instances of the subexpression. Unfortunately, it is not always straightforward to determine what the highest degree of precision required is. According to my definition of multiplication, when multiplying x by $(1 - x)$ the degree of precision required for $(1 - x)$ depends on the value of the approximation of x in the first term. This sequencing of approximations makes it impossible, in general, to find the highest precision requested for x in a given expression.

Multivariable polynomials form a common class of expressions where variables occur multiple times. This class of expressions can be optimized so that each variable is only approximated once. Let $p\colon \mathbb{Q}[X_0][X_1]...[X_{n-1}]$ be a multivariable polynomial in n variables.[11.2] I use the ellipsis instead of writing out the full recursive definition of the type of multivariable polynomials. One can recursively define an interpretation of $[\![p]\!]_n$ as a uniformly continuous function $\mathbb{Q} \to ... \to \mathbb{Q} \to \mathbb{Q} \to \mathbb{Q}$ that agrees with the evaluation of $\lambda a_{n-1}...\lambda a_0.\, p(a_{n-1})...(a_0)$ on the unit hyperinterval. The case when $n = 0$ is trivial: $[\![p]\!]_0 \;:=\; p$. Suppose that $[\![\,\cdot\,]\!]_m$ has been defined and consider the case when $n = m + 1$. Let M be an upper bound of $\lambda x_m....\lambda x_0.\, |p'(x_m)...(x_0)|$ over the unit hyperinterval, where p' is the formal derivative of p with respect to X_m. The interpretation for the multivariable polynomial p is $[\![p]\!]_{m+1} \;:=\; \lambda a\colon \mathbb{Q}.\, \left[\!\!\left[p\begin{pmatrix} \perp 1 \\ a \\ \top 0 \end{pmatrix} \right]\!\!\right]_m$ with a modulus of continuity of $\lambda\varepsilon.\, \frac{\varepsilon}{M}$. Once defined, the uniformly continuous function can be lifted, using map and ap, in order to apply a multivariable polynomial to real numbers.

[11.2] Formally this is really $\mathbb{Q}[X][X]...[X]$, because $R[X]$ is the ring of formal polynomials over R for any ring R. The different formal variables are accessed by using a differing number of injections. X is the outermost variable, $\iota(X)$ is the second most outer variable, $\iota(\iota(X))$ is the third outermost variable, where $\iota\colon R \Rightarrow R[X]$ is the natural ring monomorphism. For this informal discussion I will stick with the traditional mathematics notation.

An upper bound on the absolute value of a multivariable polynomial can be computed in a number of ways. One reasonable method is to express the multivariable polynomial in the Bernstein basis. The maximum and minimum of the coefficients in the Bernstein basis form reasonable upper and lower bounds of the polynomial [Zumkeller, 2008]. This is the method that I use.

Each variable is only approximated once. The bound on the derivative of p is used to compute the modulus of continuity for that particular variable. It is interesting to note that after a variable is approximated, it is plugged into the multivariable polynomial, and the number of variables is reduced by one. This means the degree of approximation required for successive variables depends on the approximation of the previous variables. The amount of work required to evaluate a multivariable polynomial can therefore depend on the order in which the variables appear.

11.5.3.1 Implementing More Efficient Polynomials

Implementing the above interpretation in Coq was more difficult than I had expected. Obviously the interpretation is defined recursively. The difficulty comes when trying to prove that $\lambda a\colon \mathbb{Q}.\left[\!\!\left[p\left(\overset{\bot 1}{\underset{\top 0}{a}}\right)\right]\!\!\right]_m$ is uniformly continuous. This requires supplying a proof of

$$\forall a b \varepsilon.\, \boldsymbol{B}^{\mathbb{Q}}_{\frac{\varepsilon}{M}}(a,b) \Rightarrow \boldsymbol{B}^{\mathbb{Q}\to\ldots\to\mathbb{Q}}_{\varepsilon}\left(\left[\!\!\left[p\left(\overset{\bot 1}{\underset{\top 0}{a}}\right)\right]\!\!\right]_m, \left[\!\!\left[p\left(\overset{\bot 1}{\underset{\top 0}{b}}\right)\right]\!\!\right]_m\right). \tag{11.1}$$

The difficulty is that such a proof requires reasoning about $\left[\!\!\left[p\left(\overset{\bot 1}{\underset{\top 0}{a}}\right)\right]\!\!\right]_m$, but one cannot prove anything about $[\![\,\cdot\,]\!]_m$ because it is a recursive call. All that is known about $[\![\,\cdot\,]\!]_m$ is its type.

I found no other way than to recursively define an interpretation function of type

$$\forall p\colon \mathbb{Q}[X_0][X_1]\ldots[X_{n-1}].\,\exists f\colon \mathbb{Q}\to\ldots\to\mathbb{Q}.\,\chi_n(p,f),$$

where χ_n is itself defined recursively as

$$\begin{aligned} \chi_0(p,f) &:= p = f \\ \chi_{m+1}(p,f) &:= \forall a\colon \mathbb{Q}.\,\chi_m\left(p\left(\overset{\bot 1}{\underset{\top 0}{a}}\right), f(a)\right). \end{aligned}$$

An interpretation function of this type produces a uniformly continuous function f of type $\mathbb{Q}\to\ldots\to\mathbb{Q}$ along with a proof that this function is equivalent to the input multivariable polynomial p on the unit hyperinterval. This extra information in the type gives enough information about f so that I can prove Equation 11.1 during the recursive definition of the interpretation function.

11.5.4 More Efficient Power Series

A power series converges faster for arguments closer to 0. For $\mathsf{sin}_{\mathbb{Q}}$ and $\mathsf{exp}_{\mathbb{Q}}$, the two equations mentioned before,

$$\begin{aligned}\mathsf{exp}_{\mathbb{Q}}(a) &\asymp \mathsf{exp}^2_{\mathbb{Q}}\Big(\frac{a}{2}\Big)\\ \mathsf{sin}_{\mathbb{Q}}(a) &\asymp 3\,\mathsf{sin}_{\mathbb{Q}}\Big(\frac{a}{3}\Big) - 4\,\mathsf{sin}^3_{\mathbb{Q}}\Big(\frac{a}{3}\Big),\end{aligned}$$

can be repeatedly applied to shrink the input arbitrarily close to 0. There is, of course, a trade-off between evaluating a polynomial each time such an equation is used and having the power series converge faster. The optimal trade-off depends on the input and the implementation of the library. At the moment, I do not apply any of this extra range reduction.

Unfortunately such nice reductions do not seem to exist for ln or tan^{-1}.

11.5.5 More Efficient Periodic Functions

The functions sin and cos are both periodic with period 2π. With a reasonably fast implementation of π, it is possible to subtract a multiple of 2π to reduce the magnitude of the argument to sin or cos.

$$\begin{aligned}\forall n{:}\,\mathbb{Z}.\,\mathsf{sin}(x) &\asymp \mathsf{sin}(x - n\,2\pi)\\ \forall n{:}\,\mathbb{Z}.\,\mathsf{cos}(x) &\asymp \mathsf{cos}(x - n\,2\pi)\end{aligned}$$

The best value of n would be $\left\lfloor \frac{x}{2\pi} \right\rceil$. The function mapping x to a nearest integer $\lfloor x \rceil$ is not computable for real numbers, but it is computable for the rationals (it does not matter which integer $\frac{1}{2}$ maps to). The approximation $\frac{x}{2\pi}(\frac{1}{2})$ is within $\frac{1}{2}$ of $\frac{x}{2\pi}$, so $\left\lfloor \frac{x}{2\pi}(\frac{1}{2}) \right\rceil$ is an integer within 1 of $\left\lfloor \frac{x}{2\pi} \right\rceil$, which is good enough to use for n.

One could be even more clever with the symmetries of sin and cos to reduce the range further [Lester, 2008], but I have only implemented the reductions up to 2π described above.

11.5.6 Summing Lists

It is not uncommon to want to sum a list of real numbers. The use of ap in the definition of addition means that to compute $x + y$ within ε, one must compute both x and y within $\frac{\varepsilon}{2}$. If one associates the sum of a list of numbers all to one side $a_0 + (a_1 + (\ldots + a_n))$, the first term a_0 is only approximated within $\frac{\varepsilon}{2}$, while the last term a_n is approximated within $\frac{\varepsilon}{2^n}$.

If one is summing a list with exactly a power of 2 number of terms, then one can associate the sum into a balanced binary tree. In this case, all terms will be approximated to the same degree; however, to handle more general cases it is useful to directly define the sum of n real numbers.

Definition 11.30. $\sum_{i=0}^{n} x_i := \lambda\varepsilon.\ \sum_{i=0}^{n} x_i(\frac{\varepsilon}{n})$.

Theorem 11.31. *The term* $\sum_{i=0}^{n} x_i$ *is a regular function.*

11.6 Correctness

There are two ways to prove that my functions are correct. One way is to prove that they satisfy some uniquely defining properties. The other way is to prove that each function is equivalent to a given reference implementation. I have verified that my functions are equivalent to the corresponding functions defined in the CoRN library [Cruz-Filipe et al., 2004]. The functions in the CoRN library can be seen to be correct from the large library of theorems available about them. The CoRN library contains many different characterizations of these functions and new characterizations can easily be developed.

The CoRN library defines a *real number structure* as a complete, ordered, Archimedean, constructive field. My first step was to prove that my operations form a real number structure. I first attempted to directly show that my real numbers satisfy all the axioms of a real number structure, but this approach was difficult. Instead, I created an isomorphism between my real numbers and an existing model of the real numbers developed by Niqui [Geuvers and Niqui, 2002]. This was a much simpler approach because Niqui's Cauchy sequence definition and my regular function definition are closely related. With this isomorphism in place, I proved my operations satisfied the axioms of a real number structure by passing through the isomorphism and using Niqui's existing lemmas. Niqui has also proved that all real number structures are isomorphic, so I can now create an isomorphism between my real numbers and any other real number structure.

The next step was to prove that my elementary functions are equivalent to the corresponding CoRN functions. These theorems are of the form $\Phi(f_{\mathrm{CoRN}}(x)) \asymp f(\Phi(x))$ where Φ is the isomorphism from CoRN's real numbers to my real numbers.

To aid in converting statements between different representations of real numbers, I have created a Coq rewrite database that contains the correctness lemmas (see Section 2.2.8.1). By rewriting with this database, expressions can be automatically converted from CoRN's functions into my functions. This database can easily be extended with more functions as they are developed.

The CoRN library was more than just a specification; the library was useful throughout my development. For example, I was often able to prove that a differentiable function f is uniformly continuous with modulus $\lambda\varepsilon.\ \frac{\varepsilon}{M}$ when M is a bound on the derivative of f. I could prove this because the theory of derivatives had already been developed in CoRN, and the isomorphisms allowed me to carry these results over to my real numbers. The CoRN library also helped me reduce the problem of proving the correctness of continuous functions on $\mathbb{R}$ to proving correctness only on $\mathbb{Q}$.

Theorem 11.32. *Consider $f\colon \mathbb{Q} \Rightarrow \mathbb{R}$. If there is a CoRN differentiable function g with derivative g' such that $\forall a\colon \mathbb{Q}.\, \Phi(g(\hat{a})) \asymp f(\Phi(\hat{a}))$ and $\forall x.\, |g'(x)| \leq \hat{M}$, then f is uniformly continuous with modulus $\lambda\varepsilon.\, \frac{\varepsilon}{M}$.*

Theorem 11.33. *If f is uniformly continuous and g is CoRN continuous and $\forall a\colon \mathbb{Q}.\, \Phi(g(\hat{a})) \asymp f(\Phi(\hat{a}))$, then $\forall x\colon \mathbb{R}.\, \Phi(g(x)) \asymp f(\Phi(x))$.*

11.7 Timings

Table 11.1 shows examples of real number expressions that can be approximated. Approximations of these expressions were evaluated to within 10^{-20} on a 1.4 GHz ThinkPad X40 laptop using Coq's `vm_compute` command for computing with its virtual machine [Grégoire and Leroy, 2002]. These examples are taken from the "Many Digits" friendly competition problem set [Niqui and Wiedijk, 2005].

Coq Expression			
Mathematical Expression	**Time**	**Result**	**Error**
`(CRsqrt (compress (rational_exp (1))* compress (CRinv_pos (3#1) CRpi)))%CR`			
$\sqrt{\frac{e}{\pi}}$	1 sec	0.93019136710263285866	10^{-20}
`(sin (compress (CRpower_positive 3 (translate (1#1) (compress (rational_exp (1)))))))%CR`			
$\sin((e+1)^3)$	25 sec	0.90949524105726624718	10^{-20}
`(exp (compress (exp (compress (rational_exp (1#2))))))%CR`			
$e^{e^{e^{\frac{1}{2}}}}$	146 sec	181.33130360854569351505	10^{-20}

Table 11.1. Timings of approximations of various real number expressions.

Keep in mind that these computations are carried out without using machine integer support. Everything is done with inductive and co-inductive data types.

11.8 Solving Strict Inequalities Automatically

Whether a strict inequality holds between real numbers is semi-decidable. This question can be reduced to proving that some expression $e_0\colon \mathbb{R}$ is positive. To prove e_0 is positive one must find an $\varepsilon\colon \mathbb{Q}^+$, such that $\hat{\varepsilon} \leq e_0$. I wrote a tactic to automate the search for such a witness. It starts with an initial $\delta\colon \mathbb{Q}^+$, and computes to see if $e_0(\delta) - \delta$ is positive. If it is positive, then $e_0(\delta) - \delta$ is such a witness; otherwise δ is halved and the process is repeated. If $e_0 \asymp 0$, then this process will never terminate. If $e_0 < 0$, then the tactic will notice that $e_0(\delta) + \delta$ is negative and terminate with an error indicating that e_0 is negative. I used this tactic when I needed to prove $2 \leq e \leq 3$ during my development.

This tactic has been combined with the rewrite database of correctness lemmas to produce a tactic that solves strict inequalities of closed expressions over CoRN's real numbers. An `autorewrite` tactic is applied first, which automatically transforms many CoRN functions into equivalent versions of my functions (see Section 2.2.8.1 and Section 11.6). If this completes successfully, then it runs my tactic above that solves inequalities by computation. This allows users to work entirely with CoRN's real numbers from their perspective. They need never be aware that my effective real numbers are running behind the scenes.

Recently Cezary Kaliszyk has proved that Coq's classical real numbers (from Coq's standard library) form a CoRN real number structure [Kaliszyk and O'Connor, 2009], and he has shown that Coq's elementary functions are equivalent to CoRN's. Now strict inequalities composed from elementary functions over Coq's classical real numbers can automatically be solved.

The tactic currently only works for expressions composed from total functions. Partial functions with open domains pose a problem because they need constructive proof objects. Consider the case of computing $\mathsf{ln}(x)$. In reality ln has type $\forall x{:}\,\mathbb{R}.\, 0 < x \Rightarrow \mathbb{R}$, and the proof of $0 < x$ is a required second parameter. Recall from Definition 11.10, that $0 < x$ contains constructive information. An actual witness needs to be extracted from this proof in order to proceed with the computation. However, proof objects for CoRN functions are opaque, meaning that witnesses cannot be used for computation inside Coq, and Coq's classical functions have no proof objects at all. Thus the required transparent proof object cannot automatically be translated from CoRN functions nor from Coq's classical functions. However, the required proof objects are proofs of strict inequalities, so I would like to develop a tactic that recursively solves these strict inequalities and creates transparent proof objects. This will allow one to prove strict inequalities over expressions with partial functions such as ln. However, such a tactic seems to require setoid rewriting inside dependent types. This cannot be done with Coq's current setoid rewriting mechanism.

11.9 Remarks

11.9.1 Using this Development

I envision that the CoRN real numbers will continue to be the primary structure used in developments. The CoRN real numbers already come with a wide variety of theorems and definitions about them. I expect my real number library to work behind the scenes to solve problems. For those cases where real numbers computation is needed, I expect the problem to be translated from the CoRN real numbers to my real numbers. The problem would then be solved by computation, and then converted back to CoRN real numbers if necessary. Hopefully, this process can be automated most of the time, as was done in Section 11.8, so that the user does not even need to be aware of my real number library.

11.9.2 Reworking Power Series

The most significant issue with my implementation is the way that I implement power series (Section 11.4). The problem is that the partial series is summed using exact rational arithmetic. These sums create rational numbers with large numerators and denominators. In future work, one should explore the possibility of using real number arithmetic to compute the partial sums. Even though these sums are entirely rational, the ability for real numbers arithmetic to round off intermediate results (see compress in Section 11.5.1) could improve the overall efficiency of these calculations.

11.9.3 Regular Functions vs Cauchy Sequences

I developed regular functions as the most flexible type that I imagined for real numbers. I chose this representation because it is the type that reflects how real numbers are consumed: real numbers are consumed by asking for an approximation. On the other side, Cauchy sequences reflect how real numbers are created. For example, the real numbers defined by power series (see Section 11.4) can naturally be seen as Cauchy sequences.

Now that the implementation is complete, it is worth reflecting on the actual types behind the real numbers. Real numbers are generated in one of two ways. They are either created from a rational number using unit, or they are generated by the limit of a power series, which is effectively a Cauchy sequence. The arithmetic operations on real numbers simply combine and manipulate these Cauchy sequences and their associated moduli of convergences.

The only exceptional operation is compress. In my implementation, compress effectively turns a real number into a regular sequence, which is a special kind of Cauchy sequence. Other potential implementations of compress could be implemented differently, and may not produce Cauchy sequences.

11.9.4 Haskell Prototype

Before beginning this formalization effort, I wrote a prototype Haskell module that implemented the functions described in this chapter. I wanted to ensure that the algorithms I had in mind would be practical for evaluation. This code, entitled *Few Digits* [O'Connor, 2005b], competed in the *"Many Digits" Friendly Competition* [Niqui and Wiedijk, 2005] occurring on October 3-4, 2005. Although the code did not do particularly well in the contest, finishing eighth out of nine, it still could compute thousands of decimal digits within minutes or seconds for many of the problems. This performance was satisfactory, and I proceeded with the Coq formalization.

The formalization effort uncovered a few errors in my prototype. These errors were all of the form of forgetting to clamp the input to some interval before function application (see Theorem 11.11 for example). Of course, these errors do not occur in the Coq formalization.

11.9.5 Comparisons With Other Implementations

Cruz-Filipe implemented CoRN's library of theorems and functions over the real numbers in Coq [Cruz-Filipe, 2004]. He worked with an abstract axiomatization of the real numbers, so his constructions can only be executed when given a constructive model of these axioms. Niqui implemented one model of the constructive real numbers using a Cauchy sequence representation [Geuvers and Niqui, 2002]. Although Cruz-Filipe's work is constructive, it was never designed for evaluation [Cruz-Filipe and Letouzey, 2006]. Many important definitions are opaque, so evaluation cannot be done internally in Coq. Efficiency of computation was not a concern during development, so even when programs are extracted from the development, computation is slow. Cruz-Filipe showed that it is practical to develop a constructive theory of real analysis inside Coq. My work extends this result to show that it is possible to develop a theory of real analysis that is also practical to evaluate. Cruz-Filipe's work also forms the reference specification of my work (see Section 11.6).

There have been several implementations of real numbers in Coq using a co-inductive stream of digits representation. Ciaffaglione developed ring operations using a signed binary digits representation [Ciaffaglione, 2003], and Bertot uses a similar representation to compute Euler's constant from its series definition [Bertot, 2007]. Most recently, Julien extended this work by implementing a stream of signed digits representation for an arbitrary base [Julien, 2008]. These representations allow common subexpressions to be easily shared because streams naturally memoize. Sharing does not work as well with my representation because real numbers are represented by functions. Julien also uses the new machine integers implementation in Coq's virtual machine to make his computations even faster. It remains to be seen if using machine integers to implement rational number operations would provide a similar boost to my implementation.

Muñoz and Lester implemented a system for approximating real number expressions in PVS [Muñoz and Lester, 2005]. Their system uses rational interval analysis for doing computation on monotone segments of transcendental functions. Unfortunately, this leads to some difficulties when reasoning at a local minimum or maximum, so their system cannot automatically prove $0<\sin\left(\frac{\pi}{2}\right)$, for instance.

Lester created a second implementation in PVS using fast Cauchy sequences [Lester, 2008]. The nth approximation in a fast Cauchy has an error of at most $\frac{1}{2^n}$. Lester defined several elementary functions. Because PVS works with classical mathematics, there is no internal evaluation mechanism like Coq has. Instead, the meta-system, in this case LISP, is used to create deductions that provide arbitrarily tight rational bounds on real number expressions.

Harrison implemented a system to approximate real number expressions in HOL Light using a representation similar to regular sequences [Harrison, 1998b]. Like Lester's work, his system runs a tactic that externally computes an approximation to an expression and generates a proof that the approximation is correct.

Jones created a preliminary implementation of real numbers and complete metric spaces in LEGO [Jones, 1993]. She represented real numbers as a collection containing arbitrarily small intervals of rational numbers that all intersect. Complete metric spaces were similarly represented by using balls in place of intervals. Because the only way of getting an interval from the collection is by using the arbitrarily small interval property, her representation could have been simplified by removing the collection and letting it implicitly be the image of a function that produces arbitrarily small intervals. This is similar to my work because one can interpret a regular function f as producing the interval $[f(\varepsilon) - \varepsilon, f(\varepsilon) + \varepsilon]$. Perhaps using functions that return intervals could improve computation by allowing one to see that an approximation may be more accurate than requested.

My work is largely based on Bishop and Bridges's work [Bishop and Bridges, 1985]. Some definitions have been modified to make the resulting functions more efficient. My definition of a metric space is more general; it does not require that the distance function be computable. The original motivation for the distance relation was only to develop a theory of metric spaces that did not presuppose the existence of the real numbers; however, it allowed me to form a metric space of functions (see Section 10.2). This metric space will not, in general, have a computable distance function and would not be a metric space according to Bishop and Bridges's definition.

11.9.5.1 Comparison with Classical Reals

The constructive reals are a different structure than the classical reals. One way of defining the classical reals is by starting with classical sequences of rationals. Recall that a classical sequence is a binary relation on $\mathbb{N}$ and $\mathbb{Q}$ satisfying the property given in Section 2.2.2. Classical reals can be defined as classical Cauchy classical sequences of rationals with the appropriate equivalence relation, where the classical Cauchy condition uses a classical existential quantifier.

Not surprisingly, one cannot, in general, compute approximations of classical real numbers like one can with constructive real numbers. There is an injection from the constructive real numbers to the classical real numbers, but the inverse of this function cannot be constructed.

All of the developments of real numbers above are developments of the classical real numbers except for Cruz-Filipe and Jones's developments. Even Ciaffaglione, Bertot, and Julien's work is more or less classical, even though they are using Coq. Their implementation is constructive, but they use classical real numbers to specify the correctness of their algorithms. This makes their work more like the other classical approaches.

Rather than using a model of the classical real numbers to verify algorithms, I find it better to work with constructive reals which is a theory of the algorithms themselves. The theory of the constructive reals is compatible with the theory of the classical reals, and working with the constructive reals helps prevent one from being distracted by potentially uncomputable real numbers.

Chapter 12
Integration over $\mathbb{R}$

In this chapter we develop another example of a complete metric space: the space of integrable functions. I define the integrable functions over [0,1] as the completion of the space of formal rational step functions on [0,1] with the L^1 metric. Integration is defined as a uniformly continuous function from rational step functions to the rationals. This uniformly continuous function is then lifted to define integration from the integrable functions to the real numbers.

The formal step functions also form a monad (Section 12.1.2) and the completion monad distributes over this step function monad (Section 12.1.9). However, rather than working with the monad interface for step functions, we find the applicative functor interface easier to work with (Section 12.1.3).

To every uniformly continuous function, we associate an integrable function. This allows us to integrate any uniformly continuous function over [0,1] (Section 12.1.7). Because this is all developed as constructive mathematics, and because we use my efficient real numbers from Chapter 11, this integration can be effectively carried out inside Coq.

Finally we show how, without any additional effort, this work can be extended to define a Stieltjes integral (Section 12.1.8).

12.1 Informal Presentation of Riemann Integration

We will implement Riemann integration as follows:

1. Define step functions;
2. Introduce applicative functors and show that step functions form an applicative functor;
3. Show that the step functions form a metric space under both the L^1 and L^∞ norms;
4. Define integrable functions as the completion of the step functions under the L^1 norm;
5. Define integration first on step functions and lift it to operate on integrable functions;

†. Work in this chapter was done in collaboration with Bas Spitters.

6. Define an injection from the continuous functions to the integrable functions in order to integrate them.

At the end, we will see that it is natural to generalize our Riemann integral to a Stieltjes integral.

12.1.1 Step Functions

Our first goal will be to define (formal) step functions and some important operations on them. For any type X, we first define the inductive data type of (rational) step functions from the unit interval to X, denoted by $\mathfrak{S}(X)$. A step function is either a constant function, $\mathsf{const}\ x$, for some $x\colon X$, or two step functions, $f\colon \mathfrak{S}(X)$ and $g\colon \mathfrak{S}(X)$ glued at a point in o, $\mathsf{glue}\ o f g$, where o must be a rational number strictly between 0 and 1. We will sometimes write $(\mathsf{const}\, x)$ as $\hat{x}$, and $(\mathsf{glue}\, o f g)$ as $f \rhd o \lhd g$.

Definition 12.1. The rules for constructing the inductive data type $\mathfrak{S}$:

$$\frac{x\colon X}{\mathsf{const}\, x\colon \mathfrak{S}(X)} \qquad\qquad \frac{o\colon]0,1[_{\mathbb{Q}} \quad f\colon \mathfrak{S}(X) \quad g\colon \mathfrak{S}(X)}{f \rhd o \lhd g\colon \mathfrak{S}(X)}$$

The elements of this inductive type are intended to be interpreted as step functions on [0,1]. The interpretation of $\hat{x}$ is the constant function on [0,1] returning x. The interpretation of $f \rhd o \lhd g$ is f squeezed into the interval $[0,o]$ and g squeezed into the interval $[o,1]$. In this sense f and g are "glued" together.

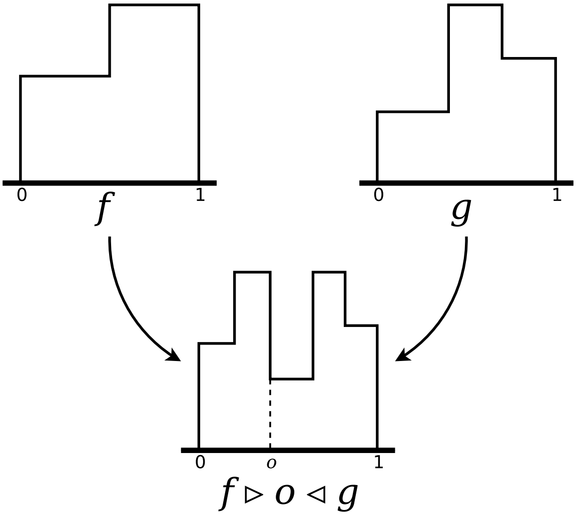


Figure 12.1. Given two step functions f and g, the step function $f \rhd o \lhd g$ is f squeezed into $[0,o]$ and g squeezed into $[o,1]$.

Even though we call step functions “functions”, they are not really functions, and we never formally interpret them as functions. They are a formal structure that takes the place of step functions from classical mathematics. It does not matter that our informal interpretation of $f \triangleright o \triangleleft g$ is not well defined at o, because the step functions are intended for integration, not for evaluation at a point.

One can see that this inductive type is a binary tree whose nodes hold data of type $]0,1[_{\mathbb{Q}}$, and whose leaves have type X. We work with an equivalence relation on this binary tree structure that identifies different ways of constructing the same step function. Informally, this is the equivalence relation induced by our interpretation; the formal equivalence relation is defined in Section 12.1.4.

We define two sorts of inverses to glue which we call left-split and right-split. Given $f\colon \mathfrak{S}(X)$ and $a\colon]0,1[_{\mathbb{Q}}$ we define left-split (written as $f \blacktriangleright a : \mathfrak{S}(X)$) and right-split (written as $a \blacktriangleleft f : \mathfrak{S}(X)$) as follows:

Definition 12.2.

$$
\begin{aligned}
\hat{x} \blacktriangleright a &:= \hat{x}\\
(f_l \triangleright o \triangleleft f_r) \blacktriangleright a &:= \begin{cases} f_l \blacktriangleright \frac{a}{o} & (\text{if } a < o)\\ f_l & (\text{if } a = o)\\ f_l \triangleright \frac{o}{a} \triangleleft (f_r \blacktriangleright \frac{a-o}{1-o}) & (\text{if } a > o)\end{cases}\\
a \blacktriangleleft \hat{x} &:= \hat{x}\\
a \blacktriangleleft (f_l \triangleright o \triangleleft f_r) &:= \begin{cases} (\frac{a}{o} \blacktriangleleft f_l) \triangleright \frac{o-a}{1-a} \triangleleft f_r & (\text{if } a < o)\\ f_r & (\text{if } a = o)\\ \frac{a-o}{1-o} \blacktriangleleft f_r & (\text{if } a > o).\end{cases}
\end{aligned}
$$

Informally, the left split $(f \blacktriangleright a)$ takes the portion of f on the interval $[0,a]$ and scales it up to the full interval $[0,1]$. The right split $(a \blacktriangleleft f)$ does the same thing for the portion of f on the interval $[a,1]$. We have that $(f \blacktriangleright a) \triangleright a \triangleleft (a \blacktriangleleft f) \asymp f$ holds, which means that gluing back the left and right pieces of a step function split at a returns an equivalent function back. However, this process does not generally return an identical representation. The formal definition of the equivalence relation is defined later in Section 12.1.4.

The inductive type for step functions has an associated catamorphism which we call fold.

Definition 12.3.

$$
\begin{aligned}
\mathsf{fold} &: (X \Rightarrow Y) \Rightarrow (]0,1[_{\mathbb{Q}} \Rightarrow Y \Rightarrow Y \Rightarrow Y) \Rightarrow \mathfrak{S}(X) \Rightarrow Y\\
\mathsf{fold}(\varphi, \psi)(\hat{x}) &:= \varphi(x)\\
\mathsf{fold}(\varphi, \psi)(f \triangleright o \triangleleft g) &:= \psi(o, \mathsf{fold}(\varphi, \psi)(f), \mathsf{fold}(\varphi, \psi)(g)).
\end{aligned}
$$

This fold operation is used in many places. For instance, it is used to define two metrics on step functions (Section 12.1.5) or to check whether a property holds globally on [0,1] (Section 12.1.4). Not every fold respects the equivalence relation on step functions, so we need to prove that each fold instance we use respects the equivalence relation.

12.1.2 Step Functions Form a Monad

The step function type constructor $\mathfrak{S}$ forms a monad similar to the reader monad $\lambda X.[0,1] \Rightarrow X$ [Wadler, 1992]. The unit of $\mathfrak{S}$ is the constant function, map is defined in the obvious way using fold, and the join from $\mathfrak{S}(\mathfrak{S}(X))$ to $\mathfrak{S}(X)$ is the formal variant of the join function from the reader monad, $\mathsf{join}(f)(z) := f(z)(z)$, which considers a step function of step functions as a step function of two inputs and returns the step function of its diagonal:

Definition 12.4.

$$\begin{aligned} \mathsf{join}\left(\hat{f}\right) &:= f \\ \mathsf{join}(f \rhd o \lhd g) &:= \mathsf{join}(\mathsf{map}(\lambda x.x \blacktriangleright o)(f)) \rhd o \lhd \mathsf{join}(\mathsf{map}(\lambda x.o \blacktriangleleft x)(g)). \end{aligned}$$

Rather than use these monadic functions, we use the applicative functor interface to this monad.

12.1.3 Applicative Functors

Let $\mathfrak{M}$ be a strong monad. To lift a function $f\colon X \Rightarrow Y$ to a function $\mathfrak{M}(X) \Rightarrow \mathfrak{M}(Y)$, we use $\mathsf{map}\colon (X \Rightarrow Y) \Rightarrow \mathfrak{M}(X) \Rightarrow \mathfrak{M}(Y)$. Lifting a function with two curried arguments is possible using a similar function map2. However, to avoid having to write a function map$n$ for each natural number n, one can use the theory of applicative functors. An *applicative functor* consists of a type constructor $\mathfrak{T}$ and two functions:

$$\begin{aligned} \mathsf{pure} &: X \Rightarrow \mathfrak{T}(X) \\ \mathsf{ap} &: \mathfrak{T}(X \Rightarrow Y) \Rightarrow \mathfrak{T}(X) \Rightarrow \mathfrak{T}(Y) \end{aligned}$$

The function pure lifts any value inside the functor. The ap function applies a function inside the functor to a value inside the functor to produce a value inside the functor. We denote $\mathsf{pure}(x)$ by $\hat{x}$, as was done for monads, and we denote $\mathsf{ap}(f)(x)$ by f @ x. An applicative functor must satisfy the following laws [McBride and Paterson, 2008]:

$$\begin{array}{ll} \hat{\mathbf{I}} \mathbin{@} v \asymp v & \text{Identity} \\ \hat{\mathbf{B}} \mathbin{@} u \mathbin{@} v \mathbin{@} w \asymp u \mathbin{@} (v \mathbin{@} w) & \text{Composition} \\ \hat{f} \mathbin{@} \hat{x} \asymp \widehat{f(x)} & \text{Homomorphism} \\ u \mathbin{@} \hat{y} \asymp \widehat{\mathrm{ev}_y} \mathbin{@} u & \text{Interchange} \end{array}$$

Where **B** and **I** are the composition and identity combinators respectively (see Section 12.2.4) and $\mathrm{ev}_y := \lambda f.\, f(y)$ is the function which evaluates at y.

Every strong monad induces the canonical applicative functor [McBride and Paterson, 2008] where

$$\begin{aligned} \mathsf{pure} &:= \mathsf{unit} \\ f \,@\, x &:= \mathsf{bind}\,(\lambda g.\, \mathsf{map}(g)(x))\, f. \end{aligned}$$

As the name suggests, every applicative functor can be seen as a functor. Given an applicative functor $\mathfrak{T}$, we define $\mathsf{map}\colon (X \Rightarrow Y) \Rightarrow \mathfrak{T}(X) \Rightarrow \mathfrak{T}(Y)$ as

$$\mathsf{map}(f)(x) := \hat{f} \,@\, x.$$

When $\mathfrak{T}$ is generated from a monad, this definition of map is equivalent to the definition of map associated with the monad.

12.1.4 The Step Function Applicative Functor

The ap function for step functions $\mathfrak{S}$ applies a step function of functions to a step function of argument pointwise. It is formally defined as follows:

Definition 12.5.

$$\begin{aligned} \hat{f} @ \hat{x} &:= \widehat{f(x)} \\ \hat{f} @ (x_l \rhd o \lhd x_r) &:= (\hat{f} @ x_l) \rhd o \lhd (\hat{f} @ x_r) \\ (f_l \rhd o \lhd f_r) @ x &:= (f_l @ (x \blacktriangleright o)) \rhd o \lhd (f_r @ (o \blacktriangleleft x)). \end{aligned}$$

For step functions $\mathfrak{S}$, we denote $\mathsf{map}(f)(x)$ by $f \male x$. This notation is meant to suggest the similarity with the composition operation, which is the definition of map for the reader monad $\lambda X.[0{,}1] \Rightarrow X$.

Definition 12.6. The binary version of map is defined in terms of map and ap.

$$\mathsf{map2}(f)(a, b) := f \male a \,@\, b.$$

Higher arity maps can be defined in a similar way; however, we found it more natural to simply use map and ap everywhere.

We will often use $\mathsf{map2}$ to lift infix operations. Because of this, we give it a special notation.

Definition 12.7. If $\circledast$ is some infix operator such that $\lambda x y. x \circledast y\colon X \Rightarrow Y \Rightarrow Z$, then we define

$$f \,\langle \circledast \rangle\, g := (\lambda x y. x \circledast y) \male f \,@\, g,$$

where $f\colon\mathfrak{S}(X)$, $g\colon\mathfrak{S}(Y)$, and $f\langle\circledast\rangle g\colon\mathfrak{S}(Z)$.

For example, if $f, g\colon\mathfrak{S}(\mathbb{Q})$ are rational step functions, then $f\langle-\rangle g$ is the pointwise difference between f and g as a rational step function.

We can lift relations to step functions as well. A relation is simply a function to $\star$, the type of propositions. Thus a binary relation $\propto$ has a type $\lambda x y. x \propto y\colon X \Rightarrow Y \Rightarrow \star$. If we use $\mathsf{map2}$, we end up with an function $\lambda f g. f\langle\propto\rangle g\colon \mathfrak{S}(X) \Rightarrow \mathfrak{S}(Y) \Rightarrow \mathfrak{S}(\star)$. The result is not a proposition, but rather a step function of propositions. Classically, this corresponds to a step function of Booleans. In other words, $\mathfrak{S}(\star)$ represents a type of step characteristic functions on [0,1].

Each way of turning a characteristic function into a proposition determines a different kind of predicate lifting [Schröder, 2005]. For our purposes, we are interested in the one that asks the characteristic function to hold everywhere. The function $\mathsf{fold}_\star : \mathfrak{S}(\star) \Rightarrow \star$ does this by folding conjunction over a step function.

Definition 12.8. $\mathsf{fold}_\star := \mathsf{fold}(\mathbf{I}, \lambda o p q. p \wedge q)$.

When this function is composed with $\mathsf{map2}$, the result lifts a relation to a relation on step functions.

Definition 12.9. $f\{\propto\}g := \mathsf{fold}_\star(f\langle\propto\rangle g)$.

For example, we define equivalence on step functions by lifting the equivalence relation on X.

Definition 12.10. $f \asymp_{\mathfrak{S}(X)} g := f\{\asymp_X\}g$.

Two step functions are equivalent if they are pointwise equivalent everywhere.

Similarly, we define a partial order on step functions by lifting the inequality relation on $\mathbb{Q}$.

Definition 12.11. $f \le_{\mathfrak{S}(\mathbb{Q})} g := f\{\le_{\mathbb{Q}}\}g$.

A step function f is less than a step function g if f is pointwise less than g everywhere.

12.1.5 Two Metrics for Step Functions

The step functions over the rational numbers, $\mathfrak{S}(\mathbb{Q})$, form a metric space in two ways, with the L^∞ metric and the L^1 metric. We first define the two norms on the step functions.

Definition 12.12.

$$\begin{aligned}\|f\|_{\infty} &:= \mathsf{fold}_{\sup}(\mathsf{abs} \mars f)\\ \|f\|_{1} &:= \mathsf{fold}_{\mathrm{affine}}(\mathsf{abs} \mars f)\end{aligned}$$

where

$$\begin{aligned}\mathsf{fold}_{\sup} &:= \mathsf{fold}(\mathbf{I}, \lambda o x y.\, \mathsf{max}\,(x, y))\\ \mathsf{fold}_{\mathrm{affine}} &:= \mathsf{fold}(\mathbf{I}, \lambda o x y. o\, x + (1 - o)\, y)\end{aligned}$$

and $\mathsf{abs}: \mathbb{Q} \Rightarrow \mathbb{Q}$ is the absolute value function on $\mathbb{Q}$.

The function $\mathsf{fold}_{\sup} : \mathfrak{S}(\mathbb{Q}) \Rightarrow \mathbb{Q}$ returns the supremum of the step function, while the function $\mathsf{fold}_{\mathrm{affine}}: \mathfrak{S}(\mathbb{Q}) \Rightarrow \mathbb{Q}$ returns the integral of a step function.

Next, the metric distance between two step functions is defined.

Definition 12.13.

$$\begin{aligned}d^{\infty}(f, g) &:= \|f \langle - \rangle g\|_{\infty}\\ d^{1}(f, g) &:= \|f \langle - \rangle g\|_{1}.\end{aligned}$$

Finally, the distance relations are defined in terms of the distance functions.

Definition 12.14.

$$\begin{aligned}\boldsymbol{B}_{\varepsilon}^{\mathfrak{S}^{\infty}(\mathbb{Q})}(f, g) &:= d^{\infty}(f, g) \leq \varepsilon\\ \boldsymbol{B}_{\varepsilon}^{\mathfrak{S}^{1}(\mathbb{Q})}(f, g) &:= d^{1}(f, g) \leq \varepsilon.\end{aligned}$$

Both these metric spaces are prelength spaces.

When we need to be clear which metric space is being used, we will use the notation $\mathfrak{S}^{\infty}(\mathbb{Q})$ or $\mathfrak{S}^{1}(\mathbb{Q})$.

The two fold functions defined in this section are uniformly continuous for their respective metrics.

$$\begin{aligned}\mathsf{fold}_{\sup} &: \mathfrak{S}^{\infty}(\mathbb{Q}) \rightarrow \mathbb{Q}\\ \mathsf{fold}_{\mathrm{affine}} &: \mathfrak{S}^{1}(\mathbb{Q}) \rightarrow \mathbb{Q}\end{aligned}$$

The identity function is uniformly continuous in one direction, $\iota: \mathfrak{S}^{\infty}(\mathbb{Q}) \rightarrow \mathfrak{S}^{1}(\mathbb{Q})$; however, the other direction is not uniformly continuous.

The metrics $\mathfrak{S}^{\infty}(X)$ and $\mathfrak{S}^{1}(X)$ can be defined for any metric space X:

$$\begin{aligned}\boldsymbol{B}_{\varepsilon}^{\mathfrak{S}^{\infty}(X)}(f, g) &:= \mathsf{fold}_{\star}(\boldsymbol{B}_{\varepsilon}^{X} \mars f @ g)\\ \boldsymbol{B}_{\varepsilon}^{\mathfrak{S}^{1}(X)}(f, g) &:= \exists h: \mathfrak{S}(\mathbb{Q}^{+}).\, \mathsf{fold}_{\star}(\boldsymbol{B}^{X} \mars h @ f @ g) \wedge \|h\|_{1} \leqslant \varepsilon\end{aligned}$$

We have implemented the generic $\mathfrak{S}^\infty(X)$ metric in our formalization. However, for the L^1 space, we have only implemented the specific $\mathfrak{S}^1(\mathbb{Q})$ metric.

12.1.6 Integrable Functions and Bounded Functions

The bounded functions and the integrable functions are defined as the completion of the step functions under the L^∞ and the L^1 metrics respectively.

Definition 12.15.

$$\begin{aligned}\mathfrak{B} &:= \mathfrak{C}\circ\mathfrak{S}^\infty\\ \mathfrak{I} &:= \mathfrak{C}\circ\mathfrak{S}^1.\end{aligned}$$

In Section 12.1.1, we informally interpreted elements of $\mathfrak{S}(X)$ as (partially defined) functions on [0,1]. Similarly, we can informally interpret each bounded function as a (partially defined) function. Consider f: $\mathfrak{B}(\mathbb{Q})$. Define $g_n := f\left(\frac{1}{n}\right)$. Then $\lim_{n\to\infty} g_n(x)$ exists for all points x in [0,1] except perhaps for the (rational) splitting points of the step functions g_n. At the points where this limit is defined, it is (classically) continuous.

To every Riemann integrable function on [0,1] we can associate an element in $\mathfrak{I}(\mathbb{Q})$. Moreover, functions f and g such that $\int |f-g| \asymp 0$ will be assigned to equivalent elements in $\mathfrak{I}(\mathbb{Q})$. This definition can be extended to every *generalized* Riemann integrable function, where a function h is generalized Riemann integrable if $h_n := \mathsf{max}\left(\mathsf{min}\,(h,\hat{n}),\widehat{-n}\right)$ is integrable for each n and the limit of $\int h_n$ converges (even though h_n may not converge pointwise everywhere). Conversely, we can informally interpret every element f of $\mathfrak{I}(\mathbb{Q})$ as a generalized Riemann integrable function. Define g_n as the sequence

$$g_n := f\left(\frac{1}{2^{2n+1}}\right).$$

By the fundamental lemma of integration [Lang, 1993], g_n converges pointwise almost everywhere. Let g be this pointwise limit. Then g is a generalized Riemann integrable function associated with f.

The bounded functions have a supremum operation, $\mathsf{sup}:\mathfrak{B}(\mathbb{Q})\to\mathbb{R}$ and, similarly, the integrable functions have an integration operation, $\int : \mathfrak{I}(\mathbb{Q})\to\mathbb{R}$ which are defined by lifting the two folds from the previous section (recall Definition 10.14).

Definition 12.16.

$$\begin{aligned}\mathsf{sup}\,(f) &:= \mathsf{map}'_{\mathfrak{C}}(\mathsf{fold}_{\text{sup}})(f)\\ \int f &:= \mathsf{map}'_{\mathfrak{C}}(\mathsf{fold}_{\text{affine}})(f)\end{aligned}$$

There is an injection from the bounded functions into the integrable functions defined by lifting the injection on step functions: $\bar{\iota} : \mathfrak{B}(\mathbb{Q}) \to \mathfrak{I}(\mathbb{Q})$. However, there is no injection from integrable function to bounded functions. Thus bounded functions can be integrated, but integrable functions may not have a supremum.

12.1.7 Riemann Integral

The process for integrating a function is as follows. Given a function f, one needs to find an equivalent representation of f as an integrable function and then this integrable function can be integrated. We will consider how to integrate uniformly continuous functions on [0,1], which is a useful class of functions to integrate.

We convert a uniformly continuous function to an integrable function by a two step process. First, we will convert it to a bounded function, and then the bounded function can be converted to an integrable function using the injection defined in the previous section.

To produce a bounded function, one needs to create a step function that approximates f within ε for any value ε: $\mathbb{Q}^{+}$. The usual way of doing this is to create a step function where each step has width no more than 2 $\mu_f(\varepsilon)$. The value at each step is taken by sampling the function at the center of the step.

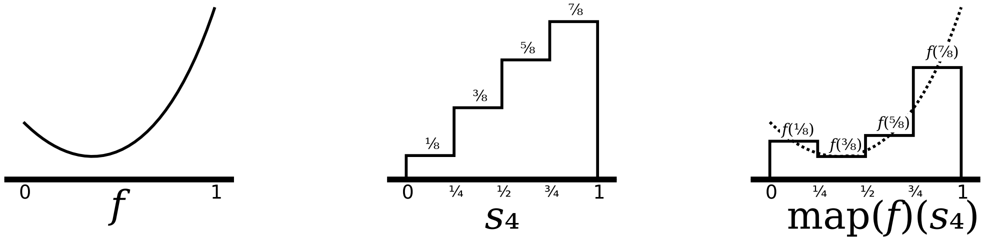


Figure 12.2. Given a uniformly continuous function f and a step function s_4 that approximates the identity function, the step function $\mathsf{map}(f)(s_4)$ (or f ♂ s_4) approximates f in the familiar Riemann way.

When developing the above, it became clear that one can achieve the desired result by creating a step function whose values are the sample inputs, and then mapping f over these "sampling step-functions" (see Figure 12.2). In fact, the limit of these "sampling step-functions" is simply the identity function on [0,1] represented as a bounded function, $\mathbf{I}_{[0,1]} : \mathfrak{B}(\mathbb{Q})$ (see Section 12.2.6). Given any uniformly continuous function $f\colon \mathbb{Q} \to \mathbb{Q}$, we can prove that $\mathsf{map}_{\mathfrak{S}^{\infty}}(f)\colon \mathfrak{S}^{\infty}(\mathbb{Q}) \to \mathfrak{S}^{\infty}(\mathbb{Q})$ is uniformly continuous. We can then lift again to operate on bounded functions, $\mathsf{map}'_{\mathfrak{C}}(\mathsf{map}_{\mathfrak{S}^{\infty}}(f)) : \mathfrak{B}(\mathbb{Q}) \to \mathfrak{B}(\mathbb{Q})$. Applying this to $\mathbf{I}_{[0,1]}$ yields f restricted to [0,1] as a bounded function, which can then be converted to an integrable function and integrated.

Definition 12.17. $\int_{[0,1]} f := \int \left(\bar{\iota}\left(\mathsf{map}'_{\mathfrak{C}}(\mathsf{map}_{\mathfrak{S}^\infty}(f))\left(\mathbf{I}_{[0,1]}\right)\right)\right)$.

With a small modification, this process will also work for $f\colon \mathbb{Q} \to \mathbb{R}$. In this case $\mathsf{map}(f)$ has type $\mathfrak{S}(\mathbb{Q}) \Rightarrow \mathfrak{S}(\mathbb{R})$, Fortunately, there is an injection $\mathsf{dist} : \mathfrak{S}(\mathbb{R}) \Rightarrow \mathfrak{B}(\mathbb{Q})$, that interprets a step function of real values as a bounded function (see Definition 12.20). We can prove that the composition $\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}(f))\colon \mathfrak{S}^\infty(\mathbb{Q}) \to \mathfrak{B}(\mathbb{Q})$ is uniformly continuous. Then, proceeding in a similar fashion, this can be lifted with bind' (recall Definition 10.21) and applied to $\mathbf{I}_{[0,1]}$ to yield f restricted to [0,1] as a bounded function, which can then be integrated.

Definition 12.18. $\int_{[0,1]} f := \int \left(\bar{\iota}\left(\mathsf{bind}'_{\mathfrak{C}}(\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}(f)))\left(\mathbf{I}_{[0,1]}\right)\right)\right)$.

An arbitrary uniformly continuous function $f\colon \mathbb{R} \to \mathbb{R}$ can be integrated on [0,1] by integrating $f \circ \mathsf{unit}_{\mathfrak{C}} : \mathbb{Q} \to \mathbb{R}$ because the Riemann integral only depends on the value of functions at rational points.

12.1.8 Stieltjes Integral

Given the previous presentation, any bounded function could be used in place of $\mathbf{I}_{[0,1]}$. A natural question arises: what happens when $\mathbf{I}_{[0,1]}$ is replaced by another bounded function, $g\colon \mathfrak{B}(\mathbb{Q})$? An analysis shows that the result is the Stieltjes integral with respect to g^{-1}, when g is non-decreasing.

Definition 12.19. $\int f \mathrm{d}g^{-1} := \int \left(\bar{\iota}\left(\mathsf{bind}'_{\mathfrak{C}}(\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}(f)))(g)\right)\right)$.

We never intended to develop the Stieltjes integral; however, it practically falls out of our work for free. This is not quite as general as the Stieltjes integral for three reasons. Because g is defined on [0,1], this means that g^{-1}'s range must go from 0 to 1. Essentially, g^{-1} must be a cumulative distribution function and, hence, g is a quantile function. Secondly, because g is a bounded function, g^{-1} must have compact support (meaning g^{-1} must be 0 to the left of its support and 1 to the right of its support). Thirdly, our bounded functions can only have discontinuities at rational points.

We have tried to allow g to be an arbitrary integrable function (this would remove some of the previous restrictions); however, we have been unable to constructively show that $\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}(f))\colon \mathfrak{S}^1(\mathbb{Q}) \Rightarrow \mathfrak{I}(\mathbb{Q})$ is uniformly continuous when f is. We have generated counterexamples where f is uniformly continuous with modulus μ and $\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}(f))$ is *not* uniformly continuous with modulus μ; however, for our particular counterexamples, $\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}(f))$ is still uniformly continuous with a different modulus.

Still, our integral should allow one to integrate with respect to some interesting distributions such as the Dirac distribution and the Cantor distribution.

12.1.9 Distributing Monads

The function $\mathsf{dist}: \mathfrak{S}(\mathbb{R}) \Rightarrow \mathfrak{B}(\mathbb{Q})$ combines two monads on metric spaces, $\mathfrak{C}$ and $\mathfrak{S}$. The function dist has type $\mathfrak{S}(\mathfrak{C}(\mathbb{Q})) \Rightarrow \mathfrak{C}(\mathfrak{S}(\mathbb{Q}))$. In general, the composition of two monads $\mathfrak{M} \circ \mathfrak{N}$ forms a monad when there is a distribution function dist : $\mathfrak{N}(\mathfrak{M}(X)) \rightarrow \mathfrak{M}(\mathfrak{N}(X))$ satisfying certain laws [Beck, 1969]. Below we state the laws in a more familiar function style [Jones and Duponcheel, 1993]:[12.1]

$$\begin{aligned}
\mathsf{dist} \circ \mathsf{map}_{\mathfrak{N}}\,(\mathsf{map}_{\mathfrak{M}}(f)) &\asymp \mathsf{map}_{\mathfrak{M}}(\mathsf{map}_{\mathfrak{N}}(f)) \circ \mathsf{dist}\\
\mathsf{dist} \circ \mathsf{unit}_{\mathfrak{N}} &\asymp \mathsf{map}_{\mathfrak{M}}(\mathsf{unit}_{\mathfrak{N}})\\
\mathsf{dist} \circ \mathsf{map}_{\mathfrak{N}}(\mathsf{unit}_{\mathfrak{M}}) &\asymp \mathsf{unit}_{\mathfrak{M}}\\
\mathsf{prod} \circ \mathsf{map}_{\mathfrak{N}}(\mathsf{dorp}) &\asymp \mathsf{dorp} \circ \mathsf{prod}
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{prod} &:= \mathsf{map}_{\mathfrak{M}}(\mathsf{join}_{\mathfrak{N}}) \circ \mathsf{dist}\\
\mathsf{dorp} &:= \mathsf{join}_{\mathfrak{M}} \circ \mathsf{map}_{\mathfrak{M}}(\mathsf{dist}).
\end{aligned}$$

Definition 12.20. In our case, the distribution function is defined as

$$\begin{aligned}
\mathsf{dist} &: \mathfrak{S}^{\infty}(\mathfrak{C}(X)) \rightarrow \mathfrak{C}(\mathfrak{S}^{\infty}(X))\\
\mathsf{dist}(f) &:= \lambda\varepsilon.\mathsf{map}_{\mathfrak{S}^{\infty}}(\lambda x.x(\varepsilon))(f).
\end{aligned}$$

The function dist maps a step function f with values in the completion of X to a collection of approximations f_{ε}: $\mathfrak{S}^{\infty}(X)$ to the function f such that for all ε in $\mathbb{Q}^{+}$, $|f - f_{\varepsilon}| \leq \varepsilon$ “pointwise”.

12.2 Implementation in Coq

In this section, we treat aspects related to our implementation in Coq.

12.2.1 Glue and Split

As discussed in Section 12.1.1, step functions are an inductive structure defined by two constructors. One constructor `constStepF` creates constant step functions, and the other constructor, `glue`, squeezes two step functions together, joining them together at a given point o: $]0,1[_{\mathbb{Q}}$. One of the first operations we defined on step functions (after defining `fold`) was `Split`, which is like the opposite of `glue`. Recall from Section 12.1.1 that, given a step function f and a point a: $]0,1[_{\mathbb{Q}}$, `Split` splits f into two pieces at a. The functions `SplitL` and `SplitR` return the left step function and the right step function respectively. Table 12.1 lists the association between our mathematical notation and the concrete syntax used in Coq.

[12.1] For the $\mathfrak{S}$ and $\mathfrak{C}$ monads, we formally checked all of these rules apart from the last one which was too tedious; however, the correctness of the integral does not depend on the proofs of these laws.

Mathematical Notation	Coq Syntax
$\hat{x}$	`constStepF x`
$f \rhd o \lhd g$	`glue o f g`
$f \blacktriangleright a$	`SplitL f a`
$a \blacktriangleleft f$	`SplitR f a`
$(f \blacktriangleright a, a \blacktriangleleft f)$	`Split f a`

Table 12.1. The concrete syntax used in Coq for our step function notation.

The key to reasoning about `Split` was to prove the `Split-Split` lemmas:

$$\begin{aligned} a\,b = c \;&\Rightarrow\; f \blacktriangleright a \blacktriangleright b \asymp f \blacktriangleright c \\ a + b - a\,b = c \;&\Rightarrow\; b \blacktriangleleft a \blacktriangleleft f \asymp c \blacktriangleleft f \\ a + b - a\,b = c \Rightarrow d\,c = a \;&\Rightarrow\; (a \blacktriangleleft f) \blacktriangleright b \asymp d \blacktriangleleft (f \blacktriangleright c) \end{aligned}$$

This collection of lemmas shows how the splits combine and distribute over each other. With sufficient case analysis, one can prove the above lemmas. These lemmas, combined with a few other useful lemmas (such as `Split-Map` lemmas) provided enough support to prove the laws for applicative functors without difficulty.

12.2.2 Equivalence of Step Functions

The work in the previous section defined an applicative functor of step functions over any type X. From this point on, we will require that X be a setoid (see Section 2.2.8). In order to help facilitate this, in our development we define new functions, `constStepF`, `glue`, `Split`, etc., that operate on step functions of setoids rather than step functions of types. These functions are definitionally equal to the previous functions, but their types now carry the setoid relation from their argument types to their result types. These new function names shadow the old function names, and the lemmas about them need to be repeated; however, their proofs are trivial by using previous proofs.

Perhaps the biggest challenge we encountered in our formalization was to prove that lifting setoid equivalence to step functions (Section 12.1.3) is indeed an equivalence relation—in particular showing that it is transitive. We eventually succeeded after creating some lemmas about the interaction between the equivalence relation and `Split`, etc.

12.2.3 Common Partitions

When reasoning about two (or more) step functions, it is common to split up one of the step functions so that it shares the same partition structure as the other step function. This allows one to do induction over two step functions and have both step functions decompose the same way. Eventually, we abstracted this pattern of reasoning into an induction-like principle.

`Lemma StepF_ind2 :`

$$\begin{array}{l}
\forall X Y.\forall \Psi : X \Rightarrow Y \Rightarrow \star.\\
(\forall s_0\, s_1\, t_0\, t_1 : \mathfrak{S}(X).\, s_0 \asymp s_1 \Rightarrow t_0 \asymp t_1 \Rightarrow \Psi(s_0, t_0) \Rightarrow \Psi(s_1, t_1) \qquad) \Rightarrow\\
(\forall x{:}\, X.\forall y{:}\, Y. \qquad \Psi(\quad \hat{x}, \qquad \hat{y} \quad)) \Rightarrow\\
(\forall o.\forall s_l\, s_r{:}\, \mathfrak{S}(X).\forall t_l\, t_r{:}\, \mathfrak{S}(Y).\, \Psi(s_l, t_l) \Rightarrow \Psi(s_r, t_r) \Rightarrow \Psi(s_l \rhd o \lhd s_r,\ t_l \rhd o \lhd t_r)) \Rightarrow\\
\forall s{:}\, \mathfrak{S}(X).\forall t{:}\, \mathfrak{S}(Y). \qquad \Psi(\quad s, \qquad t \quad).
\end{array}$$

This lemma may look complex, but it is as easy to use in Coq as an induction principle for an inductive family. Normally one would reason about two step functions by assuming, without loss of generality, that they have a common partition, then doing induction over that partition. Our lemma above combines these two steps into one. In one step, one does induction as if the two functions have a common partition. This lemma was inspired by McBride and McKinna's work on views in dependent type theory [McBride and McKinna, 2004b]. It allows one to "view" two step functions as having a common partition.

The lemma is used by applying it to a goal of the form `forall (s t : StepF X), <expr>`, which can be created by generalizing two step functions. There are only two cases to consider. One case is when `s` and `t` are both constant step functions. The other case is when `s` and `t` are each glued together from two step functions *at the same point*. There is, however, a side condition to be proved. One has to show that `<expr>` respects the equivalence relation on step functions for `s` and `t`. Fortunately, `<expr>` is typically constructed from respectful functions, and proving this side condition is easy.

This induction lemma was very useful for proving the combinator equations in Section 12.2.4.

12.2.4 Combinators

The combinators **B** and **I** are preserved by every applicative functor (see Section 12.1.3). For the applicative functor $\mathfrak{S}$, all lambda expressions are preserved. To show this, it is sufficient to show that each of the **BCKW** combinators are preserved. These are the combinators defined by:

- $\mathbf{B}(f)(g)(x) := f(g(x))$ (compose)
- $\mathbf{C}(f)(x)(y) := f(y)(x)$ (interchange)
- $\mathbf{I}(x) := x$ (identity)
- $\mathbf{K}(x)(y) := x$ (discard)
- $\mathbf{W}(f)(x) := f(x)(x)$ (duplicate)

The identity combinator is redundant because $\mathbf{I} \asymp \mathbf{W}(\mathbf{K})$, but it is still useful.

All lambda expressions can be rewritten in a "point free" form using these combinators. Using combinators allows us to reason about the lambda calculus without worrying about binders, which are notoriously difficult to do by hand (see Section 4.7).

Theorem 12.21. *The remaining combinators,* **CKW**, *are preserved by the* $\mathfrak{S}$ *monad.*

$$\begin{aligned}
\mathbf{C}\,♂\, f \mathbin{@} x \mathbin{@} y &\asymp_{\mathfrak{S}X} f \mathbin{@} y \mathbin{@} x\\
\mathbf{K}\,♂\, x \mathbin{@} y &\asymp_{\mathfrak{S}X} x\\
\mathbf{W}\,♂\, f \mathbin{@} x &\asymp_{\mathfrak{S}X} f \mathbin{@} x \mathbin{@} x
\end{aligned}$$

This means that we can lift any function definable with the λ-calculus to step functions.

12.2.5 Lifting Theorems

During our development, we often needed to prove statements like the transitivity of the order relation on the step functions:

$$\forall fgh\colon \mathfrak{S}(\mathbb{Q}).\, f\{\leq_{\mathbb{Q}}\}g \Rightarrow g\{\leq_{\mathbb{Q}}\}h \Rightarrow f\{\leq_{\mathbb{Q}}\}h$$

We would like to deduce this statement from the transitivity of the corresponding pointwise relation:

$$\forall xyz\colon \mathbb{Q}.x \leq_{\mathbb{Q}} y \Rightarrow y \leq_{\mathbb{Q}} z \Rightarrow x \leq_{\mathbb{Q}} z$$

First, we use a lemma that lifts universal statements about an arbitrary predicate $R\colon X \Rightarrow Y \Rightarrow Z \Rightarrow \star$ to a universal statement about step functions:

$$\begin{aligned}
&(\forall x\colon X.\forall y\colon Y.\forall z\colon Z.R(x,y,z)) \Rightarrow\\
&\forall f\colon \mathfrak{S}(X).\forall g\colon \mathfrak{S}(Y).\forall h\colon \mathfrak{S}(Z).\, \mathsf{fold}_{\star}\,(R\,♂\, f \mathbin{@} g \mathbin{@} h)
\end{aligned}$$

This yields

$$\forall fgh\colon \mathfrak{S}(\mathbb{Q}).\, \mathsf{fold}_{\star}((\lambda xyz.x \leq_{\mathbb{Q}} y \Rightarrow y \leq_{\mathbb{Q}} z \Rightarrow x \leq_{\mathbb{Q}} z)\,♂\, f \mathbin{@} g \mathbin{@} h).$$

Next, we would like to "evaluate" the lambda expression as "applied" to the step functions f, g, and h. Because f, g, and h are variables, we need to symbolically evaluate the expression. We avoid dealing with binders by converting the lambda expression into the combinator expression

$$\begin{gathered}
\mathbf{S}(\mathbf{B}(\mathbf{S})(\mathbf{B}(\mathbf{B}(\mathbf{B}(\mathbf{B})(\Rightarrow)))(\leq_{\mathbb{Q}})))(\mathbf{B}(\mathbf{C}(\mathbf{B}(\mathbf{S})(\mathbf{B}(\mathbf{B}(\Rightarrow))(\leq_{\mathbb{Q}}))))(\leq_{\mathbb{Q}}))\\
♂\, f \mathbin{@} g \mathbin{@} h,
\end{gathered}$$

where $\mathbf{S} := \mathbf{B}(\mathbf{B}(\mathbf{B}(\mathbf{W}))(\mathbf{C}))(\mathbf{B}(\mathbf{B}))$ and $(\Rightarrow)$ and $(\leq_{\mathbb{Q}})$ are prefix versions of these infix functions. This substitution is sound because the combinator term and lambda expression can easily be shown to be extensionally equivalent (by normalization), and map and ap are well-defined with respect to extensional equality.

We found the required combinator form by using lambdabot [Bromage and Jäger, 2006], a standard tool for Haskell programmers. It would have been interesting to implement the algorithm for finding the combinator form of a lambda term in Coq; however, this was not the aim of our current research.

Now that the lambda term is expressed in combinator form, we can repeatedly apply the combinator equations from Section 12.1.3 and Section 12.2.4. These equations are exactly the rules of "evaluation" of this expression "applied" to step functions. We put these equations into a database of rewrite rules and used Coq's `autorewrite` system (see Section 2.2.8.1) as part of a small custom tactic to automatically reduce this entire expression in one command, yielding

$$\forall fgh\colon \mathfrak{S}(\mathbb{Q}).\, \mathsf{fold}_\star(f\langle\leq_{\mathbb{Q}}\rangle g\langle\Rightarrow\rangle g\langle\leq_{\mathbb{Q}}\rangle h\langle\Rightarrow\rangle f\langle\leq_{\mathbb{Q}}\rangle h).$$

Finally, we push the $\mathsf{fold}_\star$ inside. To do so, we have proved a lemma which allows us to distribute implication over $\mathsf{fold}_\star$:

$$\forall PQ\colon \mathfrak{S}(\star).\, (\mathsf{fold}_\star\,(P\ \langle\Rightarrow\rangle\ Q)) \Rightarrow \mathsf{fold}_\star(P) \Rightarrow \mathsf{fold}_\star(Q)$$

Repeated application of this lemma yields

$$\forall fgh\colon \mathfrak{S}(\mathbb{Q}).\, f\{\leq_{\mathbb{Q}}\}g \Rightarrow g\{\leq_{\mathbb{Q}}\}h \Rightarrow f\{\leq_Q\}h$$

as required.

12.2.6 The Identity Bounded Function

In order to integrate uniformly continuous functions, we compose them with the identity bounded function to create a bounded function that can be integrated (see Section 12.1.7). This requires defining the identity bounded function on [0,1].

The bounded functions are the completion of step functions under the L^∞ metric. To create a bounded function, we need to generate a step function within ε of the identity function for every $\varepsilon\colon \mathbb{Q}^+$. The number of steps used in the approximation will determine the number of samples of the continuous function f that will be used. For efficiency, we want the approximation to have the fewest number of steps possible. Therefore, we defined a function `stepSample` : `positive` $\Rightarrow \mathfrak{S}(\mathbb{Q})$, where `positive` is the binary positive natural numbers, such that `stepSample` n produces the best approximation of the identity function with n steps.

It is unfortunate that the width of each step is computed during integration, because we know that the result will always be equivalent to $\frac{1}{n}$ for these particular step functions. Perhaps some other data structure for step functions could be used that explicitly stores the length of each step. However, the time spent computing the length of the interval is usually much smaller that the time it takes to sample the continuous function f.

12.2.7 Correctness

We have proved our implementation correct by showing that our Riemann integral is equivalent to the definition of the integral in the C-CoRN library [Cruz-Filipe, 2003]. This library contains many machine verified facts about the Riemann integral. The correctness is one theorem with a 300-line proof mostly consisting of translating facts about the fast implementation of the reals to the C-CoRN library and vice versa. The actual proof is quite general because it only uses certain general properties of the integral, such as linearity and monotonicity.

As a by-product of our development, we can also compute the supremum of any uniformly continuous function on [0,1].

12.2.8 Timings

The version of Riemann integration that we implemented applies to *general* continuous functions and hence has bad complexity behavior. If we knew more about the function, for instance if it is differentiable, faster algorithms could be used [Edalat, 1999].

Function	**Time**
`(answer 3 (Integrate01 Cunit))`	0.18s
`(answer 2 (Integrate01 cos_uc))`	0.52s
`(answer 3 (Integrate01 cos_uc))`	8.55s
`(answer 3 (Integrate01 sin_uc))`	7.48s

Table 12.2. `Time Eval vm_compute in ...` carries out the reduction using Coq's virtual machine. The expression `answer` n asks for an answer to within 10^{-n}. All computations where carried out on an IBM Thinkpad X41.

12.3 Remarks

Because of the way that I have defined uniform continuity, one modulus of continuity applies to an entire function. Even for those parts of the domain where the function changes slowly, we still must approximate the input to the same precision that is needed for those parts where the function changes quickly. This reduces performance somewhat for evaluation of these functions (at the segments where the function changes slowly), but this causes particularly bad performance for integration.

Because we only have a global modulus of continuity, we must use uniform partitions when creating an integrable function from a uniformly continuous function. This means that the function is sampled just as often where the function changes slowly as where the function changes quickly. This uniform sampling can be quite expensive for integration.

There is some potential to increase efficiency by using a “non-uniform” definition of uniform continuity. That is to say, using a definition of uniform continuity that allows different segments of the domain to have local moduli associated with them. Ulrich Berger uses such a definition of uniform continuity to define integration [Berger, 2008]. Simpson also defines an integration algorithm that uses a local modulus for a function that is computed directly from the definition of the function [Simpson, 1998]. However, implementing his algorithm directly in Coq is not possible because it relies on bar induction, which is not available in Coq.

Chapter 13

Compact Sets

How should we define what computable subsets of the plane are? Sir Roger Penrose ponders this question at one point in his book *The Emperor's New Mind* [Penrose, 1989]. Requiring that subsets be decidable is too strict; determining if a point lies on the boundary of a set is undecidable in general. Penrose gives the unit disc, $\{(x,\ y) \,|\, x^2 + y^2 \leq 1\}$, and the epigraph of the exponential function, $\{(x,\ y) \,|\, \exp(x) \leq y\}$, as examples of sets that intuitively ought to be considered computable [Brattka, 2003]. Restricting one's attention to pairs of rational or algebraic numbers may work well for the unit disc, but the boundary of the epigraph of the exponential function contains only one algebraic point. A better definition is needed.

To characterize computable sets, we draw an analogy with real numbers. The computable real numbers are real numbers that can be effectively approximated to arbitrary precision. We can define computable sets in a similar way. We need a dense subset of sets that have finitary representations. In the case of the plane, the simplest candidate is the finite subsets of $\mathbb{Q}^2$. How do we measure the accuracy of an approximation? Distances between subsets can be defined by the Hausdorff metric (see Section 13.2). To construct the real numbers, we completed the rational numbers (see Definition 11.1). Completing the finite subsets of $\mathbb{Q}^2$ with the Hausdorff metric yields the compact sets (see Section 13.3). Because we reason constructively, the generated compact sets can be effectively computed to arbitrary precision.

The unit disc is constructively compact; it can be effectively approximated with finite sets. When a computer attempts to display the unit disc, only a finite set of the pixels can be shown. So instead of displaying an ideal disc, the computer displays a finite set that approximates the disc. This is the key criterion that Penrose's examples enjoy. They can be approximated to arbitrary precision and displayed on a raster.

Technically the epigraph of the exponential function is not compact; however, it is locally compact. One may wish to consider constructive locally compact sets to be computable sets. This would mean that any finite region of a computable set has effective approximations of arbitrary precision.

The usual definition of computable sets used in computable analysis says that a set is computable if the distance to the set is a computable real-valued function. This definition is equivalent to the definition using computable approximations. However, I believe that defining computable sets by effective approximations by finite sets more accurately matches our intuition about sets that can be drawn by a computer.

13.1 Product Metrics

Given two metric spaces X and Y, their Cartesian product $X \times Y$ forms a metric space with the standard sup-metric.

Definition 13.1. $\boldsymbol{B}_\varepsilon^{X\times Y}((a_1, b_1), (a_2, b_2)) := \boldsymbol{B}_\varepsilon^X(a_1, a_2) \wedge \boldsymbol{B}_\varepsilon^Y(b_1, b_2)$.

The product metric preserves each class of metric space defined in Section 9.3, so the product of two decidable metrics is a decidable metric and so forth. Also, the product of two prelength spaces is a prelength space.

The product metric interacts nicely with the completion operation. There is an isomorphism between $\mathfrak{C}(X \times Y)$ and $\mathfrak{C}(X) \times \mathfrak{C}(Y)$. One direction I call couple. The other direction is defined by lifting the projection functions using map':

$$\begin{aligned}
\mathsf{couple} &: \mathfrak{C}(X) \times \mathfrak{C}(Y) \to \mathfrak{C}(X \times Y)\\
\bar{\pi}_1 &: \mathfrak{C}(X \times Y) \to \mathfrak{C}(X)\\
\bar{\pi}_2 &: \mathfrak{C}(X \times Y) \to \mathfrak{C}(Y)
\end{aligned}$$

Definition 13.2. $\mathsf{couple}(x, y) := \lambda\varepsilon.\,(x(\varepsilon), y(\varepsilon))$.

I denote $\mathsf{couple}(x,\, y)$ by $\langle x,\, y\rangle$. The following theorems prove that these functions form an isomorphism.

Theorem 13.3. *$\mathfrak{C}(X \times Y)$ and $\mathfrak{C}(X) \times \mathfrak{C}(Y)$ are isomorphic:*

$$\begin{aligned}
\langle \bar{\pi}_1(z), \bar{\pi}_2(z)\rangle &\asymp z\\
(\bar{\pi}_1\langle x, y\rangle, \bar{\pi}_2\langle x, y\rangle) &\asymp (x, y)
\end{aligned}$$

This isomorphism implies that $\mathfrak{C}$ is a symmetric monoidal monad. Every symmetric monoidal monad is also a commutative strong monad [Kock, 1972]. The tensorial strength is given by the function $\mathsf{strength} : X \times \mathfrak{C}(Y) \to \mathfrak{C}(X \times Y)$ defined as follows:

Definition 13.4. $\mathsf{strength}(a, y) := \langle \hat{a}, y\rangle$.

One could define strength first and then define couple in terms of strength and $\mathsf{swap} : X \times Y \to Y \times X$, but this would lead to a less efficient definition of couple. This is why I defined couple directly.

Tensorial strength should not be confused with functorial strength. Functorial strength states that map itself is a morphism. I proved functorial strength in Section 10.2. In a Cartesian closed category, tensorial and functorial strength are equivalent [Kock, 1972]. However, our category of metric spaces with uniformly continuous functions is not Cartesian closed because evaluation is not uniformly continuous (see Section 10.3.2). This is why I had to develop the two notions of strength separately.

13.2 Hausdorff Metrics

Given a metric space X, we can try to put a metric on predicates (subsets) of X. We start by defining the Hausdorff hemimetric. A hemimetric is a metric without the symmetry and identity of indiscernibles requirement. I define the hemimetric relation over $X \Rightarrow \star$ as follows.

Definition 13.5. $\boldsymbol{H}_\varepsilon^{X\Rightarrow\star}(\mathcal{A},\mathcal{B}) := \forall x \in \mathcal{A}.\, \tilde{\exists} y \in \mathcal{B}.\, \boldsymbol{B}_\varepsilon^X(x, y)$.

Notice the use of the classical existential in this definition (see Definition 2.2). In general, one does not need to know which point in $\mathcal{B}$ is close to a given point in $\mathcal{A}$; it is sufficient to know one exists without knowing which one. Furthermore, there are cases when one cannot know which point in $\mathcal{B}$ is close to a given point in $\mathcal{A}$.

This relation is reflexive and satisfies the triangle inequality. It is not symmetric. I define a symmetric relation as follows.

Definition 13.6. $\boldsymbol{B}_\varepsilon^{X\Rightarrow\star}(\mathcal{A},\mathcal{B}) := \boldsymbol{H}_\varepsilon^{X\Rightarrow\star}(\mathcal{A},\mathcal{B}) \wedge \boldsymbol{H}_\varepsilon^{X\Rightarrow\star}(\mathcal{B},\mathcal{A})$.

This relationship is reflexive, symmetric, and satisfies the triangle law. Notice that if $\mathcal{B} \subseteq \mathcal{A}$ then $\boldsymbol{H}_\varepsilon^{X\Rightarrow\star}(\mathcal{A}, \mathcal{B})$ holds for all ε. The hemimetric captures the subset relationship. If $\mathcal{B} \subseteq \mathcal{A}$ and $\mathcal{A} \subseteq \mathcal{B}$ (i.e. $\mathcal{A} \asymp \mathcal{B}$), then $\boldsymbol{B}_\varepsilon^{X\Rightarrow\star}(\mathcal{A}, \mathcal{B})$ holds for all ε. However, Axiom 5 for metric spaces requires the reverse implication: if $\boldsymbol{B}_\varepsilon^{X\Rightarrow\star}(\mathcal{A}, \mathcal{B})$ holds for all ε, then we want $\mathcal{A} \asymp \mathcal{B}$. Unfortunately, this does not hold in general. Neither does the closedness property hold as required by Axiom 4. To make a true metric space, we need to focus on a subclass of predicates that has more structure.

13.2.1 Finite Enumerations

A finite enumeration of points from X is represented by a list. A point x is in a finite enumeration if there classically exists a point in the list that is equivalent to x. We are not required to know which point in the list is equivalent to x; we only need to know that there is one. An equivalent definition is given by structural recursion on lists.

Definition 13.7.

$$\begin{aligned} x \in \mathsf{nil} &:= \bot \\ x \in \mathsf{cons}(y, l) &:= x \asymp y \,\tilde{\vee}\, x \in l. \end{aligned}$$

We use the classical disjunction (see Definition 2.2) here because the constructive union of two or more points may not be complete [Mandelkern, 1988]. To see why, consider the set defined by the predicate $z \in \mathcal{A} := z \asymp x \vee z \asymp y$ given two real numbers x, y: $\mathbb{R}$. This predicate represents the set $\{x, y\}$. If $\mathcal{A}$ were complete, then $\mathsf{max}\ (x, y) \in \mathcal{A}$. However, this would imply that $\mathsf{max}\ (x, y) \asymp x \vee \mathsf{max}\ (x, y) \asymp y$, which is equivalent to $y \leq x \vee x \leq y$. This is well-known to be unprovable. Hence we cannot always prove that the constructive union of even two points is compact.

Because (constructively) finite sets are not necessarily compact, they cannot be used as a basis for compact sets. However, when the classical disjunction is used in Definition 13.7, the resulting enumerations are always complete and compact.

Two finite enumerations are considered equivalent if they have exactly the same members.

Definition 13.8. $l_1 \asymp l_2 := \forall x. x \in l_1 \Leftrightarrow x \in l_2$.

If X is a stable metric space (recall Definition 9.14), then the space of finite enumerations over X, $\mathfrak{F}(X)$, is also a stable metric space. The Hausdorff metric with the membership predicate defines the distance relation.

Definition 13.9. $\boldsymbol{B}_\varepsilon^{\mathfrak{F}(X)}(l_1, l_2) := \boldsymbol{B}_\varepsilon^{X \Rightarrow \star}(\lambda x. x \in l_1, \lambda y. y \in l_2)$.

This distance relation is both closed (Axiom 4) and is compatible with our equivalence relation for finite enumerations (Axiom 5), so this truly is a metric space. The proof that the distance relation is closed relies crucially on stability at one point (see Section 13.2.2).

If X is a located metric space, then $\mathfrak{F}(X)$ is also a located metric space. However, to prove that $\mathfrak{F}(X)$ is a prelength space, I need to assume that X is both a prelength space and a located metric space. I believe that if a classical existential was used in the definition of prelength space (see Section 9.5.5), then one could prove that $\mathfrak{F}(X)$ is a prelength space without assuming that X is a located metric space.

Finite enumerations also form a monad (although I have not verified this in Coq). The $\mathsf{unit} : X \Rightarrow \mathfrak{F}(X)$ function creates an enumeration with a single member. The $\mathsf{join} : \mathfrak{F}(\mathfrak{F}(X)) \Rightarrow \mathfrak{F}(X)$ function returns the finite union of finite enumerations. The $\mathsf{map}\colon (X \Rightarrow Y) \Rightarrow (\mathfrak{F}(X) \Rightarrow \mathfrak{F}(Y))$ function takes a function $f\colon X \Rightarrow Y$ and replaces every element x from a finite enumeration with $f(x)$.

13.2.2 Mixing Classical and Constructive Reasoning

The proof that the distance relation for finite enumerations is closed makes essential use of classical reasoning. Given ε, suppose that $\boldsymbol{B}_\delta^{\mathfrak{F}(X)}(l_1,\, l_2)$ holds whenever $\varepsilon < \delta$. We need to show that $\boldsymbol{B}_\varepsilon^{\mathfrak{F}(X)}(l_1,\, l_2)$ holds. By the definition of the metric, this requires proving (in part) $\forall x \in l_1.\, \tilde{\exists} y \in l_2.\, \boldsymbol{B}_\varepsilon^X(x,\, y)$. From our assumptions, we know that $\forall x \in l_1.\, \tilde{\exists} y \in l_2.\, \boldsymbol{B}_\delta^X(x, y)$ holds for every δ greater than ε. If I had used a constructive existential in the definition of the Hausdorff hemimetric, we would have a problem. Each different value δ could produce a *different* y witnessing $\boldsymbol{B}_\delta^X(x,\, y)$. In order to use the closedness property from X to conclude $\boldsymbol{B}_\varepsilon^X(x,\, y)$, we need a *single* y such that $\boldsymbol{B}_\delta^X(x,\, y)$ holds for all δ greater than ε. Classically we would use the infinite pigeon hole principle to find a single y that occurs infinitely often in the stream of ys produced from $\delta \in \left\{\varepsilon + \frac{1}{n} \,\middle|\, n\colon \mathbb{N}^+\right\}$. Such reasoning does not work constructively. Given an infinite stream of elements drawn from a finite enumeration, there is no algorithm that will determine which one occurs infinitely often.

Fortunately, because I used classical quantifiers in the definition of the Hausdorff metric, we can apply the infinite pigeon hole principle to this problem. We classically know there is some y that occurs infinitely often when $\delta \in \left\{\varepsilon + \frac{1}{n} \,\middle|\, n\colon \mathbb{N}^+\right\}$, even if we do not know which one. For such y, $\boldsymbol{B}_\delta^X(x,\, y)$ holds for δ arbitrarily close to ε, and therefore $\boldsymbol{B}_\delta^X(x, y)$ must hold for all δ greater than ε. By the closedness property for X, $\boldsymbol{B}_\varepsilon^X(x, y)$ holds as required. The other half of the definition of $\boldsymbol{B}_\varepsilon^{\mathfrak{F}(X)}(l_1, l_2)$ is handled similarly.

Recall from Chapter 2 that the classical fragment of constructive logic requires that proof by contradiction hold for atomic formulas in order to deduce the rule $\neg\neg\varphi \Rightarrow \varphi$. Because $\boldsymbol{B}_\varepsilon^X(x,\, y)$ is a parameter, we do not know if it is constructed out of classical connectives. To use the classical reasoning needed to apply the pigeon hole principle, we assume that $\neg\neg\boldsymbol{B}_\varepsilon^X(x,\, y) \Rightarrow \boldsymbol{B}_\varepsilon^X(x,\, y)$ holds. This is the crucial point where stability of the metric for X is used.

13.3 Metric Space of Compact Sets

Completing the metric space of finite enumerations yields a metric space of compact sets.

Definition 13.10. $\mathfrak{K} := \mathfrak{C} \circ \mathfrak{F}$.

The idea is that every compact set can be represented as a limit of finite enumerations that approximate it. In order for a compact set to be considered a set, we need to define a membership relation. The membership is not over X, because compact sets are supposed to be complete and X may not be a complete space itself. Instead, membership is over $\mathfrak{C}(X)$, and it is defined for $x{:}\, \mathfrak{C}(X)$ and $\mathcal{S}{:}\, \mathfrak{K}(X)$ as follows:

Definition 13.11. $x \in \mathcal{S} := \forall \varepsilon_1\, \varepsilon_2.\, \tilde{\exists} y \in \mathcal{S}(\varepsilon_2).\, \boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}(x(\varepsilon_1), y)$.

A point is considered to be a member of a compact set $\mathcal{S}$ if it is arbitrarily close to being a member of all approximations of $\mathcal{S}$. Thus $\mathfrak{K}(X)$ represents the space of compact subsets of $\mathfrak{C}(X)$.

13.3.1 Correctness of Compact Sets

Bishop and Bridges define a compact set in a metric space X as a set that is complete and totally bounded [Bishop and Bridges, 1985]. In my framework, we say a predicate $\mathcal{A}{:}\, X \Rightarrow \star$ is *complete* if whenever $x{:}\, \mathfrak{C}(X)$ is made from approximations in $\mathcal{A}$, then x is in $\mathcal{A}$:

$$\forall x{:}\, \mathfrak{C}(X).\, (\forall \varepsilon. x(\varepsilon) \in \mathcal{A}) \Rightarrow \exists z \in \mathcal{A}.\, \hat{z} \asymp x$$

A set $\mathcal{B}{:}\, X \Rightarrow \star$ is *totally bounded* if there is an ε-net for every $\varepsilon{:}\, \mathbb{Q}^+$. An ε-net is a list of points l from $\mathcal{B}$ such that for every $x \in \mathcal{B}$ there (constructively) exists a point z that is constructively in l and $\boldsymbol{B}^Y_\varepsilon(x, z)$. Bishop and Bridges use the strong constructive definition of list membership that tells which member of the list the value is.

$$\forall \varepsilon{:}\, \mathbb{Q}^+. \exists l{:}\, \mathsf{list}\, X. (\forall x \in l. x \in \mathcal{B}) \wedge (\forall x \in \mathcal{B}. \exists z \in l.\, \boldsymbol{B}^Y_\varepsilon(x, z))$$

Does my definition of compact sets correspond with Bishop and Bridges's definition? The short answer is yes, but there is a small caveat. My definition of metric space is more general than the one that Bishop and Bridges use. Bishop and Bridges require a distance function $d{:}\, X \Rightarrow X \Rightarrow \mathbb{R}$. My more liberal definition of metric space does not have this requirement. I have verified that my definition of compact is the same as Bishop and Bridges's, assuming that X is a located metric (see Definition 9.12). If a metric space has a distance function, then it is a located metric. Thus, my definition of compact corresponds to Bishop and Bridges's definition of compact for those metric spaces that correspond to Bishop and Bridges's definition of metric space.

We prove that the two definitions are the same by constructing an isomorphism between the two structures. This requires constructing two functions, one that maps our compact sets to Bishop-compact sets and another that maps Bishop-compact sets to our compact sets.

The two definitions are fairly similar. The major difference is that the points in Bishop's ε-nets must be members of the compact set. On the other hand, it is possible that none of the points in my ε-approximations are actually part of the compact set.

Defining the function from Bishop-compact sets to our compact sets is fairly easy. Given a Bishop-compact set $\mathcal{S}$, we can define the ε-approximation by first finding a $\frac{\varepsilon}{2}$-net, and then approximating each of the points in the net to within $\frac{\varepsilon}{2}$. The result is a list of points in X that when interpreted as a finite enumeration is within ε of the compact set that $\mathcal{S}$ represents. These approximations are coherent; therefore, the function producing these finite sets is a regular function, so it is a compact set.

Computing the Bishop-compact set given a compact set $\mathcal{S}$ is more difficult. We need to prove that this set is complete and totally bounded. Proving that it is complete is easy. Proving that it is totally bounded is the difficult part. We need to construct an ε-net that contains points from S, but our approximations generally do not lie inside $\mathcal{S}$. To construct the ε-net, we start by computing a δ_1-approximation as our seeds. For each seed, we will construct a new point in $\mathfrak{C}(X)$ that is a member of the compact set $\mathcal{S}$, and within γ of its seed. By choosing δ_1 and γ such that $\delta_1 + \gamma \leq \varepsilon$, the list of constructed points will be an ε-net.

Given a point from a δ_1-approximation of $\mathcal{S}$, we would like to select a point inside the $k\delta_1$-approximation of $\mathcal{S}$ that is within $\delta_1 + k\delta_1$ of the first point. Classically we know such a point exists because the approximations are coherent (Definition 13.5); however, we don't know which point it is. Without a decidable metric (Definition 9.11), we cannot search the list to find the point.

To overcome this we use the assumption that X is a located metric (Definition 9.12). Using locatedness we still cannot search for a point within $\delta_1 + k\delta_1$, but we can search for a point within $\delta_1 + k\delta_1 + \delta_2$.

With a suitable point in the $k\delta_1$-approximation of $\mathcal{S}$ found, we can repeat the above procedure to find a point in the $k^2\delta_1$-approximation that is within $k\delta_1 + k^2\delta_1 + k\delta_2$ of our second point. By repeating this procedure and taking the limit, a point $y\colon \mathfrak{C}(X)$ is constructed that is within $\frac{\delta_1 + k\delta_1 + \delta_2}{1-k}$ of the initial point from the δ_1-approximation. Furthermore, y is in the compact set because it is constructed from points that are drawn from finer and finer approximations of $\mathcal{S}$.

Using the above procedure we can map every point in the in the δ_1-approximation to new point from $\mathcal{S}$. These new points form a $\left(\delta_1 + \delta_2 + \frac{\delta_1 + k\delta_1 + \delta_2}{1-k}\right)$-net because for every point x in the set $\mathcal{S}$ we can find some point z in the δ_1-approximation that is within $\delta_1 + \delta_2$ of x and the point in our net generated from the seed z is within $\frac{\delta_1 + k\delta_1 + \delta_2}{1-k}$ of z. By choosing suitable values for δ_1, δ_2, and k, we can ensure that $\left(\delta_1 + \delta_2 + \frac{\delta_1 + k\delta_1 + \delta_2}{1-k}\right) \leq \varepsilon$, and hence our net is an ε-net. For concreteness, in my construction I chose $k := \frac{1}{4}$, $\delta_1 := \frac{\varepsilon}{5}$, and $\delta_2 := \frac{\varepsilon}{5}$.

Thus our compact sets are interpreted as a totally bounded and complete set, and hence correspond to a Bishop and Bridges style compact set. To complete the proof that our two notions are isomorphic, we need to show that composing the functions to and from Bishop-compact sets and to and from my compact sets yield the identity function. This proof is straightforward.

13.3.2 Distribution of $\mathfrak{F}$ over $\mathfrak{C}$

Recall that composition of two monads, $\mathfrak{A} \circ \mathfrak{B}$, forms a monad when there is a distribution function $\mathsf{dist} : \mathfrak{B}(\mathfrak{A}(X)) \to \mathfrak{A}(\mathfrak{B}(X))$ satisfying the laws given in Section 12.1.9. For compact sets, $\mathfrak{K}(X) := (\mathfrak{C} \circ \mathfrak{F})(X)$, the distribution function $\mathsf{dist} : \mathfrak{F}(\mathfrak{C}(X)) \to \mathfrak{C}(\mathfrak{F}(X))$ is defined by

Definition 13.12. $\mathsf{dist}(l) := \lambda\varepsilon.\, \mathsf{map}_{\mathfrak{F}}(\lambda x. x(\varepsilon))(l)$.

This function interprets a finite enumeration of points from $\mathfrak{C}(X)$ as a compact set. Thus $\mathfrak{K}$ is also a monad.[13.1]

13.3.3 Compact Image

One can define the compact image of a compact set $\mathcal{S}\colon \mathfrak{K}(X)$ under a uniformly continuous function $\check{f} : \mathfrak{C}(X) \to \mathfrak{C}(Y)$, by recalling that $\check{f}$ denotes $\mathsf{bind}(f)$, and by noting that applying f to every point in a finite enumeration is a uniformly continuous function, $\mathsf{map}_{\mathfrak{F}}(f)\colon \mathfrak{F}(X) \to \mathfrak{F}(\mathfrak{C}(Y))$. Composing this with dist yields a uniformly continuous function from finite enumerations $\mathfrak{F}(X)$ to compact sets $\mathfrak{K}(Y)$. Using bind (Definition 10.18), this function can be lifted to operate on $\mathfrak{K}(X)$. The result is the *compact image* function.

Definition 13.13. $f \upharpoonright \mathcal{S} := \mathsf{bind}_{\mathfrak{C}}(\mathsf{dist} \circ \mathsf{map}_{\mathfrak{F}}(f))(\mathcal{S})$.

Although Bishop and Bridges would agree that the result of this function is compact, they would not say that it is the image of $\mathcal{S}$ because one cannot constructively prove

$$\forall y.\, y \in f \upharpoonright \mathcal{S} \Rightarrow \exists x \in \mathcal{S}.\, \check{f}(x) \asymp y.$$

Instead, they would consider $f \upharpoonright \mathcal{S}$ to be the closure of the image of $\mathcal{S}$ under f. In constructive analysis, the image of $\mathcal{S}$ under f, namely $\lambda y. \exists x \in \mathcal{S}.\, \check{f}(x) \asymp y$, is not necessarily compact [Mandelkern, 1988]. Consider $f\colon \mathsf{Bool} \Rightarrow \mathbb{R}$ where $f(\mathsf{true}) := x$ and $f(\mathsf{false}) := y$ given two real numbers $x y\colon \mathbb{R}$. The Booleans are compact (using the discrete topology), but the image of f may not be provably compact for the reason given in Section 13.2.1.

[13.1] I have not yet verified the distribution laws for the $\mathfrak{F}$ and $\mathfrak{C}$ monads, but the laws are not needed for any of the proofs in my work.

I believe one can prove (but I have not verified this in Coq) the classical statement

$$\forall y. y \in f \restriction \mathcal{S} \Rightarrow \tilde{\exists} x \in \mathcal{S}. \check{f}(x) \asymp y.$$

When $\check{f}$ is injective, as it will be for our graphing example in Section 13.4.1, the constructive existential statement holds.

13.4 Plotting Functions

There are many examples of constructively compact sets. This section illustrates one application of compacts sets: plotting functions.

13.4.1 Graphing Functions

Given a uniformly continuous function $\check{f} : \mathfrak{C}(X) \to \mathfrak{C}(Y)$ and a compact set $\mathcal{D}\colon \mathfrak{K}(X)$, the graph of the function over $\mathcal{D}$ is the set of points $\{(x, \check{f}(x)) | x \in \mathcal{D}\}$. This graph can be constructed as a compact set $\boldsymbol{G}(\mathcal{D}, f)\colon \mathfrak{K}(X \times Y)$. A single point is graphed by the function $\boldsymbol{g}(f)(x) := \langle \hat{x}, f(x) \rangle$. This function is uniformly continuous, $\boldsymbol{g}(f)\colon X \to \mathfrak{C}(X \times Y)$. The graph $\boldsymbol{G}(\mathcal{D}, f)$ is defined as the compact image of $\mathcal{D}$ under $\boldsymbol{g}(f)$.

Definition 13.14. $\boldsymbol{G}(\mathcal{D}, f) := \boldsymbol{g}(f) \restriction \mathcal{D}$.

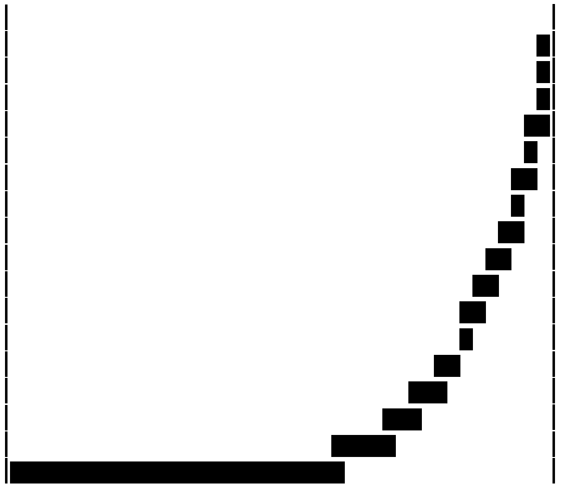
13.4.2 Rasterizing Compact Sets

Given a compact set in the plane $\mathcal{S}\colon \mathfrak{K}(\mathbb{Q} \times \mathbb{Q})$, we can draw an image of it, or rather we can plot an approximation of it. This process consists of two steps. The first step is to compute an ε-approximation $l := \mathcal{S}(\varepsilon)$. The finite enumeration l is a list of rational coordinates. The next step is to move these points around so that all the points lie on a raster. A raster is simply a two dimensional matrix of Booleans. Given coordinates for the top-left and bottom-right corners, a raster can be interpreted as a finite enumeration. Using advanced notation features in Coq, a raster can be displayed inside the proof assistant [Stein, 2003]. Most importantly, when the constructed raster is interpreted, it is provably close to the original compact set.

13.4.3 Plotting the Exponential Function

Given a uniformly continuous function $\check{f} : \mathbb{R} \to \mathbb{R}$ and an interval $[a, b]$, the graph of $\check{f}$ over this compact interval is a compact set. The graph is an ideal mathematical curve. This graph can then be plotted yielding a raster that when interpreted as a finite enumeration is provably close to the ideal mathematical curve.

```
ball (m:=Compact stableQ2) (324 # 2592)
  graphCR (boundBelow 0 ∘ boundAbove 3 ∘ exp_bound_uc 1) [- (6) .. 1]
  (Cunit
     (- (6), 3)⇱
```



```
                                                        ⇲(1, 0))
```

Figure 13.1. A theorem in Coq stating that a plot on a 42 by 18 raster is close to the graph of the exponential function on $[-6, 1]$.

Figure 13.1 shows a theorem in Coq that states the (ideal mathematical) graph of the exponential function (which is uniformly continuous on $(-\infty,\ 1])$ restricted to the range $[0,\ 3]$ on the interval $[-6,\ 1]$ is within $\frac{324}{2592}$ (which is equivalent to $\frac{1}{8}$) of the finite set represented by raster shown with the top-left corner mapped to $(-6, 3)$ and the bottom-right corner mapped to $(1, 0)$. The raster is 42 by 18, so, by considering the domain and range of the graph, each pixel represents a $\frac{1}{6}$ by $\frac{1}{6}$ square. The error between the plot and the graph must always be greater than half a pixel. I chose an ε that produces a graph with an error of $\frac{3}{4}$ of a pixel. In this case $\frac{3}{4} \cdot \frac{1}{6} = \frac{1}{8}$, which is the error given in the theorem.

There is one small objection to this image. Each block in the picture represents an infinitesimal mathematical point lying at the center of the block, but the block appears as a square the size of the pixel. This could be fixed by interpreting each block as a filled square instead of as a single point. This change would simply add an additional $\frac{1}{2}$ pixel to the error term. This has not been done yet in this early implementation.

13.5 Alternative Hausdorff Metric Definition

There is another possible definition for the Hausdorff metric. One could define the Hausdorff hemimetric as

$$\boldsymbol{H'}^{X \Rightarrow \star}_{\varepsilon}(\mathcal{A}, \mathcal{B}) := \forall x \in \mathcal{A}. \forall \delta. \exists y \in \mathcal{B}. \boldsymbol{B}^{X}_{\varepsilon+\delta}(x, y).$$

The extra flexibility given by the δ term also allows one to conclude that there is some $y \in \mathcal{B}$ that is within ε of x without telling us which one (again, it may be the case that we cannot know which y is the one). Our original definition $\boldsymbol{H}_{\varepsilon}^{X\Rightarrow\star}(\mathcal{A},\mathcal{B})$ is implied by $\boldsymbol{H'}_{\varepsilon}^{X\Rightarrow\star}(\mathcal{A},\ \mathcal{B})$; however, the alternative definition yields more constructive information.

The two definitions are equivalent under mild assumptions. When X is a located metric, then $\boldsymbol{H}_{\varepsilon}^{X\Rightarrow\star}(\mathcal{A},\mathcal{B}) \Rightarrow \boldsymbol{H'}_{\varepsilon}^{X\Rightarrow\star}(\mathcal{A},\mathcal{B})$. This is a very common case and allows us to recover the constructive information in the $\boldsymbol{H'}$ version from the $\boldsymbol{H}$ version. This was implicitly done in the proof from Section 13.3.1

The constructive existential in the definition of $\boldsymbol{H'}$ would make the resulting metric not provably stable. It is somewhat unclear which version is the right definition for the constructive Hausdorff metric. The key deciding factor for me was that I had declared the distance relation to be in the `Prop` universe (see Section 2.3.2). This means that even if I used the $\boldsymbol{H'}$ definition in the Hausdorff metric, its information would not be allowed by Coq to construct values in `Set`. For this reason, I chose the $\boldsymbol{H}$ version with the classical quantifiers for the definition of the Hausdorff metric. Values with classical existential quantifier type have no information in them and naturally fit into the `Prop` universe.

13.6 Compactness and Computability

Warning: This section contains a philosophical discussion of the merits of constructive language over the language of classical computation theory. The mathematics used in this section serve to illustrate my position. I have no guarantees about the accuracy of the images; I have made no formal, nor informal proofs of the convergence of Equation 13.1 below, and I cannot guarantee that the constructive formula in Equation 13.2 is correct. However, I believe that the images are reasonably accurate and the formulas are correct and could be formalized.

Is there a computer program that outputs 1 if the Riemann hypothesis is true and outputs 0 if the Riemann hypothesis is false? This is a common puzzle posed in computer science courses on the theory of computation. The classical solution to this puzzle is, Yes! Consider two programs, one that outputs 0 and another that outputs 1. One of these two programs satisfies the specification, so a computer program does exist even if we don't know which one.

This sort of paradoxical result is due to the language of classical logic that is used in classical computation theory. If we phrase the same question in terms of constructive mathematics, we ask, "does RH $\vee$ $\neg$RH hold?" where RH is the statement of the Riemann hypothesis. Now we must answer, "We don't know," which is the expected answer.

One might dismiss this as a silly example that everyone understands after encountering it once and it is no longer a concern after that. However, this problem with the language of classical computation theory can potentially appear in far more subtle places.

Consider, for example, Braverman and Yampolsky's theorem that all filled Julia sets are computable [Braverman and Yampolsky, 2009]. Yet, there are Julia sets that we do not know how to compute. Let us focus on quadratic polynomials $p_c(z) := z^2 + c$ for this example. Let $\mathcal{K}_c$ be the filled Julia set for the polynomial p_c. Classically, the set $\mathcal{K}_c$ is a compact subset of $\mathbb{C}$ that can be defined as follows:

$$z \in \mathcal{K}_c := \exists M. \forall n{:}\, \mathbb{N}. \left| p_c^{(n)}(z) \right| \leq M$$

This means that the set $\mathcal{K}_c$ contains all the complex numbers z such that all the iterations of z under p_c remain bounded. Classically, a compact set $\mathcal{A}$ is said to be a computable subset of the real plane when there exists a computable function $f{:}\, \mathbb{N} \Rightarrow \mathfrak{F}(\mathbb{Q}^2)$, such that $d(f(n), \mathcal{A}) \leq 2^{-n}$ using the Hausdorff metric.[13.2] In other words, $\mathcal{A}$ can be drawn on any arbitrarily high resolution raster display by a single program (note the similarity with my definition of constructively compact sets). When we say $\mathcal{K}_c$ is computable, we are identifying $\mathbb{C}$ with the real plane in the usual way.

We are going to construct an example of a computable filled Julia set that we do not know how to compute. We are going to do this by creating a Brouwerian counterexample [Richman, 1983]. Let $\forall n{:}\ \mathbb{N}. \vartheta(n)$ be your favourite unsolved Π_1 problem. For concreteness, I will take $\vartheta(n) := \neg\varsigma(n)$ where $\varsigma(n)$ states that when n is considered as an ASCII file, it contains a valid Coq proof of `False` from no axioms and without using the tactic language (only (co-)inductive declarations and definitions allowed). This means that $\forall n{:}\ \mathbb{N}. \vartheta(n)$ states that Coq is consistent. I have chosen this example because, by the second incompleteness theorem (see Section 7.6), Coq cannot prove this statement (unless Coq is inconsistent).

Consider the sequence $\varrho{:}\ \mathbb{N} \Rightarrow \mathbb{R}$ where $\varrho_0 := \frac{1}{4}$ and ϱ_{n+1} is the unique solution in $[0, \varrho_n]$ such that

$$p^{(n+1)}_{\frac{1}{4} + \varrho_{n+1}}\left(\frac{3}{8}\right) = \frac{5}{8}. \tag{13.1}$$

The sequence ϱ is strictly decreasing and has a limit of 0. The sequence is constructive because $p^{(n)}_{\frac{1}{4} + \varrho_n}\left(\frac{3}{8}\right) - \frac{5}{8}$ is a polynomial in ϱ_n, so its roots can be constructed [Geuvers et al., 2002].

Define $\epsilon{:}\, \mathbb{R}$ to be the regular function

$$\epsilon(\varepsilon) := \mathsf{inf}\ (\{\varrho_0\} \cup \{\varrho_{n+1} \mid \varepsilon < \varrho_{n+1} \wedge \forall m. m \leq n \Rightarrow \vartheta(m)\}).$$

[13.2] This definition of computable is somewhat different than the one given by Braverman and Yampolsky, but it is equivalent.

The value ϵ is a well-defined constructive real number, because $\{\varrho_{n+1} \mid \varepsilon < \varrho_{n+1}\}$ is a finite set, and $\forall m.m \leq n \Rightarrow \vartheta(m)$ is a decidable predicate. If Coq is consistent, then $\epsilon \asymp 0$. If Coq is inconsistent, then $\epsilon \asymp \varrho_n$ where n is an ASCII encoding of a proof of `False`. According to Braverman and Yampolsky's theorem, $\mathcal{K}_{\frac{1}{4}+\epsilon}$ is a computable filled Julia set, because all filled Julia sets are computable. Consider Figure 13.2. If Coq is consistent, then $\mathcal{K}_{\frac{1}{4}+\epsilon}$ is approximately the image on the left, and if Coq is inconsistent, then $\mathcal{K}_{\frac{1}{4}+\epsilon}$ is approximately the image on the right.[13.3] However, just like the previous Riemann hypothesis puzzle, we do not know which image is correct. The language of classical computation theory misleads us again, and this time it is not with a result that is trivial enough to dismiss as a harmless puzzle.

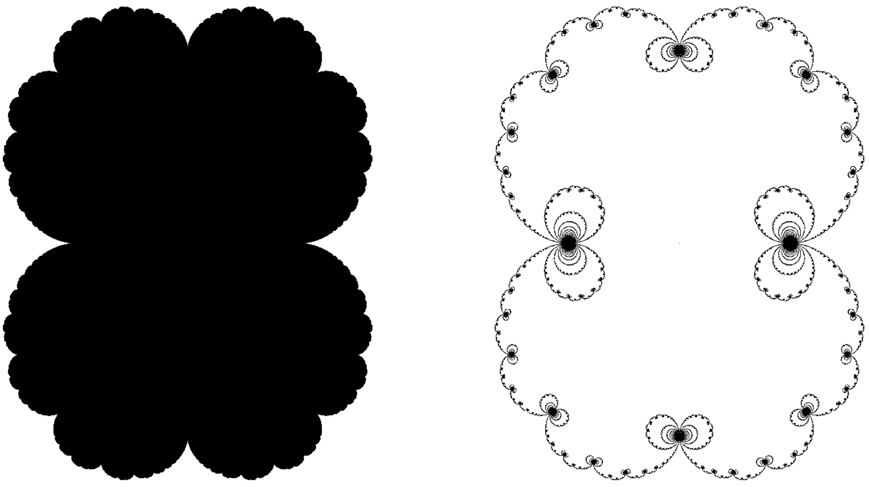


Figure 13.2. One of these images represents the computable set $\mathcal{K}_{\frac{1}{4}+\epsilon}$, but which one?

Of course, Braverman and Yampolsky do not mislead their readers. They make themselves perfectly clear that the filled Julia sets are not *uniformly* computable. However, the fact that they need to go through hoops to clarify their result serves to illustrate the deficiency of the language of classical computation.

[13.3] These images were generated by the program Mandel 5.2 [Jung, 2008], which does not use a correct algorithm. It instead estimates the image and uses floating point calculations. I make no guarantee of the accuracy of these images, but they are presumably reasonably accurate. If one used the algorithms given by Braverman and Yampolsky, then "loops" "inside" the right hand image would not get drawn until ϵ had been computed to sufficient accuracy to prove that Coq is inconsistent. In order to generate a reasonable image for this case, I have carefully chosen the sequence ϱ_n so that $\mathcal{K}_{\frac{1}{4}+\varrho_n}$ converges more uniformly [Douady, 1994]. This allows me to approximate $\mathcal{K}_{\frac{1}{4}+\varrho_n}$ for large n by $\mathcal{K}_{\frac{1}{4}+\varrho_n}$ for small n. I have not computed the modulus of convergence and simply used an n that seems to be large enough to give accurate results for the resolution used.

13.6.1 A Constructive Approach to Filled Julia Sets

Suppose that Braverman and Yampolsky used the language of constructive mathematics instead. How would they present their results? Instead of asking if a compact set was classically computable, they would simply ask, "Is every filled Julia set (constructively) compact?". Instead of concluding, "Yes", their work would result in a theorem such as:

$$\begin{aligned} \forall c{:}\,\mathbb{C}.\,(\quad & p_c \text{ has an attracting orbit} \\ \lor \quad & p_c \text{ has a parabolic orbit} \\ \lor \quad & p_c \text{ has a Siegel orbit} \\ \lor \quad & p_c \text{ has a Cremer orbit} \\ \lor \quad & \text{all orbits of } p_c \text{ are repelling } \;) \Rightarrow \mathcal{K}_c \text{ is compact} \end{aligned} \tag{13.2}$$

The constructive result clearly states, right in the theorem, exactly the extra information beyond the value of c that is needed to compute $\mathcal{K}_c$. One does not need to go fishing through the proofs to get this information (as is needed for the classical theorems). Note that a proof of the big disjunction in the hypothesis of the theorem contains more information than simply which clause holds. Each clause contains constructive details about the particular type of orbit.

According to classical mathematics, the big disjunction is true and can be removed from the hypothesis. However, because of the discontinuity of $\lambda c.\ \mathcal{K}_c$ at various points, the theorem $\forall c{:}\,\mathbb{C}.\,\mathcal{K}_c$ is compact cannot be constructed.

The constructive theorem cannot be used to prove that $\mathcal{K}_{\frac{1}{4}+\epsilon}$ is compact in Coq. Coq cannot prove $\left(p_{\frac{1}{4}+\epsilon} \text{ is parabolic} \lor \text{all orbits of } p_{\frac{1}{4}+\epsilon} \text{ are repelling}\right)$—all the other clauses are provably false—because such a theorem would contradict the second incompleteness theorem (unless Coq is inconsistent). This corresponds to our understanding that we do not know how to compute $\mathcal{K}_{\frac{1}{4}+\epsilon}$. However, we can prove

$$(\text{Coq is consistent} \lor \text{Coq is inconsistent}) \Rightarrow \mathcal{K}_{\frac{1}{4}+\epsilon} \text{ is compact}.$$

This example illustrates how the constructive formulation better conveys computability results than the language of classical computability theory.

Chapter 14
Conclusion

The major goal of this part was to develop an effective implementation of real numbers that is efficient enough to be used for computation inside a proof assistant. I approached this problem by first creating a generic completion operation on metric spaces. I needed to create a completion operation that was independent of the real numbers. This naturally led to defining metric spaces using a distance relation (see Chapter 9). This definition is more general than the one given by Bishop and Bridges [Bishop and Bridges, 1985].

I showed that the completion operation has a familiar monad structure (see Chapter 10), and I used these monad operations to define functions on complete metric spaces. The monad operations nicely separate the problem of defining functions on complete spaces into two separate parts: defining the operation on the underlying dense space and then proving that the definition is uniformly continuous.

I defined the real numbers as the completion of the rational numbers (see Chapter 11). I defined various elementary functions on the real numbers and showed that they can be implemented efficiently enough to allow real numbers to be approximated inside the Coq proof assistant. I used my efficient real numbers to build a simple tactic for proving strict equalities between real number expressions.

With the major goal achieved, I then looked at other applications of the completion operation. I defined the integrable functions as the completion of the rational step functions under the L^1 metric (see Chapter 12). I defined Riemann integration of uniformly continuous functions by converting them to integrable functions. We saw that this definition was easy to extend to define Stieltjes integration.

The other application of the completion operation was for compact sets. I defined the compact sets as the completion of the finite sets under the Hausdorff metric (see Chapter 13). The generated compact sets were effective and could be plotted inside the Coq proof assistant. I showed how to plot uniformly continuous functions and use these plots in theorems to prove that the plots accurately represent the graph of the function.

The development of compact sets also showed how one can mix classical and constructive reasoning. I used the classical infinite pigeon hole principle to prove the closedness property of the metric (see Section 13.2.2). I estimate that 80% of the time developing constructive analysis, one is in a context where classical reasoning can apply. For example, when your (sub-)goal is to prove that $x \asymp y$ for two real numbers x and y, then that goal is stable and you are allowed to perform classical case analysis on $y \leq z \mathrel{\tilde{\vee}} z \leq y$ or any other classical disjunction or existential (see Theorem 2.3 and Theorem 2.4). Even when developing constructive theorems, one usually provides a witness early in the proof and classical reasoning can be used for the remaining goal.

Bibliography

[Ait-Mohamed, 2008] Ait-Mohamed, O., editor (2008). *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montréal, Canada, August 18-21, 2008, Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer.

[Apple and Haken, 1976] Apple, K. and Haken, W. (1976). Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82:711–712.

[Aydemir et al., 2005] Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, N. J., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. (2005). Mechanized metatheory for the masses: The POPLmark challenge. In [Hurd and Melham, 2005].

[Barendregt and Geuvers, 2001] Barendregt, H. and Geuvers, H. (2001). Proof-assistants using dependent type systems. In *Handbook of Automated Reasoning*, pages 1149–1238. Elsevier Science Publishers B. V., Amsterdam, The Netherlands.

[Bauer and Taylor, 2008] Bauer, A. and Taylor, P. (2008). The Dedekind reals in abstract stone duality. to appear.

[Beck, 1969] Beck, J. (1969). Distributive laws. In Eckman, B., editor, *Seminar on Triples and Categorical Homology Theory*, number 80 in Lecture Notes in Mathematics, pages 119–140. Springer, Berlin.

[Berger, 2008] Berger, U. (2008). From coinductive proofs to exact real arithmetic. Draft.

[Bertot, 2007] Bertot, Y. (2007). Affine functions and series with co-inductive real numbers. *Mathematical Structures in Computer Science*, 17(1):37–63.

[Bird and Meertens, 1998] Bird, R. and Meertens, L. (1998). Nested datatypes. In Jeuring, J., editor, *Proceedings 4th International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422, pages 52–67. Springer-Verlag, Berlin.

[Bishop and Bridges, 1985] Bishop, E. and Bridges, D. (1985). *Constructive Analysis*. Number 279 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag.

[Brattka, 2003] Brattka, V. (2003). The emperor's new recursiveness: The epigraph of the exponential function in two models of computability. In Ito, M. and Imaoka, T., editors, *Words, Languages & Combinatorics III*, pages 63–72, River Edge, NJ. World Scientific Publishing. ICWLC 2000, Kyoto, Japan, March 14–18, 2000.

[Braverman and Yampolsky, 2009] Braverman, M. and Yampolsky, M. (2009). *Computability of Julia Sets*. Number 23 in Algorithms and Computation in Mathematics. Springer. `info:arxiv/math/0610340`.

[Bromage and Jäger, 2006] Bromage, A. J. and Jäger, T. (2006). Lambdabot. `http://www.cse.unsw.edu.au/~dons/lambdabot.html`.

[Burago et al., 2001] Burago, D., Burago, Y., and Ivanov, S. (2001). *A Course in Metric Geometry*, volume 33 of *Graduate Studies in Mathematics*. American Mathematical Society.

[Burris, 1997] Burris, S. N. (1997). Logic for mathematics and computer science: Supplementary text. `http://www.math.uwaterloo.ca/~snburris/htdocs/LOGIC/stext.html`.

[Capretta, 2005] Capretta, V. (2005). General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18. `http://www.lmcs-online.org/ojs/viewarticle.php?id=55`.

[Caprotti and Oostdijk, 2001] Caprotti, O. and Oostdijk, M. (2001). Formal and efficient primality proofs by use of computer algebra oracles. *Journal of Symbolic Computation*, 32(1–2):55–70. Special Issue on Computer Algebra and Mechanized Reasoning.

[Ciaffaglione, 2003] Ciaffaglione, A. (2003). *Certified reasoning on Real Numbers and Objects in Co-inductive Type Theory*. PhD thesis, Dipartimento di Matematica e Informatica Università di Udine, Italy.

[Coen, 2004] Coen, C. S. (2004). A semi-reflexive tactic for (sub-)equational reasoning. In Filliâtre, J.-C., Paulin-Mohring, C., and Werner, B., editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 98–114. Springer.

[CoRN, 2008] CoRN (2008). Constructive Coq repository at Nijmegen. `http://corn.cs.ru.nl/`.

[Cruz-Filipe, 2003] Cruz-Filipe, L. (2003). A constructive formalization of the fundamental theorem of calculus. In Geuvers, H. and Wiedijk, F., editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*, pages 108–126. Springer–Verlag.

[Cruz-Filipe, 2004] Cruz-Filipe, L. (2004). *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. PhD thesis, University of Nijmegen.

[Cruz-Filipe et al., 2004] Cruz-Filipe, L., Geuvers, H., and Wiedijk, F. (2004). C-CoRN: the constructive Coq repository at Nijmegen. In Asperti, A., Bancerek, G., and Trybulec, A., editors, *Mathematical Knowledge Management, Third International Conference, MKM 2004*, volume 3119 of *LNCS*, pages 88–103. Springer–Verlag.

[Cruz-Filipe and Letouzey, 2006] Cruz-Filipe, L. and Letouzey, P. (2006). A large-scale experiment in executing extracted programs. *Electronic Notes in Theoretical Computer Science*, 151(1):75–91.

[Cruz-Filipe and Spitters, 2003] Cruz-Filipe, L. and Spitters, B. (2003). Program extraction from large proof developments. In Basin, D. and Wolff, B., editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 205–220. Springer–Verlag.

[Despeyroux and Hirschowitz, 1994] Despeyroux, J. and Hirschowitz, A. (1994). Higher-order abstract syntax with induction in Coq. In *LPAR '94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, London, UK. Springer-Verlag.

[Douady, 1994] Douady, A. (1994). Does a Julia set depend continuously on the polynomial? *Complex dynamical systems: The mathematics behind the Mandelbrot set and Julia sets*, 49:91–138.

[Edalat, 1999] Edalat, A. (1999). Numerical integration with exact real arithmetic. In *Automata, Languages and Programming, 26th International Colloquium, ICALP'99 Prague, Czech 227 Republic, July 11-15, 1999, Proceedings, volume 1644 of Lecture Notes in Computer Science*, pages 90–104, London, UK. Springer-Verlag.

[Frege, 1879] Frege, G. (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelprache des reinen Denkens*. Verlag Nebert, Halle.

[Geuvers and Niqui, 2002] Geuvers, H. and Niqui, M. (2002). Constructive reals in Coq: Axioms and categoricity. In Callaghan, P., Luo, Z., McKinna, J., and Pollack, R., editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 79–95. Springer.

[Geuvers et al., 2002] Geuvers, H., Wiedijk, F., and Zwanenburg, J. (2002). A constructive proof of the fundamental theorem of algebra without using the rationals. In *TYPES '00: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 96–111, London, UK. Springer-Verlag.

[Gonthier, 2005] Gonthier, G. (2005). A computer-checked proof of the four colour theorem. Technical report, Microsoft Research Cambridge.

[Grégoire, 2007] Grégoire, B. (2007). personal correspondence.

[Grégoire and Leroy, 2002] Grégoire, B. and Leroy, X. (2002). A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press.

[Gödel, 1931] Gödel, K. (1931). Ueber Formal Unentscheidbare sätze der Principia Mathematica und Verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198. english translation: On Formally Undecidable Propositions of Principia Mathematica and Related Systems I, Oliver & Boyd, London, 1962.

[Hales, 2002] Hales, T. C. (2002). A computer verification of the Kepler conjecture. In *Proceedings of the International Congress of Mathematicians, Vol. III (Beijing, 2002)*, pages 795–804, Beijing. Higher Ed. Press.

[Hales, 2006] Hales, T. C. (2006). The flyspeck project fact sheet. `http://www.math.pitt.edu/~thales/flyspeck/`.

[Harrison, 1998a] Harrison, J. (1998a). Formalizing basic first order model theory. In Grundy, J. and Newey, M., editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *Lecture Notes in Computer Science*, pages 153–170, Canberra, Australia. Springer-Verlag.

[Harrison, 1998b] Harrison, J. (1998b). *Theorem Proving with the Real Numbers*. Springer-Verlag.

[Harrison, 2000] Harrison, J. (2000). The HOL-Light manual. `http://www.cl.cam.ac.uk/users/jrh/hol-light/`.

[Hirschowitz and Maggesi, 2006] Hirschowitz, A. and Maggesi, M. (2006). A solution for the POPLmark challenge. `http://web.math.unifi.it/users/maggesi/fsub/`.

[Hodel, 1995] Hodel, R. E. (1995). *An Introduction to Mathematical Logic*. PWS Publishing Company, Boston.

[Hurd and Melham, 2005] Hurd, J. and Melham, T. F., editors (2005). *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*. Springer.

[Jones, 1993] Jones, C. (1993). Completing the rationals and metric spaces in LEGO. In *Papers presented at the second annual Workshop on Logical environments*, pages 297–316, New York, NY, USA. Cambridge University Press.

[Jones and Duponcheel, 1993] Jones, M. P. and Duponcheel, L. (1993). Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University.

[Julien, 2008] Julien, N. (2008). Certified exact real arithmetic using co-induction in arbitrary integer base. In *Functional and Logic Programming Symposium (FLOPS)*, LNCS. Springer.

[Jung, 2008] Jung, W. (2008). Mandel: software for real and complex dynamics. `http://www.mndynamics.com/indexp.html`.

[Kaliszyk and O'Connor, 2009] Kaliszyk, C. and O'Connor, R. (2009). Computing with classical real numbers. Under consideration for publication in Journal of Formalized Reasoning.

[Kock, 1972] Kock, A. (1972). Strong functors and monoidal monads. *Archiv der Mathematik*, 23:113–120.

[Lambek, 1980] Lambek, J. (1980). From lambda calculus to Cartesian closed categories. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 375–402. Academic Press.

[Lang, 1993] Lang, S. (1993). *Real and Functional Analysis*. Springer.

[Lester, 2008] Lester, D. R. (2008). Real number calculations and theorem proving: Validation and use of an exact arithmetic. In [Ait-Mohamed, 2008], pages 215–229.

[Mandelkern, 1988] Mandelkern, M. (1988). Constructively complete finite sets. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 34(2):97–103.

[Marche, 2005] Marche, C. (2005). Fwd: Question about fixpoint. Coq club mailing list correspondence, `http://pauillac.inria.fr/pipermail/coq-club/2005/001641.html`.

[McBride, 1999] McBride, C. (1999). *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

[McBride and McKinna, 2004a] McBride, C. and McKinna, J. (2004a). Functional pearl: I am not a number–I am a free variable. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9, New York, NY, USA. ACM.

[McBride and McKinna, 2004b] McBride, C. and McKinna, J. (2004b). The view from the left. *Journal of Functional Programming*, 14(1):69–111.

[McBride and Paterson, 2008] McBride, C. and Paterson, R. (2008). Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13.

[Mitchell, 2003] Mitchell, J. C. (2003). *Concepts in Programming Languages*. Cambridge University Press.

[Moggi, 1989] Moggi, E. (1989). Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA. IEEE Press.

[Muñoz and Lester, 2005] Muñoz, C. and Lester, D. (2005). Real number calculations and theorem proving. In [Hurd and Melham, 2005], pages 195–210.

[Niqui, 2004] Niqui, M. (2004). *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. PhD thesis, Radboud Universiteit Nijmegen.

[Niqui and Wiedijk, 2005] Niqui, M. and Wiedijk, F. (2005). The "Many Digits" friendly competition 2005. `http://www.cs.ru.nl/~milad/manydigits`.

[O'Connor, 2005a] O'Connor, R. (2005a). Essential incompleteness of arithmetic verified by Coq. In [Hurd and Melham, 2005], pages 245–260.

[O'Connor, 2005b] O'Connor, R. (2005b). Few digits 0.4.0. `http://r6.ca/FewDigits/`.

[O'Connor, 2005c] O'Connor, R. (2005c). The Gödel-Rosser 1st incompleteness theorem. `http://r6.ca/Goedel20050512.tar.gz`.

[O'Connor, 2007] O'Connor, R. (2007). A monadic, functional implementation of real numbers. *Mathematical Structures in Computer Science*, 17(1):129–159.

[O'Connor, 2008a] O'Connor, R. (2008a). Certified exact transcendental real number computation in Coq. In [Ait-Mohamed, 2008], pages 246–261.

[O'Connor, 2008b] O'Connor, R. (2008b). A computer verified theory of compact sets. In Buchberger, B., Ida, T., and Kutsia, T., editors, *In Proceedings of Austrian-Japanese Workshop on Symbolic Computation in Software Science (SCSS 2008)*, number 08-08 in RISC-Linz Report Series, pages 148–162, Castle of Hagenberg, Austria. RISC.

[O'Connor and Spitters, 2009] O'Connor, R. and Spitters, B. (2009). A computer verified, monadic, functional implementation of the integral. *Theoretical Computer Science*. under consideration.

[Odlyzko and teRiele, 1985] Odlyzko, A. M. and te Riele, H. J. J. (1985). Disproof of the Mertens conjecture. *Journal für die reine und angewandte Mathematik*, 357:138–160.

[Paulin-Mohring, 1989] Paulin-Mohring, C. (1989). *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7.

[Penrose, 1989] Penrose, R. (1989). *The Emperor's New Mind: Concerning Computers, Minds, and The Laws of Physics*. Oxford, NY.

[Pitts, 2006] Pitts, A. M. (2006). Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506.

[Richman, 1983] Richman, F. (1983). Church's thesis without tears. *Journal of Symbolic Logic*, 48(3):797–803.

[Richman, 2008] Richman, F. (2008). Real numbers and other completions. *Mathematical Logic Quarterly*, 54(1):98–108.

[Rosser, 1936] Rosser, J. B. (1936). Extensions of some theorems of Gödel and Church. *Journal of Symbolic Logic*, 1:87–91.

[Schröder, 2005] Schröder, L. (2005). Expressivity of coalgebraic modal logic: The limits and beyond. In Sassone, V., editor, *Foundations of Software Science and Computational Structures*, number 3441 in Lecture Notes in Mathematics, pages 440–454. Springer, Berlin.

[Shankar, 1994] Shankar, N. (1994). *Metamathematics, Machines, and Gödel's Proof*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK.

[Shoenfield, 1967] Shoenfield, J. R. (1967). *Mathematical Logic*. Addison-Wesley.

[Simpson, 1998] Simpson, A. K. (1998). Lazy functional algorithms for exact real functionals. In *MFCS '98: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, pages 456–464, London, UK. Springer-Verlag.

[Stein, 2003] Stein, J. (2003). Coqoban. `http://coq.inria.fr/contribs/Coqoban.html`.

[Stoughton, 1988] Stoughton, A. (1988). Substitution revisited. *Theoretical Computer Science*, 59(3):317–325.

[The Coq Development Team, 2004] The Coq Development Team (2004). *The Coq Proof Assistant Reference Manual – Version V8.0*. `http://coq.inria.fr`.

[Thompson, 1991] Thompson, S. (1991). *Type Theory and Functional Programming*. Addison Wesley.

[Troelstra and Schwichtenberg, 1996] Troelstra, A. S. and Schwichtenberg, H. (1996). *Basic Proof Theory*, volume 43 of *Cambridge tracts in theoretical computer science*. Cambridge University Press.

[Wadler, 1992] Wadler, P. (1992). Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493.

[Wadler, 1995] Wadler, P. (1995). Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK. Springer-Verlag.

[Whitehead and Russell, 1925] Whitehead, A. N. and Russell, B. (1925). *Principia Mathematica*. Cambridge University Press, London, second edition.

[Wiedijk, 2006] Wiedijk, F. (2006). *The Seventeen Provers of the World*. Number 3600 in Lecture Notes in Artificial Intelligence. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Williams, 2002] Williams, R. (2002). Arctangent formulas for PI. `http://www.cacr.caltech.edu/~roy/upi/pi.formulas.html`. Available from `http://web.archive.org/web/20021208234336/http://www.cacr.caltech.edu/~roy/upi/pi.formulas.html`.

[Zumkeller, 2008] Zumkeller, R. (2008). *Global Optimization in Type Theory*. PhD thesis, École Polytechnique Paris.

Glossary of Symbols

$\forall x.\varphi(x)$	3
$\varphi \Rightarrow \psi$	3
$\varphi \wedge \psi$	3
$\top$	3
$\bot$	3
$\neg\varphi$	3
$\varphi \tilde{\vee} \psi$	3
$\tilde{\exists} x.\varphi(x)$	3
$\varphi \vee \psi$	4
$\exists x.\varphi(x)$	4
$\Sigma x.\varphi(x)$	6
$\Pi x.\varphi(x)$	6
$n =_{\mathbb{N}} m$	6
$a\colon A$	8
$a_{\mathbb{N}}$	8
$a_{\mathbb{Q}}$	8
$A + B$	8
$A \times B$	8
$A \Rightarrow B$	8
π_1	8
π_2	8
$\star$	8
$x = y$	10
$x \in P$	10
$\forall x \in P.\varphi$	10
$\exists x \in P.\varphi$	10
$\forall i < n.\varphi$	10
$\exists i \geq n.\varphi$	10
$x \asymp y$	11
$x \neq y$	11
$x \# y$	11
`P -> Q`	14
`P /\ Q`	14
`P \/ Q`	14
`~P`	14
`A + B`	14
`A * B`	14
`A -> B`	14
`n * m`	14
`n + m`	14
`S n`	14
`a :: as`	14
`as ++ bs`	14
`_`	14
$\varphi[\boldsymbol{x}_i/t]$	23
$\Gamma \vdash \varphi$	24
$\Gamma \supseteq F \vdash \varphi$	27
$\varphi[\boldsymbol{x}_0/t_0, \boldsymbol{x}_1/t_1]$	30
$\ulcorner n \urcorner$	32
$\mathbb{A}^{+}$	54
$\mathbb{A}^{0+}$	54
$\mathbb{A}^{+}_{\infty}$	54
$f^{(n)}$	54
f^{n}	54
f^{-1}	54
$\overset{\perp b}{a}$	54
$\underset{\top b}{a}$	54
$\overset{\perp c}{\underset{\top b}{a}}$	54
$\lceil a \rceil$	54
$\lfloor a \rfloor$	54
$\lfloor a \rceil$	54
$\boldsymbol{B}^{X}$	59
$\breve{\boldsymbol{B}}^{X}$	60
$X \to Y$	60
$\breve{\mu}$	61
$\mathfrak{C}$	63
$\hat{a}$	69
$\bar{f}$	70
$\check{f}$	72
$\mathbb{R}$	81
$x +_{\mathbb{R}} y$	82
$x \cdot_{\mathbb{R}} y$	82
$\mathbb{R}^{0+}$	83
$x \leq_{\mathbb{R}} y$	83
$\mathbb{R}^{+}$	83
$x <_{\mathbb{R}} y$	83
x^{-1}	83
$\sum_{i=0}^{\infty} a_i$	84
$\sum_{i=0}^{\infty} a_i$	85
π	85
$\sqrt{x}$	89
$[\![p]\!]_n$	91
χ_n	92
$\sum_{i=0}^{n} x_i$	94
$\mathfrak{S}$	102
$\hat{x}$	102
$f \rhd o \lhd g$	102
$f \blacktriangleright a$	103
$a \blacktriangleleft f$	103
$f @ x$	104
$f ♂ x$	105
$f \langle \circledast \rangle g$	105
$f \{\propto\} g$	106
$f \asymp_{\mathfrak{S}(X)} g$	106
$f \leq_{\mathfrak{S}(\mathbb{Q})} g$	106
$\lVert f \rVert_{\infty}$	107
$\lVert f \rVert_{1}$	107
$\mathfrak{S}^{\infty}$	107
$\mathfrak{S}^{1}$	107
ι	107
$\mathfrak{B}$	108
$\mathfrak{I}$	108
$\int$	108
$\mathbf{I}_{[0,1]}$	109
$\int_{[0,1]}$	110
$\int_{[0,1]}$	110
$\int f \,\mathrm{d}g^{-1}$	110
$\mathbf{B}$	113
$\mathbf{C}$	113
$\mathbf{I}$	113
$\mathbf{K}$	113
$\mathbf{W}$	113
$\langle x, y \rangle$	120
$\boldsymbol{H}^{X \Rightarrow \star}$	121
$x \in l$	122
$\mathfrak{F}$	122
$\mathfrak{K}$	124
$x \in \mathcal{S}$	124
$f \restriction \mathcal{S}$	126
$\boldsymbol{G}(\mathcal{D}, f)$	127
$\boldsymbol{g}(f)$	127
$\boldsymbol{H'}^{X \Rightarrow \star}_{\varepsilon}$	128
$\mathcal{K}_c$	130

Index

abs 107
addition
  real 82
alternating decreasing series 84
andH 22
anonymous function 10
ap
  applicative functor 104
  completion monad 76
  step function 105
ap′ 77
apartness 11
applicative functor 104
apply 20
approx$_{\mathbb{Q}}$ 89
ArcTan_multiple 88
arctangent sum law 88
arity 19
atomic 22
autorewrite 11
bar induction 117
bind 72
bind′ 72
Bishop-compact 124
bounded function 108
bounded search 33
calculus of (co)inductive constructions 5
Cantor distribution 110
Cantor pairing function 31
Cartesian product type 8
Cauchy sequence 56
ceiling 54
characteristic function 32
charFunction 32
checkPrf 36
checkSubFormulaTrace 34
Chinese remainder theorem 37
CiC 5
clamp 54
classical computation theory 129
classical logic 3
codeFormula 31
codeSubFormula 33
codeSysPf 41
codeSysPrf 41
coercion 10
coinductive type 9
combinator 113
compact image 126
compact set 123
  Bishop and Bridges 124
complete 124
completion 63
compose 78
compress 89
computable set 130
computation
  trace 34
computational interpretation 4
Con$_T$ 47
conjunction 3
consistency
  Peano arithmetic 43
ω-consistent 39
const 102
constructive logic 3
constStepF 111
continuity
  uniform 60
coprime 55
cos 85
couple 120
course-of-values recursion 33, 34
CP 24
cPair 31
cPairPi1 33
cPairPi2 33
c-setoid 11
cumulative distribution function 110
curried function 9
curry 78
Curry-Howard isomorphism 5
de Bruijn indices 23, 34
decidable metric 62
DecidableSet 40
Dedekind reals 66
deduction theorem 26
dependent function type 9
dependent type theory 6
Dirac distribution 110

- disjoint union type 8
- disjunction
  - classical 3
  - constructive 4
- dist
  - finite enumeration 126
  - monad 111
  - step function 111
- distribution
  - Cantor 110
  - Dirac 110
- dyadic rational numbers 56
- `Ensemble` 26
- ε-net 124
- `equal` 22
- equality
  - extensional 10
  - intensional 12
  - setoid 11
- α-equivalence 27
- eval 78
- `eval hnf` 88
- `evalPrimRec` 32
- `evalStrongRec` 33
- excluded middle 3, 27
- existential quantifier
  - classical 3
  - constructive 4
- `existH` 22
- $\mathsf{exp}_{\mathbb{Q}}$ 85, 93
- exp 86
- expresses 40
- expressible 40
- extensional
  - `Ensembles` 45
- extensional equality 10, 32
- `extEqual` 32
- false 3
- Few Digits 97
- finite enumeration 122
- fix 6
- fixed point theorem 41
- `Fixpoint` 21
- floor 54
- Flyspeck 53
- fold 103
- $\mathsf{fold}_{\star}$ 106
- $\mathsf{fold}_{\mathrm{sup}}$ 107
- $\mathsf{fold}_{\mathrm{affine}}$ 107
- `fold` 111
- `forallH` 22
- `Formula` 22
- function
  - anonymous 10
  - bounded 108
  - Cantor pairing 31
  - characteristic 32
  - classical 9
  - constructive 9
  - cumulative distribution 110
  - curried 9
  - dependent 9
  - integrable 108
    - Riemann 108
  - primitive recursive 32
  - regular 57, 63
  - respectful 11, 11
  - step 102
- β-function 37
- function type 8
- functional interpretation 4
- `Functions` 19
- functorial strength 75
- glue 102
- `glue` 111
- graph 127
- hemimetric space 121
  - Hausdorff 121, 128
- hereditarily finite set theory 47
- higher order abstract syntax 23
- Hilbert-Bernays-Löb derivability conditions 48
- idempotent monad 74
- identity of indiscernibles 59, 67
- `iffH` 22
- `impH` 22
- implication 3
- `Incompleteness` 39
  - essential 39
    - NN 42
    - Peano arithmetic 42
  - theorem 39
    - second 47
- `Inconsistent` 39
- inductive predicate 9
- inductive type 9
- inductive type families 9
- inequality
  - real 83
  - strict
    - real 83
- `inl` 14
- `inr` 14
- integrable function 108
  - Riemann 108
- integral
  - Riemann 109
  - Stieltjes 110
- intensional equality 12
- `isPR` 32
- `isPRrel` 32
- `iterate_pos` 87

- join
  - completion monad 69
  - finite enumeration 123
  - step function 104
- Julia set
  - filled 130
- Kepler conjecture 53
- Kuroda's negative translation 43
- lambdabot 115
- `Language` 19
- `LazyNat` 86
- left-split 103
- length space 61
- limit
  - sequence 84
- ln$_{\mathbb{Q}}$ 85
- ln 86
- `LNN` 29
- `LNT` 29
- located metric 62
- logic
  - classical 3
  - constructive 3
- `LT` 29
- map
  - applicative functor 105
  - completion monad 70
  - finite enumeration 123
  - step function 104
- map′ 71
- map2
  - applicative functor 105
  - completion monad 77
- map2′ 77
- max 54
- `mem` 25
- Mertens conjecture 53
- metric space 59
  - $\mathbb{Q}$ 62
  - compact sets 123
  - completion 63
  - decidable 62
  - L^∞ 106
  - L^1 106
  - located 62
  - prelength 61
  - product 120
  - stable 62
  - with approximate midpoints 62
- min 54
- `Model` 43
- `ModelConsistent` 43
- modulus
  - continuity 60
- monad 69
  - commutative strong 120
  - finite enumeration 123
  - idempotent 74
  - strength
    - functorial 75
    - tensorial 121
  - symmetric monoidal 120
- multiplication
  - real 82
- multivariable polynomial 91
- `naryFunc` 32
- `naryRel` 32
- `nattoTerm` 32
- negation 3
- negative translation 43
  - Kuroda's 43
- `NN` 29
- `notH` 22
- opaque 13
- `orH` 22
- `PA` 29
- `PAIncomplete` 43
- Peano arithmetic 29
  - consistent 43
- π 85
- pigeon hole principle 123
- plot 127
- `Plus` 29
- polynomial
  - multivariable 91
- power series 84
- predicate 8
- prelength space 61, 67
- `Prf` 24
- primitive recursive expression 32
- primitive recursive function 32
- `PrimRec` 32
- program extraction 13
- projection
  - first and second 8, 33
- `Prop` 12
- proposition 8
- propositions as types 8
- Prov$_T$ 47
- `PRsolveFV` 38
- pure 104
- Q
  - Robinson's system 46
- `Qred` 55
- quantifier
  - existential
    - classical 3
    - constructive 4
  - universal 3
- quotient type 11
- raster 127
- rational numbers 55

  - dyadic 56
- real number structure 94
- real numbers 81
- reciprocal
  - real 83
- record type 9
- recursion
  - course-of-values 33, 34
  - general 6
  - structural 9
  - well-founded 6
- `reflect_right` 88
- reflection 7
- regular
  - function 57, 63
  - sequence 56
- `Relations` 19
- representable 36
- `RepresentsInSelf` 40
- respectful function 11, 11
- rewrite database 11
- Riemann integrable function 108
- Riemann integral 109
- right-split 103
- Robinson's system 46
- `round` 54
- search
  - bounded 33
- `searchXY` 33
- second incompleteness theorem 47
- `Sentence` 40
- sequence
  - Cauchy 56
  - regular 56
- series
  - alternating decreasing 84
  - sub-geometric 84
- `Set` 12
- set theory
  - hereditarily finite 47
- setoid 11
  - constructive 11
- `setoid_replace` 11
- `setoid_rewrite` 11
- $\mathsf{sin}_{\mathbb{Q}}$ 85, 93
- split 103
- `Split` 111
- `SplitL` 111
- `SplitR` 111
- stable
  - formula 3
  - metric 62
  - relation 44
- step function 102
- `stepSample` 115
- Stieltjes integral 110
- `strength` 120
- sub-geometric series 84
- `substituteFormula` 23
- `Succ` 29
- `sup` 108
- `swap` 121
- `switchPR` 33
- `SysPrf` 25
- `System` 25
- $\mathsf{tan}_{\mathbb{Q}}^{-1}$ 85
- $\mathsf{tanh}_{\mathbb{Q}}^{-1}$ 85
- Tarski's theorem 41
- `Tcons` 20
- `Term` 20
- `Terms` 20
- `Times` 29
- `Tnil` 20
- totally bounded 124
- trace of computation 34
- transparent 13
- triangle law 59
- true 3
- type
  - Cartesian product 8
  - coinductive 9
  - dependent function 9
  - disjoint union 8
  - function 8
  - inductive 9
  - inductive predicate 9
  - quotient 11
  - record 9
- `uncurry` 78
- uniform
  - continuity 60
- `unit`
  - completion monad 69
  - finite enumeration 123
  - step function 104
- universal quantifier 3
- `var` 20
- `Vcons` 20
- `Vector` 20
- `Vnil` 20
- Z2 47
- `Zero` 29

Samenvatting

Onvolledigheid & Volledigheid: Formalisatie van Logica en Analyse in Typetheorie

Dit proefschrift omvat twee bewijsontwikkelingen in de constructieve typetheorie: een bewijs van de eerste onvolledigheidsstelling van Gödel en een ontwikkeling van exacte reële rekenkunde. Ik werk in een constructieve typetheorie, omdat dit systeem de unieke mogelijkheid heeft om afhankelijk getypeerd programmeren vrij te combineren met bewijzen in natuurlijke deductie. Door het Curry-Howard isomorfisme gebruiken deze twee taken precies dezelfde taal. Het programmeeraspect en het bewijsaspect van deze taal ondersteunen elkaar. Bewijzen worden gebruikt voor het certificeren van de correctheid van functies, en correcte functies kunnen worden geëvalueerd in bewijzen om problemen op te lossen.

Hoofdstuk 1 en Hoofdstuk 2 introduceren het onderwerp van mijn proefschrift en geven een korte introductie tot de soort typetheorie die ik zal gaan gebruiken. Beide ontwikkelingen zijn geformaliseerd met de Coq bewijs-assistent, een implementatie van constructieve typetheorie. Ik toon hoe klassieke logica kan worden gezien als een fragment van constructieve logica, zodat klassiek redeneren ook wordt ondersteund in constructieve typetheorie.

In deel I beschrijf ik mijn ontwikkeling van de eerste onvolledigheidsstelling van Gödel. Het begint met Hoofdstuk 3, dat een korte inleiding en een beschrijving van de onvolledigheidstelling geeft.

De beschrijving van mijn formele bewijzen begint in Hoofdstuk 4, waarin ik mijn datastructuur voor eerste orde formules en mijn datastructuur voor bewijzen in de eerste-orde logica beschrijf. Ik definiëer twee axiomasystemen: Peano's rekenkunde en een zeer zwak rekenkundig systeem, genaamd NN. Deze interne logica is klassiek, terwijl de logica van Coq constructief is. Ik heb gekozen voor een traditionele definitie van substitutie in formules waarin gebonden variabelen hernoemd worden om te voorkomen dat vrije variabelen worden gebonden. Deze keuze veroorzaakt veel problemen door het hele bewijs.

Hoofdstuk 5 bespreekt hoe ik verschillende structuren als natuurlijke getallen codeer en hoe ik aantoon dat de functies die op deze structuren werken voorgesteld kunnen worden door rekenkundige formules. In plaats van de hoofdstelling van de rekenkunde, zoals Gödel oorspronkelijk gebruikte, gebruik ik recursief Cantor's paringsfunctie om de abstracte syntaxbomen van formules en bewijzen te coderen. In Sectie 5.2, definiëer ik een taal voor primitief recursieve functies. Ik heb primitief recursieve programma's geschreven voor alle benodigde functies op formules en bewijzen. Ik bewijs dat deze primitief recursieve programma's correcte representaties van de oorspronkelijke Coq functies zijn. In Sectie 5.4, beschrijf ik mijn bewijs dat alle primitief recursieve functies voorgesteld kunnen worden door rekenkundige formules.

Hoofdstuk 6 geeft mijn formele uitspraak van de onvolledigheidsstelling. Ik bewijs dat het systeem NN wezenlijk onvolledig is. In het bijzonder is Peano's rekenkunde onvolledig, omdat ik ook bewijs dat Peano's rekenkunde consistent is door aan te tonen dat de natuurlijke getallen van Coq een model van Peano's rekenkunde vormen. Tenslotte, geeft Hoofdstuk 7 een paar slotopmerkingen over dit bewijs en beschrijft wat nog meer nodig zal zijn om de tweede onvolledigheidsstelling te bewijzen.

In deel II, beschrijf ik mijn implementatie van exacte reële rekenkunde door eerst een algemene implementatie van volledige metrische ruimten maken. Dit deel begint met Hoofdstuk 8, dat een korte inleiding en achtergrondinformatie over Bishop en Bridges's benadering van de reële getallen geeft [Bishop and Bridges, 1985].

Hoofdstuk 9 introduceert mijn formele definitie van een metrische ruimte en verschillende classificaties van metrische ruimten met extra eigenschappen. Uniform continue functies tussen metrische ruimten worden gedefinieerd. Zij spelen een centrale rol in dit werk.

Hoofdstuk 10 definiëert de completeringsoperatie die een volledige metrische ruimte maakt uit een willekeurige metrische ruimte. Ik toon aan dat deze completeringsoperatie de vereiste eigenschappen heeft om er een monad van te maken. Ik definieer operaties die uniform continue functies over metrische ruimten liften tot uniform continue functies over de completering van die metrische ruimten.

Hoofdstuk 11 definiëert de reële getallen als de completering van de rationale getallen. De lichaamsoperaties voor de reële getallen worden gedefinieerd door de operaties van de rationale getallen te liften. Andere elementaire functies definieer ik eerst op de rationale getallen als de limiet van een machtreeks. Daarna worden ook zij gelift zodat ze op alle reële getallen werken. Sectie 11.5 beschrijft manieren om sommige veelvoorkomende operaties efficiënter te definiëren. Al mijn functies kunnen worden uitgevoerd binnen Coq en zijn efficiënt genoeg om benaderingen praktisch uit te rekenen. Ik heb een tactiek geschreven die zulke berekeningen gebruikt om ongelijkheden tussen gesloten uitdrukkingen van reële getallen op te lossen.

Het doel van mijn werk aan volledige metrische ruimten is de reële getallen te definiëren. Echter, mijn implementatie van volledige metrische ruimten is algemeen. Als illustratie hiervan beschrijft Hoofdstuk 12 de metrische ruimte van integreerbare functies als de completering van de formele stapfuncties met de L^1-metriek. Uniform continue functies kunnen worden afgebeeld op integreerbare functies op [0, 1] en geïntegreerd worden. Sectie 12.1.8 toont hoe de Stietljes integraal gratis uit dit werk volgt.

Hoofdstuk 13 geeft een andere toepassing van volledige metrische ruimten, dit keer op compacte verzamelingen. De metrische ruimte van compacte verzamelingen wordt als de completering van de ruimte van eindige verzamelingen met de Hausdorffmetriek gedefiniëerd. Dit betekent dat elke compacte verzameling benaderd kan worden door een eindige verzameling. Bovendien kan zo'n eindige verzameling worden gerasterd en *weergegeven* in het Coq systeem. Ik toon bijvoorbeeld aan dat de grafiek van een uniform continue functie op een compact interval compact is. Figuur 13.1 toont een benadering van de exponentiële functie getekend met Coq. Sectie 13.2.2 geeft een mooi voorbeeld hoe men klassieke en constructieve redeneringen kan mixen. Sectie 13.6 bespreekt waarom ik denk dat constructieve logica een betere taal is om te praten over berekenbaarheid dan de klassieke berekenbaarheidstheorie.

Hoofdstuk 14 eindigt met conclusies uit dit onderzoek.

Curriculum Vitae

Born in Winnipeg, Manitoba, Canada on December 19th, 1977.

Sep. 1989 – Jun. 1995. Pinawa Secondary School (Pinawa, Manitoba, Canada). Graduated high school with honours.

Sep. 1995 – Apr. 2000. University of Waterloo (Waterloo, Ontario, Canada). Participated in co-op program. Received Double Honours Bachelor of Mathematics, Computer Science and Pure Mathematics with Distinction—Dean's Honours list.

Sep. 2000 – Dec. 2003. University of California at Berkeley (Berkeley, California, USA). PhD candidate in the Logic and Methodology of Science group.

Jan. 2004 – Dec. 2004. Inscriber Technology Corporation (Waterloo, Ontario, Canada). Software Engineer.

Jan. 2005 – Dec. 2008. Radboud University Nijmegen (Nijmegen, the Netherlands). PhD student in the Foundations group under the supervision of Dr. Bas Spitters and Prof. Dr. Herman Geuvers.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support- .* Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics.* Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems.* Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication.* Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data.*

Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell.* Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for*

Prime Time. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19