Veli Peltola

Interactively explorable formal proofs for textbooks of mathematics

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 15.9.2015

Thesis supervisor:

Prof. Camilla Hollanti

Thesis instructor:

D.Sc. (Tech.) Tomi Janhunen

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Veli Peltola

Title: Interactively explorable formal proofs for textbooks of mathematics

Date: 15.9.2015 Language: English Number of pages: 6+73

Department of Mathematics and Systems Analysis

Professorship: Mathematics Code: Mat-1

Supervisor: Prof. Camilla Hollanti

Instructor: D.Sc. (Tech.) Tomi Janhunen

Electronic textbooks of mathematics would benefit from interactively explorable proofs, where the reader can request a more detailed explanation for any part of the proof he or she does not understand. This requires that definitions, theorem statements, and proofs are written in some formal language. In this thesis we investigate the theoretical and practical challenges of producing such textbooks.

Any particular choice of language cannot be adequate for all textbooks, so we construct a theoretical framework for discussing extensible languages. Under this framework we define three languages for expressing definitions and theorem statements. The first two correspond to the standard languages of propositional and first-order logics. The third language is expressive enough for most definitions and theorem statements in discrete mathematics, but conceptually less problematic than the languages of set theory and higher-order logic, because it cannot express unrestricted quantification over sets that are larger than the set of real numbers.

In addition, an implementation of an interactive proof explorer is presented. Its capabilities are demonstrated with an explorable proof of Bertrand's postulate. The focus of this thesis is on the experience of the reader, and the results seem promising in that regard. With further work on making the authoring tools more efficient to use, it should be feasible to formalize an entire textbook.

Keywords: formal languages, formal proofs, mathematics education, first-order logic, second-order arithmetic, Bertrand's postulate, continuum hypothesis

AALTO-YLIOPISTO
PERUSTIETEIDEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Veli Peltola

Työn nimi: Formaalien todistusten interaktiivinen tarkastelu matematiikan oppikirjoissa

Päivämäärä: 15.9.2015 Kieli: Englanti Sivumäärä: 6+73

Matematiikan ja systeemianalyysin laitos

Professuuri: Matematiikka Koodi: Mat-1

Valvoja: Prof. Camilla Hollanti

Ohjaaja: TkT Tomi Janhunen

Sähköisissä matematiikan oppikirjoissa olisi hyötyä interaktiivisista todistuksista, joiden avulla lukija voisi pyytää tarkentavaa selitystä mihin tahansa kohtaan, jota hän ei ymmärrä. Tällaisen kirjan kirjoittaminen vaatii, että määritelmät, lauseet ja todistukset on kirjoitettu jollain formaalilla kielellä. Tässä diplomityössä tutkitaan sekä käytännön että teorian tasolla interaktiivisia todistuksia sisältävien oppikirjojen kirjoittamiseen liittyviä haasteita.

Koska mikään yksittäinen kieli ei voi olla riittävä kaikkien oppikirjojen tarpeisiin, työssä määritellään kielen laajentamisen käsite. Tämän viitekehyksen sisällä kehitetään määritelmille ja lauseille kolme kieltä, joista kaksi ensimmäistä vastaavat lauselogiikkaa ja ensimmäisen kertaluvun predikaattilogiikkaa. Kolmas kieli, joka on näiden laajennus, on ilmaisuvoimaltaan riittävä monien diskreettiin matematiikkaan keskittyvien oppikirjojen formalisoimiseen. Tämä kieli on käsitteellisesti yksinkertaisempi kuin joukko-opin tai korkeamman kertaluvun predikaattilogiikan kielet, koska siinä ei pysty rajoittamattomasti kvantifioimaan sellaisten joukkojen yli, jotka ovat suurempia kuin reaalilukujen joukko.

Lisäksi työssä esitellään tietokoneohjelma, jonka avulla interaktiivisesti tarkasteltavia formaaleja todistuksia voi lukea, sekä tämän ohjelman avulla kirjoitettu Bertrandin postulaatin todistus. Tässä työssä keskitytään todistuksiin lukijan näkökulmasta, ja tulokset ovat siltä osin lupaavia. Kokonaisen oppikirjan formalisoimiseksi vaaditaan vielä lisätyötä oppikirjan kirjoittajan työn helpottamiseksi.

Avainsanat: formaalit kielet, formaalit todistukset, matematiikan opetus, ensimmäisen kertaluvun logiikka, toisen kertaluvun aritmetiikka, Bertrandin postulaatti, kontinuumihypoteesi

Preface

In one way or another, I have been thinking about formal proofs and foundations of mathematics for more than ten years. This thesis has been my first real opportunity to put some of those thoughts on paper. I would like to thank my supervisor, Professor Camilla Hollanti, for valuable comments, and for providing me the opportunity to work on this subject. I am also grateful to my instructor, Dr. Tomi Janhunen, who has given me detailed and insightful feedback throughout the process of writing this thesis. I feel it has sharpened both my thinking and my writing on this matter.

Several of my friends have been interested enough—or perhaps, patient enough—to talk with me about these things. I owe many ideas, and even more importantly, interesting questions, to the conversations I have had with Kristian Nybo, Markus Ojala, Ville Pettersson, and Koos Zevenhoven—to name a few. Finally, I would like to thank my family and my dearest Maija for their continuing support.

Espoo, 15.9.2015

Veli Peltola

Contents

1 Introduction

Education is becoming increasingly electronic. Traditional textbooks are printed on paper, but newer, electronic textbooks are only meant to be read on screen. This makes it possible for the textbooks to contain interactive material, which can be useful for many different purposes. Textbooks of mathematics, in particular, might benefit from *interactively explorable proofs*. When reading a proof, a student may encounter a step that he or she does not fully understand. An interactive textbook could, upon request, provide additional explanation of that particular step, making it easier for the student to fully understand the proof.

Elementary mathematics education is not so much focused on proofs as it is focused on teaching various skills and techniques: how to perform mental calculation, how to solve an equation, or how to simplify an expression. Therefore, in this thesis we focus on textbooks of pure mathematics at the university level. No particular subject is assumed, but possible examples include number theory, abstract algebra, combinatorics, and graph theory. Such books are often, at their core, about definitions, theorem statements, and their proofs. In order for the textbook software to produce detailed explanations of proof steps, it has to understand the proofs at some level. Definitions, theorem statements, and proofs are usually written in some natural language, such as English, but this makes it extremely difficult to develop such software. The problem becomes more approachable if they are written in some *formal language* instead.

Let us look at definitions first. A mathematical definition is a description that uniquely specifies some mathematical object and gives it a name. The object can be of several different types. It can be a constant (define e as $\lim_{n\to\infty}(1+1/n)^n$) or a function (define the factorial $! : \mathbb{N} \to \mathbb{N}$ so that $0! = 1$ and $n! = n(n-1)!$ for $n > 0$). It can even be a new class of mathematical objects, such as when we define what is meant by a *graph*, a *metric space*, or a *complex number*. In this work we will mostly focus on function definitions. Today, programming languages are the most widely used formal languages that can be used to unambiguously define functions. The benefit of defining functions in programming languages is that these definitions are executable: a computer can be used to calculate the value of the function for a given input. However, as will be shown later in this thesis, there are mathematical functions that cannot be defined in a programming language, which means that sometimes we need to use a more expressive definition language instead.

Next, we turn to theorem statements, which often take the form of *conditional statements*. They introduce some variables, and then claim that whenever certain hypotheses are satisfied, some conclusion will follow. Let us take *Fermat's Last Theorem* as an example. Suppose that $a, b, c, n \in \mathbb{Z}^+$. If $n > 2$, then $a^n + b^n \neq c^n$, as proven by Andrew Wiles [53]. In this theorem, we have four variables (a, b, c, and n) whose range is the set of positive integers, a single hypothesis ($n > 2$), and a conclusion ($a^n + b^n \neq c^n$). To formalize conditional statements like this, we need to define precisely what kind of variables can be introduced, what kind of statements are allowed as hypotheses and the conclusion, and what exactly does it mean for a conclusion to follow from the hypotheses. Conditional statements such as this are implicitly *universally quantified*, meaning that they must hold for all values in the ranges of their variables.

Finally, when we have chosen a language for representing theorem statements, we can choose a language for representing their proofs. Specifically, we want a language for which there exists an algorithm that can check if a potential proof is correct or not. In conventional mathematics, a proof that is written in English may be called *formal* if it is rigorous and does not appeal to intuition, but in this work we only use the term *formal proof* for proofs that are written in some special proof language. These formal proofs do not need to be displayed to the reader as static text. It is possible to create *proof explorers*: programs that can show the structure of the proof in various ways, and provide clarification for any part of the proof that the reader finds unclear. This is the main reason why formal proofs, in conjunction with suitable software, have the potential to be more understandable than conventional proofs.

1.1 Formal proofs and computers

The idea of formal proofs originated before computers, which meant that early formal proofs had to be checked by humans. Checking formal proofs is straightforward in principle, but it is not something humans are particularly good at. Replacing a conventional proof with a formal proof was likely to make the proof harder to verify, not easier. Formal proofs were mainly seen as a theoretical idea to be studied, rather than a tool to be actually used [24].

This changed with the advent of computers: it became possible to write programs that check formal proofs. Of course, the proof checking program might still contain programming errors, or the computer hardware might be faulty. For this reason, computer-verified proofs cannot give us any kind of absolute philosophical certainty, but they do seem to work well in practice. Thomas Hales argues [21, p. 1376]:

> We can assert with utmost confidence that the error rates of top-tier theorem proving systems are orders of magnitude lower than error rates in the most prestigious mathematical journals.

The value of computers becomes most apparent when we look at proofs that are long and complicated. If we have read and understood the proof of the fact that $\sqrt{2}$ is irrational, a computer-verified proof will add nothing to our feeling of certainty. There are, however, mathematical theorems whose currently known proofs are unusually difficult for humans to verify. Two examples, the *four color theorem* and the *Kepler conjecture*, are presented in more detail in Sections 3.1 and 3.2, respectively. In such cases having a formal proof instead of a conventional one can greatly increase the confidence we can have in the result.

Formal proofs can also be used for formal verification of software, which is discussed in Section 3.3. This is particularly useful for safety-critical systems, such as medical devices or aircraft flight control. Proof checkers used for this type of work should be exceptionally reliable. One of the best ways to achieve this is to use a minimalistic system, where the number of things to be trusted is as small as possible. For example, it takes fewer than 500 lines of computer code to implement the kernel of the *HOL Light* proof assistant [21, p. 1376].

The attitude taken in this work is slightly more relaxed. A textbook is not a safety-critical system, so we are satisfied with an ordinary level of reliability. This does not mean that we will knowingly allow false statements to be proven; it means that it is more important to have readable proofs than to have a system that is as minimalistic as possible. Our main goal is not to reduce the number of errors in textbooks, but to ensure that the proofs are documented so carefully that the proof explorer is able to explain every step.

1.2 Human-readable formal proofs

Despite the success in other areas, formal proofs are rarely used when communicating mathematical ideas between humans. The overwhelming majority of textbooks and research articles use conventional proofs instead of formal ones. This is despite the fact that several potential benefits of adopting formal proofs have been listed [5]. For example, if definitions, theorem statements, and proofs in research articles were written completely formally, then peer review could be partly automated. Referees could spend their time assessing the originality and importance of a submission rather than its correctness [25].

One possible explanation why formal proofs are not widely used is the fact that writing formal proofs is harder than writing conventional proofs. As a rule of thumb, it currently takes about one week to formalize a page of textbook-level mathematics [26]. For research articles, the situation may be much worse. Some of this hardness is unavoidable, since in conventional proofs one can freely cite any previously published theorems, but in strictly formal proofs one can only use the tiny percentage of theorems that have already been formalized.

We will focus on textbooks, because their situation looks more promising in the short term. Textbooks typically rely on a much smaller base of existing results than research articles, but they can still contain nontrivial and interesting proofs. Moreover, they are written for a larger audience. Extra effort required for writing formal proofs might be justifiable, if only the experience of reading formal proofs can be made better than the experience of reading conventional proofs.

One of the simplest ways to make formal proofs readable is to display expressions using standard mathematical notation instead of plain text. The person reading the proof is likely to find $\sqrt{2}$ easier to read than `sqrt(2)`, and several of the current proof assistants provide a way to produce a typeset version of the proof with the help of LaTeX or by other means [52]. Ideally, a statically typeset proof produced by such a program might be as readable as a conventional proof, but it could hardly be *more* readable. Because of this, we will focus on interactively explorable proofs.

1.3 Automated theorem proving

In addition to proof checking, there exists a closely related field of *automated theorem proving.* A proof checker is a program that automatically checks a formal proof, but an automated theorem prover tries to also find the proof. We will briefly look at some successes of automated theorem proving, such as the *Erdős discrepancy problem* and the *Robbins conjecture* in Sections 5.6 and 6.2, respectively. Some of the most difficult formally proven theorems, such as the four color theorem and the Kepler conjecture, have used a mixed approach, where a human expert provides an outline of the proof, and an automated theorem prover fills in the low level details.

The system presented in this thesis contains a proof checker and a proof explorer, but no algorithms that try to search for proofs. Automated theorem provers can make authoring of formal proofs easier, but they do not change the experience of the reader. Therefore, they are not needed to answer the first question we should be asking ourselves: is it possible to write formal proofs that readers prefer to conventional proofs? If the answer turns out to be yes, then as the next step it would certainly be important to try to help textbook authors write formalized textbooks efficiently. Technically, it would be possible to use existing theorem provers in conjunction with the system presented in this thesis, but this is not necessarily desirable. Automatic theorem provers do not always prefer the same building blocks as human readers would. For example, there is an influential proof system introduced by John Alan Robinson, which is based on a single rule of inference called *resolution* [46]. This approach has been extremely successful for automated theorem proving, but the resulting proofs are in a form that is not particularly illuminating for human readers [12, p. 4].

Since we are not using formal proofs just to have a guarantee of correctness, we would not necessarily be satisfied with the first proof that an automated theorem prover happens to find. Instead, we are looking for a proof that a human reader can easily follow. One possibility would be to create a theorem prover that tries to find several proofs for a given statement, and automatically chooses the best one. It would be easy to choose the shortest proof, but there are other desirable properties of proofs that are not as easy to measure. As an example of this, let us consider the *Brahmagupta–Fibonacci identity.* We state it for $a, b, c, d \in \mathbb{Z}$, but it also holds in a more general setting:

$$(a^2 + b^2)(c^2 + d^2) = (ac - bd)^2 + (ad + bc)^2. \tag{1}$$

This result is used when determining what natural numbers can be represented as a sum of two squares [23]. It is a polynomial identity, and it would not be hard for an automated theorem prover to find a proof along these lines:

$$\begin{aligned}(a^2 + b^2)(c^2 + d^2) &= a^2c^2 + a^2d^2 + b^2c^2 + b^2d^2 \\ &= (a^2c^2 - 2abcd + b^2d^2) + (a^2d^2 + 2abcd + b^2c^2) \\ &= (ac - bd)^2 + (ad + bc)^2.\end{aligned}$$

Besides this, there is another proof that uses complex numbers. It is conceivable that an automated theorem prover could find this proof too:

$$\begin{aligned}(a^2+b^2)(c^2+d^2) &= |a+bi|^2 \cdot |c+di|^2 = (|a+bi| \cdot |c+di|])^2 \\ &= |(a+bi)(c+di)|^2 = |(ac-bd)+(ad+bc)i|^2 \\ &= (ac-bd)^2+(ad+bc)^2.\end{aligned}$$

There are several reasons for considering the second proof to be better. It makes the identity more memorable, and gives a plausible explanation of how one could have discovered it. It also suggests a generalization. The identity

$$\begin{aligned}&(a_1^2+a_2^2+a_3^2+a_4^2)(b_1^2+b_2^2+b_3^2+b_4^2) \\ &\qquad = (a_1b_1+a_2b_2+a_3b_3+a_4b_4)^2+(a_1b_2-a_2b_1+a_3b_4-a_4b_3)^2 \\ &\qquad + (a_1b_3-a_2b_4-a_3b_1+a_4b_2)^2+(a_1b_4+a_2b_3-a_3b_2-a_4b_1)^2\end{aligned}$$

is known as *Euler's four-square identity*, and it can be proven (and discovered!) in a similar way using *quaternions*, which are an extension of complex numbers [23].

Memorability, discoverability, and generalizability are all desirable features in a proof, but measuring them algorithmically is challenging. This is one reason why it would be hard to develop a theorem prover that automatically produces the kind of proofs that human readers prefer.

1.4 Goals of this thesis

This thesis investigates the theoretical and practical aspects of producing electronic textbooks of mathematics that contain interactively explorable formal proofs. The outline of this thesis is as follows. In Chapter 2 we cover mathematical preliminaries needed for later chapters. To put the work in a wider context, in Chapter 3 we look at some examples of prior work.

In Chapter 4 we argue that no fixed language can be expressive enough for all definitions, theorem statements, and proofs that can arise in textbooks. For this reason, we define a theoretical framework for discussing extensible languages. In particular, we introduce the concepts of a *function definition language*, a *conditional statement language*, and a *proof language*, which will be used in later chapters.

In Chapter 5 we formalize *propositional logic* as the conditional statement language PL. This language is insufficiently expressive to formalize the contents of any realistic textbook. However, because of its simplicity, it serves as a motivating example for later chapters. In addition, we discuss the computational and theoretical applications of propositional logic. In Chapter 6 we define the conditional statement language FL, which corresponds to *first-order logic*, and is an extension of PL. It is needed to formalize several concepts from abstract algebra. In addition, we look at some of the theoretical properties of first-order logic. This is particularly important to understand its limitations, and to see why it is sometimes necessary to go beyond first-order logic.

One common way to extend first-order logic is to move to *higher-order logic*. However, as we argue in Chapter 7, it is not entirely clear what it means for a conditional statement in higher-order logic to be true. This seems undesirable from a pedagogic point of view, so we extend first-order logic in another direction instead. This results in a language denoted by SD. Of the commonly studied formal languages it has the most in common with *second-order arithmetic*. The conditional statement language SD also comes with a function definition language for defining new computable functions. It is the most expressive conditional statement language in this thesis, but like PL and FL, it should not be seen as an end to itself, but as a starting point for future extensions.

On the more practical side, in Chapter 8 we look at a proof checker and an interactive proof explorer that were implemented as a part of this thesis project. Their capabilities are demonstrated with an explorable proof of Bertrand's postulate. Finally, in Chapter 9 we make some concluding remarks.

2 Preliminaries

In this chapter we cover mathematical preliminaries needed for later chapters.

2.1 Set theory

Familiarity with basic set theory is assumed, so in this section we will mostly cover some notational conventions. The set of natural numbers is taken to contain the number zero: $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$. We write $X \subseteq Y$ when X is a subset of Y, and $X \subset Y$ when X is a *proper subset* of Y, meaning that $X \subseteq Y$ and $X \neq Y$. If X is a set, then the *power set of* X, which is the set of all sets whose members belong to X, is usually denoted by $\mathcal{P}(X)$. We write $\textsc{Set}(X)$ instead, because it is analogous with the notation $\textsc{List}(X)$ to be introduced in Section 2.3. The set of Booleans is denoted by $\mathbb{B} = \{\textsc{False}, \textsc{True}\}$, and a Boolean-valued function is called a *predicate*. Boolean-valued functions are sometimes called *finitary relations*, but we reserve the word *relation* for predicates that take two inputs, such as $=$, $<$, and $\in$. *Universal quantification* over a set is denoted by $\forall$, and *existential quantification* over a set is denoted by $\exists$. In particular, if X is a set and $\phi : X \to \mathbb{B}$ is a predicate, then we write $\forall x \in X : \phi(x)$ to indicate that $\phi(x) = \textsc{True}$ for all x in X, and $\exists x \in X : \phi(x)$ to indicate that there exists at least one x in X for which $\phi(x) = \textsc{True}$.

We will also use several basic facts about the cardinality (size) of sets. Suppose that X and Y are sets. The cardinality of X is denoted by $|X|$. The set of all functions from X to Y is denoted by $X \to Y$, and we have $|X \to Y| = |Y|^{|X|}$. The sets $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ have cardinality $\aleph_0$, and are said to be *countably infinite*. A set that is either finite or countably infinite is called *countable*. For all sets X, we have $|\textsc{Set}(X)| = 2^{|X|} > |X|$. The cardinal number $|\textsc{Set}(\mathbb{N})| = |\mathbb{R}| = 2^{\aleph_0}$ is often denoted by the letter c, which stands for the word *continuum*. In this work we simply refer to it as $2^{\aleph_0}$. If $2 \leq |X| \leq \aleph_0$ and $\aleph_0 \leq |Y| \leq 2^{\aleph_0}$, then $|X \to Y| = |Y|^{|X|} = 2^{\aleph_0}$. In particular, this implies that $|\mathbb{Q} \to \mathbb{R}| = 2^{\aleph_0}$. The cardinality of arbitrary subsets of real numbers or arbitrary real functions is $|\textsc{Set}(\mathbb{R})| = |\mathbb{R} \to \mathbb{R}| = 2^{2^{\aleph_0}}$.

2.2 Strings

A *string* is a finite sequence of characters. We write strings inside quotation marks using a monospaced font. Thus, “`hi there`” is a string consisting of 8 characters, including the space between the words. The exact definition of a string depends on the *character set* in use. For simplicity, we will use a fixed character set throughout this work. The character set is a subset of the ASCII characters.

Definition 1. The countably infinite set $\mathbb{S}$ consists of all strings that contain zero or more of the following characters:

Letters:	`abcdefghijklmnopqrstuvwxyz`
	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
Digits:	`0123456789`
Punctuation:	`()[]{},+-*/=<>^:`
The space character:	

Our main use for strings is the representation of mathematical expressions. For example, the expression $2+3$ can be represented by the string “`2 + 3`”. When doing this for more complicated expressions, we run into two difficulties. First, the character set of $\mathbb{S}$ is quite limited: it is missing the Greek letters and several common mathematical symbols, including $\leq$, $\neq$, ∞, $\mathbb{N}$, and $\in$. Second, traditional mathematical notation includes various “two-dimensional” constructions, such as those found in the nonsensical formula

$$\sum_{k=1}^{\infty} \frac{1}{k^2} > \binom{n}{k} - \lim_{n \to \infty} \sqrt{a_n}.$$

Both of these problems will be handled by introducing an alternative notation for those strings that represent mathematical expressions. To distinguish it from the usual notation, we use single quotes instead of double quotes. For example, we could define ‘$\sqrt{1-\theta^2}$’ as an alternative notation for the string “`sqrt(1 - theta^2)`”. The original notation with double quotes corresponds to how the expressions will be typed into the computer, so we call it the *input notation*. The alternative notation with single quotes corresponds to what will be displayed to the person reading the proof, so we call it the *display notation*.

It is sometimes useful to insert a string inside another. We denote this by drawing a rounded rectangle around the string-valued expression whose value should be inserted. For example, if x = “`def`”, then “`abc`$\boxed{x}$`ghi`” = “`abcdefghi`”. If no ambiguity arises, we will use this notational device in display notation also.

The use of quotation marks for strings is uncommon in mathematics; most writers omit them. This is feasible if the strings studied are sufficiently different from the notation used to study them. In our case, the input notation is designed to be completely indistinguishable from the mathematical notation we use elsewhere, so there would be greater potential for confusion if quotation marks were left out.

2.3 Tuples and lists

We distinguish between two types of finite sequences: *tuples* and *lists*. The elements of a tuple are written inside parentheses, as in $(x_1, \ldots, x_k)$, and the elements of a list are written inside square brackets, as in $[x_1, \ldots, x_k]$. Technically, two separate types are not needed, but the typical use cases are somewhat different. Tuples are used when the number of elements is fixed in advance, and the elements may be of different types. Lists are used when all elements are of the same type, and the number of elements is not fixed in advance.

The Cartesian product of k sets produces a set of k-tuples. If $X_1, \ldots, X_k$ are sets, then $(x_1, \ldots, x_k) \in X_1 \times \ldots \times X_k$ if and only if $x_i \in X_i$ for all $i \in \{1, \ldots, k\}$. Conventionally, 2-tuples and 3-tuples are called *pairs* and *triples*, or if the context requires, *ordered pairs* and *ordered triples*. In this thesis we use tuples for one purpose only: the definition of various mathematical objects. When doing so, the number of elements is fixed in advance, and the elements are informally given names. For example, a *graph* G is defined as a pair (V, E) consisting of a set of *vertices* V and a set of *edges* E. Since this is our only use case for tuples, we do not need k-tuples for $k < 2$. This is convenient, because 1-tuples could be confused with the ordinary use of parentheses for controlling the order of evaluation.

If X is a set, then $\textsc{List}(X)$ denotes the set of all lists whose elements belong to X. For example, $[2, 5, 3, 2] \in \textsc{List}(\mathbb{Z})$. We use the set membership symbol '$\in$' for lists as well. This defined so that $y \in [x_1, \ldots, x_k]$ if $y = x_i$ for some $i \in \{1, \ldots, k\}$. We have, for example, $2 \in [2, 5, 3, 2]$ and $4 \notin [2, 5, 3, 2]$. If X is a nonempty countable set, then $\textsc{List}(X)$ is countably infinite. Unlike tuples, we use lists in circumstances where they may have less than 2 elements.

2.4 Free and bound variables

Suppose that $n \in \mathbb{Z}^+$, and let us look at the expression

$$\sum_{k=1}^{n} k^2. \tag{2}$$

The value of this expression cannot depend on the value of k, since k is a *bound variable*. We can think of k as a placeholder name that is introduced by the special notation $\sum$, and which does not even properly exist outside of (2). In addition to $\sum$, there are several other constructions that can introduce bound variables, such as the quantifiers $\forall$ and $\exists$. In contrast to k, the value of the expression (2) can depend on the value of n, and we say that n in (2) is a *free variable*. In logic, it is common to allow the *capture* of variables that have already been introduced, so that every occurrence of a variable refers to the "innermost" introduction of that variable. For example, suppose that $n \in \mathbb{Z}$. Then the expressions

$$n + \sum_{n=1}^{10} \left(n^2 \cdot \sum_{n=1}^{5} 2n \right) \qquad \text{and} \qquad n + \sum_{i=1}^{10} \left(i^2 \cdot \sum_{j=1}^{5} 2j \right)$$

are equivalent. In the first expression, the first of occurrence of n is free, but the rest are not. Allowing variable capture has some uses in logic, but it is difficult to see a place for it in a proof that attempts to maximize readability. Therefore, at the cost of complicating the definitions somewhat, the formal languages in this thesis forbid variable capture.

2.5 Computability

Functions on natural numbers ($\mathbb{N}^k \to \mathbb{N}$) can divided into two classes: *computable* and *uncomputable* [49]. Intuitively speaking, a function is computable if it could be implemented on an ideal computer with no limitations on time or memory. A precise definition can be given using Turing machines [8]. The restriction to natural numbers is inconveniently limiting for our purposes, so let us introduce the set $\mathbb{D}$ (standing for *data*) to denote all values that will be used for inputs or outputs of computable functions.

Definition 2. Let $\mathbb{D}$ be the smallest set such that $\mathbb{Z} \cup \mathbb{S} \cup \mathbb{B} \cup \textsc{List}(\mathbb{D}) \subseteq \mathbb{D}$ and $(x_1, \ldots, x_k) \in \mathbb{D}$ for all $x_1, \ldots, x_k \in \mathbb{D}$.

In other words, the set $\mathbb{D}$ is built by starting with integers, strings, and Booleans, and then recursively adding all lists and tuples whose members belong to $\mathbb{D}$. This allows the elements of $\mathbb{D}$ to be rather complex. For example, if

$$d = \big(-2,\ \big[\,(\text{“x”}, \textsc{True}),\ (\text{“y”},\ \textsc{False})\,\big]\,\big), \tag{3}$$

then $d \in \mathbb{D}$. We will also need an injective function $s : \mathbb{D} \to \mathbb{S}$ that gives a *string encoding* for all members of $\mathbb{D}$. For $n \in \mathbb{N}$, we define $s(n)$ as the standard decimal representation of n, so that, for example, $s(314) =$ “`314`”. In more complicated cases, such as (3), one possible string encoding would be

$$s(d) = \text{“}\texttt{(-2, [("x", True), ("y", False)])}\text{”}.$$

A precise definition of s is omitted. As remarked by Poonen [43, p. 2], it is not necessary to specify the encoding as long as it is clear that a Turing machine could convert between reasonable encodings imagined by two different readers.

Definition 3. Suppose that $X_1, \ldots, X_k, Y \subseteq \mathbb{D}$. A function $f : X_1 \times \ldots \times X_k \to Y$ is called *computable function* if there exists a Turing machine that can produce the string $s(f(x_1, \ldots, x_k))$ when given the strings $s(x_1), \ldots, s(x_k)$ as input.

For example, the function $+ : \mathbb{N}^2 \to \mathbb{N}$ is computable, because $\mathbb{N} \subseteq \mathbb{D}$ and there exists a Turing machine that, for any $a, b \in \mathbb{N}$, can produce the string $s(a + b)$ when given the strings $s(a)$ and $s(b)$ as input. When given the strings “`59`” and “`84`” as input, the Turing machine must produce the string $s(59 + 84) =$ “`143`” as output. When the sets $X_1, \ldots, X_k$, and Y are all equal to $\mathbb{N}$, computable functions are often called *recursive*. However, Soare gives a historical and conceptual analysis of computability and recursion, and recommends the term *computable* even in this case [49].

The set $\mathbb{D}$ could be extended even further, but note that the requirement of a string encoding means that $\mathbb{D}$ has to be countable. Thus, the classification of functions into computable and uncomputable does not apply to, for example, real functions ($\mathbb{R} \to \mathbb{R}$). Computable functions are very common in the sense that the vast majority of the most commonly appearing functions are computable. From another perspective they are very rare, since the number of functions $\mathbb{D}^k \to \mathbb{D}$ is uncountable, but the number of all programs, and therefore, the number of all computable functions, is only countable.

Having dealt with functions, we will now define what is meant by computable and computably enumerable *sets*. Just as with computable functions, both of these concepts are often defined only for subsets of natural numbers, but we adopt an extended definition by defining them as subsets of $\mathbb{D}$.

Definition 4. A set $S \subseteq \mathbb{D}$ is *computable* if its characteristic function $\phi : \mathbb{D} \to \mathbb{B}$, $\phi(x) = (x \in S)$ is a computable function.

As an example, $\mathbb{Z}^+ \subseteq \mathbb{D}$ and the set $s(\mathbb{Z}^+) = \{``\texttt{1}", ``\texttt{2}", ``\texttt{3}", \ldots\}$ consists of all nonempty strings whose all characters are digits, and the first character is not "0". It is possible to write a computer program that checks for this, so $\mathbb{Z}^+$ is a computable set.

Definition 5. A set $S \subseteq \mathbb{D}$ is *computably enumerable (c.e.)* if we have $S = \emptyset$ or $S = f(\mathbb{N}) = \{f(0), f(1), f(2), \ldots\}$ for some computable function $f : \mathbb{N} \to \mathbb{D}$.

Computable and computably enumerable sets can also be called *recursive* and *recursively enumerable* sets, or *decidable* and *semidecidable* sets. In Definition 5, we can think of the function f as an infinitely running computer program that generates a stream of values $(f(0), f(1), f(2), \ldots)$, and S as the set of all values that f will generate at least once. After the program has generated the values $f(0), \ldots, f(n)$ for some $n \in \mathbb{N}$, we know that at least these values belong to the set S. For any other value in $\mathbb{D}$ we do not know in general if it is a value that will eventually appear, or a value that never will.

We state without proof a few basic properties of computable and computably enumerable sets. The first three are straightforward to prove, but would require us to specify an explicit string encoding. The remaining properties are standard results, whose proofs can be found in [8].

1. The sets $\mathbb{Z}^+$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{S}$, and $\mathbb{B}$ are computable sets.
2. If $S \subseteq \mathbb{D}$ is a computable (respectively, c.e.) set, then $\textsc{List}(S)$ is a computable (resp., c.e.) set.
3. If $S_1, \ldots, S_k \subseteq \mathbb{D}$ are computable (resp., c.e.) sets, then $S_1 \times \ldots \times S_k$ is a computable (resp., c.e.) set.
4. If $S \subseteq \mathbb{D}$ is a computable (resp., c.e.) set and $\phi : S \to \mathbb{B}$ is a computable function, then the restriction $\{x \in S \mid \phi(x)\}$ is a computable (resp., c.e.) set.
5. If $S \subseteq \mathbb{D}$ is finite, then it is a computable set.
6. If $S \subseteq \mathbb{D}$ is a computable set, then it is also a computably enumerable set.
7. Suppose that $S \subseteq T \subseteq \mathbb{D}$, and that T is a computable set. If S and $T \setminus S$ are computably enumerable sets, then both are also both computable sets.

2.6 Operations and relations

A function $f : A \times B \to C$ with two inputs is called a *binary function.* There are several binary functions that traditionally use *infix notation*: instead of writing $f(a, b)$ we choose some symbol $\star$, and write $a \star b$ instead. Two important special cases include *binary operations* and *relations.* A *relation*, as we have already defined, is a predicate with two inputs, or in other words, a function of the type $A \times B \to \mathbb{B}$. We give two different definitions of a binary operation. A *binary operation in the strict sense* is a function of the type $A^2 \to A$ [6, p. 38]. This does not quite capture all common use of the word *operation*. Matrix multiplication, function composition, and dot product, which are listed in Table 1, are often called operations although they do not satisfy the strict definition. Therefore, we define a *binary operation in the extended sense* to be any binary function that is not a relation. One reason why the distinction between operations and relations is important is that *chaining* works differently for them. If $\star$ is an operation, then the expression $a \star b \star c$ is equivalent to $(a \star b) \star c$, or sometimes $a \star (b \star c)$. If $\sim$ is a relation, then $a \sim b \sim c$ is equivalent to saying that both of the statements $a \sim b$ and $b \sim c$ are true.

A binary operation $\star$ is *associative* if $(x \star y) \star z = x \star (y \star z)$ for all x, y, and z. If an operation is associative, we can drop the parentheses, and write $x \star y \star z$ without ambiguity. The way that mathematicians generally use the term *associative* is not entirely consistent. The standard definition of associativity is given for the strict definition of a binary operation, since having a function of type $A^2 \to A$ guarantees that both $(x{\star}y){\star}z$ and $x{\star}(y{\star}z)$ are defined. Despite this, mathematicians also use the word *associative* for binary operations in the extended sense, such as for matrix multiplication and function composition. This remark does not directly affect the remainder of this thesis, because all binary operations to be used later are binary operations in the strict sense, but it serves as an example of a more general phenomenon. Formal textbooks have to be more careful with definitions than traditional textbooks, since they cannot contain anything that is technically incorrect even in a relatively harmless way. A formal textbook cannot define the term *associative* in the strict sense, and then use it in the extended sense. This implies that formal textbooks cannot always use the same definitions that are given in traditional textbooks.

Chaining of binary relations works even if the relations are not the same. Suppose that we have the relations $\overset{1}{\sim} : A \times B \to \mathbb{B}$ and $\overset{2}{\sim} : B \times C \to \mathbb{B}$. If $(a, b, c) \in A \times B \times C$, then $a \overset{1}{\sim} b \overset{2}{\sim} c$ is an abbreviation of $(a \overset{1}{\sim} b)$ and $(b \overset{2}{\sim} c)$. It is best not to overuse the chaining of different relations. As a somewhat extreme example, the statement $0 \leq 1 < 2 = 2 \in \mathbb{Z} \subset \mathbb{Q} \subseteq \mathbb{R} \neq \mathbb{C}$ has seven different relations chained together, but readability begins to suffer.

Name	Symbol	Associative	Type
Addition of real numbers	$+$	Yes	$\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
Subtraction of real numbers	$-$	No	$\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
Multiplication of real numbers	$\cdot$	Yes	$\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
Union of sets	$\cup$	Yes	$\textsc{Set}(\tau) \times \textsc{Set}(\tau) \to \textsc{Set}(\tau)$
Intersection of sets	$\cap$	Yes	$\textsc{Set}(\tau) \times \textsc{Set}(\tau) \to \textsc{Set}(\tau)$
Multiplication of real matrices	$\cdot$	Yes	$\mathbb{R}^{k \times m} \times \mathbb{R}^{m \times n} \to \mathbb{R}^{k \times n}$
Function composition	$\circ$	Yes	$(\beta \to \gamma) \times (\alpha \to \beta) \to (\alpha \to \gamma)$
Dot product	$\cdot$	N/A	$\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$

Table 1: Common binary operations. Note that different operations sometimes use the same symbol.

2.7 Logical operations and relations

The functions $\vee, \wedge, \rightarrow, \leftrightarrow : \mathbb{B}^2 \rightarrow \mathbb{B}$ and $\neg : \mathbb{B} \rightarrow \mathbb{B}$ are defined as follows [40, p. 12]:

Name	Symbol	Read as	Definition
Disjunction	$\vee$	"or"	If $p = q = \textsc{False}$, then $(p \vee q) = \textsc{False}$. Otherwise, $(p \vee q) = \textsc{True}$.
Conjunction	$\wedge$	"and"	If $p = q = \textsc{True}$, then $(p \wedge q) = \textsc{True}$. Otherwise, $(p \wedge q) = \textsc{False}$.
Negation	$\neg$	"not"	$(\neg\textsc{False}) = \textsc{True}$ $(\neg\textsc{True}) = \textsc{False}$
Conditional	$\rightarrow$	"implies"	If $p = \textsc{False}$, then $(p \rightarrow q) = \textsc{True}$. Otherwise, $(p \rightarrow q) = q$.
Biconditional	$\leftrightarrow$	"if and only if"	If $p = q$, then $(p \leftrightarrow q) = \textsc{True}$. Otherwise, $(p \leftrightarrow q) = \textsc{False}$.

The functions $\vee$ and $\wedge$ have a higher precedence than $\rightarrow$ and $\leftrightarrow$, and the function $\neg$ has a higher precedence than $\vee$ and $\wedge$. For example, if $p, q, r \in \mathbb{B}$, then $p \rightarrow q \wedge r$ means $p \rightarrow (q \wedge r)$ rather than $(p \rightarrow q) \wedge r$, and $\neg p \wedge q$ means $(\neg p) \wedge q$ rather than $\neg(p \wedge q)$. The functions $\vee$ and $\wedge$ have equal precedence, so $p \wedge q \vee r$ must written either as $(p \wedge q) \vee r$ or as $p \wedge (q \vee r)$ to clarify which one is meant. The function $\leftrightarrow$ can be seen as a special case of equality $(=)$. In many formalisms, such as the propositional and first-order logics introduced in Chapters 5 and 6, respectively, the use of $=$ for Booleans is not even allowed, and $\leftrightarrow$ must be always used instead. Programming languages, on the other hand, tend to use the same equality symbol for Booleans as well.

There are several logical equivalences that can be used for manipulating expressions that contain $\vee$, $\wedge$, $\neg$, $\rightarrow$, and $\leftrightarrow$. The following statements are true for all $p, q, r \in \mathbb{B}$:

(L$_1$)	$(p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$	(L$_2$)	$(p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$
(L$_3$)	$p \vee q \leftrightarrow q \vee p$	(L$_4$)	$p \wedge q \leftrightarrow q \wedge p$
(L$_5$)	$p \vee (p \wedge q) \leftrightarrow p$	(L$_6$)	$p \wedge (p \vee q) \leftrightarrow p$
(L$_7$)	$p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$	(L$_8$)	$p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$
(L$_9$)	$p \vee \neg p \leftrightarrow \textsc{True}$	(L$_{10}$)	$p \wedge \neg p \leftrightarrow \textsc{False}$
(L$_{11}$)	$(p \leftrightarrow (q \leftrightarrow r)) \leftrightarrow ((p \leftrightarrow q) \leftrightarrow r)$	(L$_{12}$)	$(p \rightarrow q) \leftrightarrow \neg p \vee q$

A straightforward way to verify any of these equivalences is to check the statement for all $2^3 = 8$ combinations of values that the variables p, q, and r can have.

The functions $\vee, \wedge, \rightarrow, \leftrightarrow : \mathbb{B}^2 \rightarrow \mathbb{B}$ can be regarded as binary operations in the strict sense, since they are of the form $A^2 \rightarrow A$. However, they can also be regarded as relations, since they are of the form $A \times B \rightarrow \mathbb{B}$. Therefore, if we want to allow chaining, we must first decide which mode of chaining we want to use. If we decide that the chaining of $\star : \mathbb{B}^2 \rightarrow \mathbb{B}$ works like an operation, then $p \star q \star r$ is equivalent to $(p \star q) \star r$. If we decide that it works like a relation, then $p \star q \star r$ is equivalent to $(p \star q) \wedge (q \star p)$. In this work we chain $\vee$ and $\wedge$ as operations, and $\rightarrow$ and $\leftrightarrow$ as relations. Correspondingly, we call $\vee$ and $\wedge$ *logical operations*, and $\rightarrow$ and $\leftrightarrow$ *logical relations*.

As noted in the property (L$_{11}$), the function $\leftrightarrow : \mathbb{B}^2 \rightarrow \mathbb{B}$ is associative, so we might have also chosen to chain it as an operation. However, if we think of $\leftrightarrow$ as a special case of $=$, it makes more sense to define it as a relation. Chaining $\leftrightarrow$ as a relation allows us to support chains of equivalences, where $p_1 \leftrightarrow \ldots \leftrightarrow p_n$ is an abbreviation of $(p_1 \leftrightarrow p_2) \wedge \ldots \wedge (p_{n-1} \leftrightarrow p_n)$. In particular, this means that ($\textsc{True} \leftrightarrow \textsc{False} \leftrightarrow \textsc{False}$) is false. If we defined $\leftrightarrow$ as an associative operation this would be true instead.

If $\leftrightarrow$ is treated as a logical relation, then it seems natural to handle $\rightarrow$ in the same way. Just as with ordinary relations, the logical relations $\rightarrow$ and $\leftrightarrow$ can be mixed. This means, for instance, that if $a, b, c, d \in \mathbb{B}$, then we take $a \leftrightarrow b \rightarrow c \leftrightarrow d$ to be equivalent to $(a \leftrightarrow b) \wedge (b \rightarrow c) \wedge (c \leftrightarrow d)$. Incidentally, if $\leftrightarrow$ can be seen as a special case of $=$, then $\rightarrow$ could be seen as a special case of $\leq$ with the convention that $\textsc{False} < \textsc{True}$. Therefore, the situation is analogous to how we would read $a = b \leq c = d$ as being equivalent to $(a = b) \wedge (b \leq c) \wedge (c = d)$.

2.8 A language for exponentiation

Before giving a more general picture of formal languages in Chapter 4, we will define a simple statement language EX in order to introduce several key concepts. This language is intended to formalize simple statements about the exponentiation of positive integers, such as $2^3 = 8$ (a true statement) and $52^{301} = 4$ (a false statement). These statements will be formalized by the strings "`2^3 = 8`" and "`52^301 = 4`", but in display notation we can abbreviate these to '$2^3 = 8$' and '$52^{301} = 4$'. To distinguish mathematical statements from their formal representations as strings, the strings are called *formulas*.

Definition 6. (A special case of Definition 14) A *statement language* $L = (\mathcal{F}_L, \models_L)$ consists of a computable set of strings $\mathcal{F}_L \subseteq \mathbb{S}$ and a predicate $\models_L : \mathcal{F}_L \rightarrow \mathbb{B}$. The members of $\mathcal{F}_L$ are called *formulas of* L. If $\phi \in \mathcal{F}_L$, then $\models_L \phi$ can be read as "the formula ϕ is true in the language L."

When we define the set of formulas $\mathcal{F}_L$, we are defining the *syntax* of L. When we define the predicate $\models_L$, we are defining the *semantics* of L. In linguistics, semantics is the study of what languages mean (or refer to), and natural languages can refer to a wide variety of things: think, for example, of the terms *Napoleon*, *green*, *justice*, *physics*, and *Sherlock Holmes*. The languages studied in this thesis only refer to mathematical objects, such as integers, lists, sets, and functions. Accordingly, we use *semantics* as a purely technical term that has a precise mathematical meaning.

We are now ready to define the statement language EX. Again, for $n \in \mathbb{N}$ let $s(n)$ be the standard decimal representation of n, so that, for example, $s(314) =$ "`314`".

Definition 7. (*Syntax of* EX) The set $\mathcal{F}_{\mathsf{EX}}$ consists of all strings that are of the form "$\boxed{s(a)}$`^`$\boxed{s(b)}$` = `$\boxed{s(c)}$" for some $a, b, c \in \mathbb{Z}^+$. In display notation we write such strings as '$\boxed{s(a)}^{\boxed{s(b)}} = \boxed{s(c)}$'.

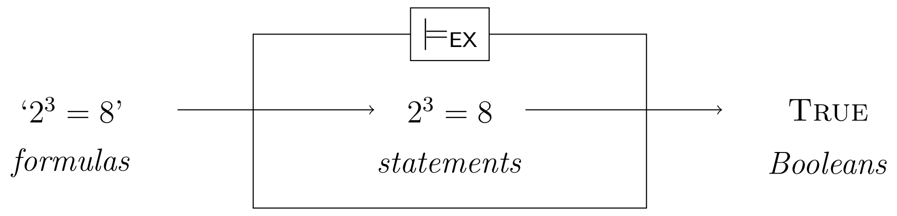
Thus, the strings "`2^3 = 8`" and "`52^301 = 4`" are formulas of EX, and they can be written in display notation as '$2^3 = 8$' and '$52^{301} = 4$'. This illustrates how display notation can handle the superscripts used in exponentiation. We will now define the semantics of EX.

Definition 8. (*Semantics of* EX) If $s \in \mathcal{F}_{\mathsf{EX}}$, then $s =$ '$\boxed{s(a)}^{\boxed{s(b)}} = \boxed{s(c)}$' for certain values $a, b, c \in \mathbb{Z}^+$. The predicate $\models_{\mathsf{EX}} : \mathcal{F}_{\mathsf{EX}} \to \mathbb{B}$ is defined so that $\models_{\mathsf{EX}} s$ if and only if $a^b = c$.

Using Definition 8, we can conclude that since $2^3 = 8$, we have $\models_{\mathsf{EX}}$ '$2^3 = 8$'. This conclusion may seem close to circular, so it may be in order to say a little more of the purpose of Definition 8. When dealing with formal languages it is important not to confuse the language we are studying (the *object language*) with the language we are using to study it (the *metalanguage* or the *observer's language*). The metalanguage used throughout this work is the language of standard symbolic mathematical notation. To be a bit more concrete, we will call it the *language of set theory*, since sets can be used to represent numbers, strings, lists, tuples, functions, and all other objects we will be working with. Therefore, we can say that $2^3 = 8$ is a (true) statement of set theory, although it does not explicitly mention sets at all.

The object language we are studying in this section is the language EX, whose syntax and semantics were defined by using the language of set theory. From the point of view of set theory, formulas of EX are merely strings, and are not considered true or false by themselves. However, the predicate $\models_{\mathsf{EX}}$ classifies certain formulas, such as '$2^3 = 8$', as true, and others, such as '$52^{301} = 4$', as false.

It may be helpful to think of the predicate $\models_{\mathsf{EX}}$ as a combination of two things. When it is given the formula '$2^3 = 8$' it first translates it into the mathematical statement $2^3 = 8$. If the formula were written in input notation, then this translation would be more visible, but in display notation the translation can be done by simply dropping the quotation marks. Second, this statement is evaluated to produce the value True. The reason why we cannot break this single predicate down into two independent functions is that mathematical statements, unlike strings or Booleans, are not something that set-theoretical functions take as an input or produce as an output.



The relationship between EX and set theory is somewhat unusual. We could say that the language EX is, both syntactically and semantically, a *sublanguage* of the language of set theory: syntactically in the sense that every formula of EX in display notation can be read as meaningful statement of set theory, and semantically in the sense that the formula is true in EX if and only if the corresponding statement is true in set theory. This will be made more precise in Chapter 4.

2.9 Ambiguity in set theory

The language EX is relatively simple, and it is clear that every syntactically valid formula of EX has an unambiguous truth value. It is far more problematic to assume that in the language of set theory every syntactically valid statement without free variables has an unambiguous truth value. In particular, the language of set theory can quantify over very large sets, such as $\mathrm{SET}(\mathbb{R})$ and $\mathbb{R} \to \mathbb{R}$, and this can cause problems when trying to interpret the statements of set theory. We will look at two questions that shed some light on this matter:

1. Does *Cauchy's functional equation* have nonlinear solutions? In other words, suppose that $f : \mathbb{R} \to \mathbb{R}$ and $f(x + y) = f(x) + f(y)$ for all $x, y \in \mathbb{R}$. Does there exist such an f that is not of the form $f(x) = ax$ for some $a \in \mathbb{R}$?
2. Does there exist a set $X \subseteq \mathbb{R}$ such that $\aleph_0 < |X| < 2^{\aleph_0}$?

A precise treatment of these questions is far beyond the scope of this thesis, so they will only be presented briefly. It follows from the axiom of choice that Cauchy's functional equation does have nonlinear solutions, but all of these functions are necessarily non-measurable [28]. In 1938 Gödel defined a model of Zermelo–Fraenkel set theory called the *constructible universe*, and showed that the axiom of choice is true in this model [14]. But as shown by Solovay (1970), there also exists a model of Zermelo–Fraenkel set theory in which all real functions are measurable [50]. Roughly speaking, we can assume that nonlinear solutions to Cauchy's functional equation exist, and this does not lead to any inconsistency, but not a single specific example of such a function can be given. It seems fair to say that it is not entirely obvious what does it mean for such functions to "exist".

As for the second question, the assumption that such a set X does not exist is known as the *continuum hypothesis*, and it was posed by Cantor in 1878. Through the work of Gödel (1940) and Cohen (1963) in particular, it is now known that this question is not settled by any usual axiomatizations of set theory, including Zermelo–Fraenkel set theory, the axiom of choice, and several large cardinal axioms [54]. In this case it seems even less clear what does it mean for such a set to "exist". Feferman has argued that the continuum hypothesis is not a definite mathematical problem [11]. Roughly speaking, we could argue that there may be several different "flavors" of existence, and the two questions above are ambiguous, because we have not explicitly said which flavor of existence the quantifier $\exists$ is referring to.

Modern set theory studies and even embraces phenomena like this, but we are simply trying to keep the theoretical foundations of formalized textbooks as simple as possible. Therefore, we only need a rule of thumb that allows us to recognize potentially ambiguous statements. Both of these examples are characterized by quantification over a set whose cardinality is larger than $2^{\aleph_0}$. In the first question we quantify over the set $(\mathbb{R} \to \mathbb{R})$, and in the second question $X \subseteq \mathbb{R}$ is technically an abbreviation of $X \in \mathrm{SET}(\mathbb{R})$, so we quantify over $\mathrm{SET}(\mathbb{R})$. Therefore, let us make the following simplistic assumption:

$$\text{If a set-theoretic statement without free variables does not quantify over sets larger than } 2^{\aleph_0}\text{, then it is unambiguously true or false.} \tag{4}$$

Note that it is still possible for statements that quantify over larger sets to be unambiguous; we just do not assume *a priori* that they are. It is beyond the scope of this thesis to discuss assumption (4) in detail, but for related technical details, see [54] for an argument on why there exist axioms for $H(\omega_1)$ that are as canonical as those of number theory.

The formal languages discussed in this thesis handle the problem of ambiguity in set theory by not using its full expressive power. Instead, they have syntactical limitations that make it impossible for the truth values of statements to depend on quantification over sets whose size is larger than $2^{\aleph_0}$. This makes it impossible to state either of the two example problems given in this section. One possible way of approaching them is to formulate variations where arbitrary subsets and arbitrary real functions by some of their smaller subclasses. The number of continuous real functions is only $2^{\aleph_0}$, because $|\mathbb{Q} \to \mathbb{R}| = 2^{\aleph_0}$, and a continuous real function is uniquely specified by its values on rational numbers. This restriction settles the first problem: all continuous solutions to Cauchy's functional equation are linear [28]. Similarly, the cardinality of all closed subsets of $\mathbb{R}$ is only $2^{\aleph_0}$, because $|\textsc{Set}(\mathbb{Q})| = 2^{\aleph_0}$, and every closed set in $\mathbb{R}$ is uniquely specified by the rational numbers it does not contain. This restriction settles the second problem: if $X \subseteq \mathbb{R}$ is closed, then either $|X| \leq \aleph_0$ or $|X| = 2^{\aleph_0}$. This result is a corollary of the Cantor–Bendixson theorem [29, p. 34].

3 Prior work

In this chapter we look at a few examples of prior work related to formal proofs. In Sections 3.1 and 3.2 we look at the four color theorem and the Kepler conjecture, respectively. The known proofs of these theorems are unusually difficult for humans to verify, so they serve as examples of situations where having a formal proof can greatly increase our confidence in the results. Moreover, if it is feasible to formalize proofs that are as difficult as these, then the comparatively simple proofs in textbooks should be formalizable as well. By using these examples we can argue that the biggest open question about formalized textbook proofs is if they can be written to be readable, not if they can be written at all.

In Section 3.3 we discuss the applications of formal proofs to the formal verification of computer programs. In Section 3.4 we look at Metamath, which is an existing example of an interactive proof explorer. Finally, in Section 3.5 we look at the human-style automated theorem prover by Ganesalingam and Gowers, and see how it relates to our current efforts.

3.1 Four color theorem

A *simple planar map* consists of *regions*: connected and open subsets of $\mathbb{R}^2$ that are pairwise disjoint. The *four color theorem* states that it is always possible to color a simple planar map with only four colors in such a way that adjacent regions have different colors. Some care is required to state precisely what it means for two regions to be adjacent. If we divide a circle to n sectors, then all sectors touch each other at the center of the circle, but they are not all adjacent in the sense meant by the four color theorem. To rule out cases such as this, a *corner* is defined as a point that belongs to the closure of at least three regions, and two regions are *adjacent* if their closures have a common point that is not a corner [15, p. 3].

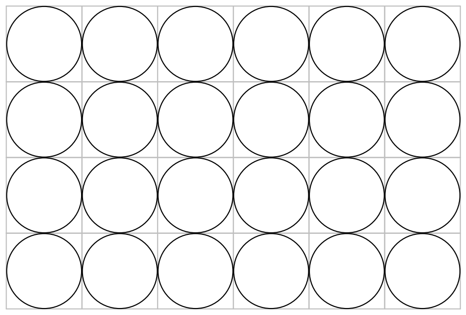
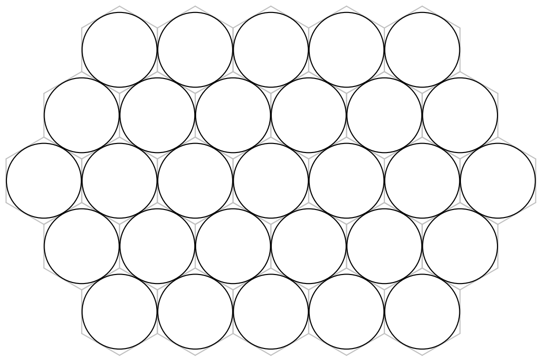
The statement of the four color theorem allows for the number of regions to be infinite, but as will be shown in Section 5.7, if there were an infinite counterexample to the four color theorem, then there would have to be a finite counterexample as well. The basic strategy for proving the four color theorem is to assume that there exists a counterexample that is minimal in some sense, and use it to prove that there has to be a counterexample that is even smaller. Since this contradicts the assumption of minimality, we can conclude that the four color theorem has to be true. As a simple example, a counterexample with the least number of regions cannot contain a region that only has three neighbors. If there is a way to color the other regions using four colors, then we can choose a color that is not used by any of the three neighbors. If there is no way to color the other regions using four colors, then the counterexample is not minimal.

The four color theorem was proved in 1976 by Appel and Haken [3]. They listed a total of 1936 *reducible configurations*, and with the help of a computer, showed that none of them can appear in a minimal triangulated counterexample. To complete the proof, they laboriously verified by hand—in over 400 pages—that a minimal counterexample would have to contain at least one of these reducible configurations.

In 2005 Werner and Gonthier presented a complete formal proof of the four color theorem using the Coq proof assistant [15]. Even though the original proof also used a computer, it used it in a very different way. To verify the original proof one has to check hundreds of pages of case analysis, and all of the computer code. To verify the formal proof, it is only necessary to check the definitions and the statement of the theorem, and to run the proof checker.

3.2 Kepler conjecture

If we want to pack an infinite number of identical circles densely, two configurations quickly come to mind. We can tile the plane with either squares or regular hexagons, and put an inscribed circle inside each polygon. Routine calculations show that the density (proportion of the plane covered by circles) achieved by these packings is $\pi/4 \approx 0.79$ for squares and $\pi/\sqrt{12} \approx 0.91$ for hexagons.

If we want to pack an infinite number of identical *spheres* densely, we could do it by taking infinite cubic or hexagonal layers, and stacking them together as closely as possible. If we use hexagonal layers, then the spheres on a single layer are packed more densely, but if we use cubic layers, the distance between two consecutive layers will be smaller. It turns out that in both cases the density is equal to $\pi/\sqrt{18} \approx 0.74$. Kepler conjectured in 1611 that this is the highest density possible for any arrangement of spheres. The special case that this the highest density achievable by a lattice was proven by Gauss in 1831, but the case for irregular packings remained open [18].

Thomas Hales announced a proof of the Kepler conjecture in 1998, and submitted it to the *Annals of Mathematics*. A team of referees was assigned to it, and the proof was under review for five years [19] before it was accepted for publication [20]. But even at this point, the referees were not completely convinced. In a letter of qualified acceptance for publication, an editor described the process [21]: "The referees put a level of energy into this that is, in my experience, unprecedented. ... They have not been able to certify the correctness of the proof, and will not be able to certify it in the future, because they have run out of energy to devote to the problem." As a result, in early 2003 Hales launched an ambitious project called *Flyspeck* in order to completely formalize the proof of the Kepler conjecture. Completion of the project was announced in August 2014, and an official report of the completed project was published in January 2015 [22].

The complete formal proofs of the four color theorem and the Kepler conjecture are both examples of proofs that are significantly more difficult than the proofs that are typically found in textbooks. They are not given as examples of the kind of proofs that electronic textbooks should contain, but they do show that even difficult proofs can be formalized. The question that remains is if those formal proofs can be made readable.

3.3 Program correctness

Let us suppose that we have written a computer program. Our goal is, of course, that the actual operation of the program corresponds to what we intended the program to do. It is often useful to split this large goal into three smaller subgoals. We should write, in addition to the program itself, a *formal specification* that describes what the program is supposed to do. Our first goal is to make sure that the specification captures our original intentions, and naturally, this can be only done by careful thinking and introspection. The second goal is to make sure that the program, if run on an ideal computer, would satisfy the specification. This part is completely mathematical, and all mathematical methods—including computer-verified proofs—are available. The third and final goal is to make sure that the actual physical computer we are using corresponds to our idealized model of it. This includes trying to make sure that no errors are caused by the compiler, the operating system, or the hardware. Because of the involvement of hardware, this last part is necessarily somewhat empirical. These three goals and their methods of verification are illustrated in Figure 1. We will focus on the mathematical part of checking that a program satisfies a specification. A program that satisfies its specification is considered to be *correct*, and a proof of this is called *proof of correctness*.

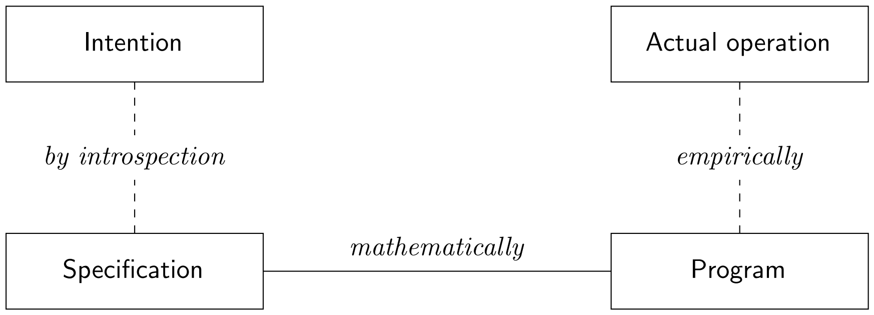


Figure 1: The formal specification should capture the intention, the program should implement the specification, and the actual operation of the physical machine should be faithful to the idealized mathematical model of it.

The proofs of correctness that are typically found in mathematical textbooks are somewhat different than the proofs that are usually found in other applications. For the purposes of this thesis, we can roughly divide formal specifications into three classes:

1. The specification uniquely identifies a mathematical function that the program must implement.
2. The program has to implement some mathematical function, but the specification does not identify it uniquely.
3. The execution of the program must have observable effects beyond returning a value. It is not simply an implementation of a mathematical function.

Much of the work on proofs of correctness is focused on the third class. In contrast, the proofs of correctness that appear in textbooks are most likely to belong to the first or second class, with the first class being most common. As an example from the first class, suppose that we wish to implement a membership predicate for the set of prime numbers $\{2, 3, 5, 7, 11, \ldots\}$. As a formal specification, define $\mathrm{IsPrime} : \mathbb{Z}^+ \to \mathbb{B}$ so that

$$\mathrm{IsPrime}(n) = \big(n \neq 1 \wedge \neg(\exists a, b \in \mathbb{Z}^+ : a > 1 \wedge b > 1 \wedge ab = n)\big). \tag{5}$$

One straightforward—and very inefficient—implementation of this function in the programming language Haskell is

```
IsPrime n = n > 1 && and [n `mod` k /= 0 | k <- [2..n-1]].     (6)
```

A proof of correctness is, then, a mathematical proof that (6) implements the function defined by (5). The specification is not actually very useful in this case, since the implementation is almost as simple as the specification. The value of having a separate specification becomes clearer when the implementation is much more complicated than the specification. When using a more sophisticated algorithm, such as the polynomial time AKS primality test [1], the correctness of the implementation becomes a nontrivial theorem.

Efficient multiplication of large integers is another example of a problem where having a separate specification is clearly useful. The specification of integer multiplication is trivial, but a state-of-the-art implementation can be quite complex. The GMP arithmetic library adaptively employs seven different algorithms for integer multiplication depending on the sizes of the multiplicands [17, p. 93].

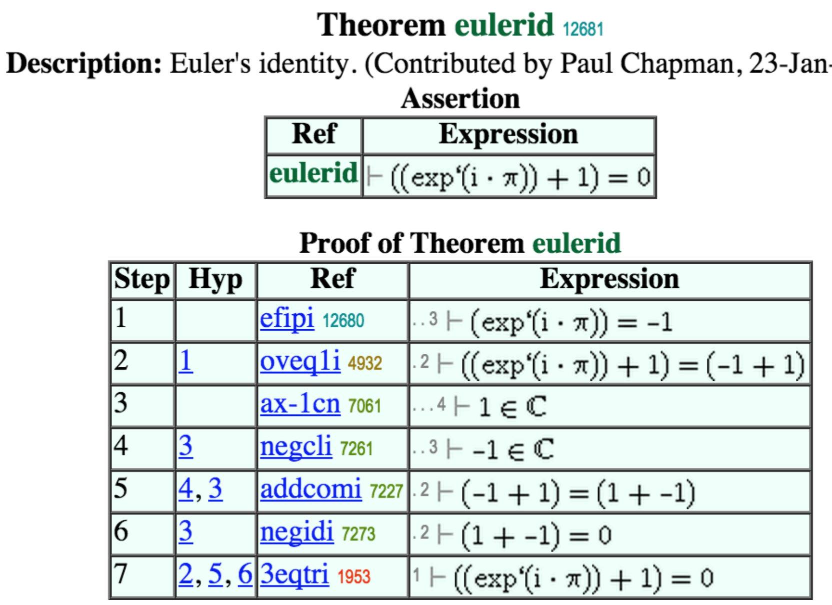
As an example from the second class, a *C compiler* is a program that translates programs written in the programming language C into machine code. There are several acceptable ways of doing this translation. A formal specification of a C compiler should not demand any particular output, but only insist that the translated program is faithful to the original. This requires a formal mathematical description of both the C programming language and the machine code.

CompCert is a formally verified C compiler that has been developed using the Coq proof assistant [35]. The advantage of formal verification is particularly clear when it comes to compiler optimization. Compilers usually perform various transformations to the program in order to make the resulting code as efficient as possible. However, it is extremely difficult to make sure that these transformations and their various combinations do not cause errors in any circumstances. For this reason, safety-critical software is commonly compiled with all optimizations turned off [9, p. 9]. Since CompCert is formally verified, it can achieve extremely high levels of reliability even with optimizations turned on. The effectiveness of this approach has been confirmed by Yang et al. [55], who created a program called Csmith that generates random C programs in order to find bugs in compilers. They report: "As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task."

3.4 Metamath

Metamath is a minimalistic language for formalizing mathematical definitions and proofs [39]. It is based on first-order logic, which will be covered in more detail in Chapter 6. One interesting implementation detail is that the proofs are readable as web pages, and the proofs of the theorems are hyperlinked. The complete Metamath system can therefore be considered to be an interactive proof explorer. The Metamath website contains over 12 000 formal proofs based on the ZFC axioms of set theory.

Because of the simplicity of the design, the proof checker is easy to implement and understand. As a drawback, this means that the proofs are quite low level. Most of the visible steps in the proof of the Euler's identity in Figure 2 are quite



Theorem eulerid 12681

Description: Euler's identity. (Contributed by Paul Chapman, 23-Jan-2008.)

Assertion

Ref	Expression
eulerid	$\vdash ((\exp`(\mathrm{i} \cdot \pi)) + 1) = 0$

Proof of Theorem eulerid

Step	Hyp	Ref	Expression
1		efipi 12680	..3 $\vdash (\exp`(\mathrm{i} \cdot \pi)) = -1$
2	1	oveq1i 4932	.2 $\vdash ((\exp`(\mathrm{i} \cdot \pi)) + 1) = (-1 + 1)$
3		ax-1cn 7061	...4 $\vdash 1 \in \mathbb{C}$
4	3	negcli 7261	..3 $\vdash -1 \in \mathbb{C}$
5	4, 3	addcomi 7227	.2 $\vdash (-1 + 1) = (1 + -1)$
6	3	negidi 7273	.2 $\vdash (1 + -1) = 0$
7	2, 5, 6	3eqtri 1953	1 $\vdash ((\exp`(\mathrm{i} \cdot \pi)) + 1) = 0$

Figure 2: An explorable proof of the Euler's identity $e^{i\pi} + 1 = 0$ in Metamath.

trivial manipulations. Due to its low level proofs, Metamath may be a good tool for explaining what is meant by a formal proof in first-order logic, but at the same time it is not as good for documenting the proofs in human-readable form. Intuitive understanding of why $e^{i\pi} + 1 = 0$ is true may be hard to achieve by reading a proof such as the one displayed in Figure 2.

3.5 Human-style theorem proving

Ganesalingam and Gowers have developed an automated theorem prover that generates *human-style* output. The following proof is an example of the output generated by the program [12]:

Theorem 1. *If $f : X \to Y$ is a continuous function and U is an open subset of Y, then $f^{-1}(U)$ is an open subset of X.*

Proof. Let x be an element of $f^{-1}(U)$. Then $f(x) \in U$. Therefore, since U is open, there exists $\eta > 0$ such that $u \in U$ whenever $d(f(x), u) < \eta$. We would like to find $\delta > 0$ s.t. $y \in f^{-1}(U)$ whenever $d(x, y) < \delta$. But $y \in f^{-1}(U)$ if and only if $f(y) \in U$. We know that $f(y) \in U$ whenever $d(f(x), f(y)) < \eta$. Since f is continuous, there exists $\theta > 0$ such that $d(f(x), f(y)) < \eta$ whenever $d(x, y) < \theta$. Therefore, setting $\delta = \theta$, we are done. $\square$

The program needs two key features to achieve this. First, it clearly needs an algorithm that converts proofs from some internal representation to English prose. Second, and more subtly, the proofs found by the program have to structurally resemble proofs that humans would write.

Both of these features are potentially relevant for formal proofs in textbooks. The ability to automatically find human-style proofs could be useful for the author of the textbook, since this would mean that the computer could automatically generate the low-level parts of the proof. Even so, the textbook author should review the generated proof. If the author is not satisfied with the automatically generated proof, he or she can modify it, or discard it altogether and write a new proof. The proof of the Brahmagupta–Fibonacci identity (1) in Section 1.3 was an example of a situation where it would be hard for a theorem prover to automatically find the proof that the author preferred.

The ability represent mathematical statements in English may also be useful. A symbolic mathematical statement, such as $\neg\mathrm{IsPrime}(2k)$, can be expressed in English as “$2k$ is not prime”. The difference is not large, and somewhat a matter of opinion, but familiarity might make the option “$2k$ is not prime” slightly more preferable. The human-style theorem prover of Ganesalingam and Gowers goes further by using English to represent not only mathematical statements, but also their proofs.

The proof explorer presented in Chapter 8 uses methods like this to convert symbolic statements to their English equivalents, but it does not present proofs as prose. It seems almost certain that the conventional way of writing proofs as paragraphs of text has been influenced by the need to save space in printed books. This does not apply to electronic textbooks in the same way, and different possibilities for displaying the proofs ought to be explored. At the same time, the use of prose proofs in the development of the human-style theorem provers is easily justified. A human-style theorem prover should be evaluated by the proofs that it produces, and comparisons between computer-generated proofs and proofs written by humans are much easier, if the computer writes proofs in the same format as humans currently do.

4 A framework for extensible languages

The formal contents of a textbook consist of definitions, theorem statements, and proofs, and in this chapter we define an extensible framework for discussing the formal languages that are needed to express them. For simplicity, we assume that all definitions in the textbook are *function definitions*, and that all theorems statements are *conditional statements*. We then introduce *function definition languages*, *conditional statement languages*, and *proof languages* in Sections 4.1, 4.2, and 4.3, respectively.

All three of these need some notion of extensibility. As an example, the number of definable functions in any particular function definition language is countable, but the number of all possible functions is not. Therefore, it is not possible to find a fixed language that is adequate for defining all functions that the textbook author might need to define.

The concept of a conditional statement language forms the basis for Chapters 5, 6, and 7. In Chapter 5 we formalize *propositional logic* as the conditional statement language PL, and in Chapter 6 we do the same for *first-order logic*, and end up with the conditional statement language FL, which is an extension of PL. Finally, in Chapter 7 we extend FL further by defining the conditional statement language SD, which is related to *second-order arithmetic*. Like all conditional statement languages, SD has its limitations, and depending on the textbook it may be necessary to extend SD even further by adding new features to it. One of the benefits of having an explicit framework for extensions is that it ensures that new features cannot interfere with the existing ones in SD.

4.1 Function definition languages

There are several kinds of mathematical objects that we may want to define, but for simplicity, this thesis will focus on functions. Let us assume that we are working on with some large set U, which contains all possible inputs and outputs of the functions that our language is able to define.

Definition 9. If U is a set, then $\textsc{Func}(U)$ is the set that consists of all functions $X_1 \times \ldots \times X_k \to Y$ where $k \in \mathbb{N}$ and $X_1, \ldots, X_k, Y \subseteq U$.

Before defining formally what is meant by a *function definition language*, let us look at a motivating example. Define $\mathit{square} : \mathbb{Z} \to \mathbb{Z}$ so that for all $n \in \mathbb{Z}$ we have $\mathit{square}(n) = n \cdot n$, and $\mathit{squareSum} : \textsc{List}(\mathbb{Z}) \to \mathbb{Z}$ so that for all $x_1, \ldots, x_n \in \mathbb{Z}$ we have $\mathit{squareSum}([x_1, \ldots, x_n]) = \mathit{square}(x_1) + \ldots + \mathit{square}(x_n)$. We will now restate these definitions in a formal language, which in this case is a simple programming language. This example also demonstrates that having a separate notation for input and display is possible for programs also. Here "=", or '$\leftarrow$' in display notation, is used as the *variable assignment* operator.

```
square : Int -> Int
square(n):
    return n * n

squareSum : List(Int) -> Int
squareSum(list):
    a = 0
    for x in list:
        a = a + square(x)
    return a
```

```
$\mathit{square} : \mathbb{Z} \to \mathbb{Z}$
$\mathit{square}(n)$:
    **return** $n \cdot n$

$\mathit{squareSum} : \textsc{List}(\mathbb{Z}) \to \mathbb{Z}$
$\mathit{squareSum}(\mathit{list})$:
    $a \leftarrow 0$
    **for** $x$ **in** $\mathit{list}$:
        $a \leftarrow a + \mathit{square}(x)$
    **return** $a$
```

A *function definition language* should capture the idea that these lines of code are syntactically valid, and that they implement the mathematical functions *square* and *squareSum*. The character set used for the set of strings $\mathbb{S}$ does not contain a character for changing lines (the *newline* character), but we can easily represent a multi-line function definition as a list of strings, where each string corresponds to a single line. These two definitions will be represented as a list of lists of strings.

Let $[d_1, d_2] \in \textsc{List}(\textsc{List}(\mathbb{S}))$ be a list containing the two definitions above, so that, for example, the first string in the list d_1 is "`square : Int -> Int`". To formalize the idea of a function definition language L, we first need a set $\mathcal{F}_L \subseteq \textsc{List}(\textsc{List}(\mathbb{S}))$ so that the statement $[d_1, d_2] \in \mathcal{F}_L$ indicates that these definitions are syntactically valid. Second, we need a function $\mathcal{S}_L : \mathcal{F}_L \to \textsc{List}(\textsc{Func}(U))$ so that $\mathcal{S}([d_1, d_2]) = [\mathit{square}, \mathit{squareSum}]$. Finally, we should require that the logical ordering of the definitions is respected: if $[d_1, \ldots, d_n]$ is a list of function definitions, then the syntactical validity or semantics of d_i, where $i \in \{1, \ldots, n-1\}$, cannot depend on any later definition d_j, where $j \in \{i+1, \ldots, n\}$.

Definition 10. A *function definition language* $L = (\mathcal{F}_L, \mathcal{S}_L)$ *on the set* U consists of a nonempty computable set $\mathcal{F}_L \subseteq \textsc{List}(\textsc{List}(\mathbb{S}))$ (the *syntax of* L), and a function $\mathcal{S}_L : \mathcal{F}_L \to \textsc{List}(\textsc{Func}(U))$ (the *semantics of* L). Let $d = [d_1, \ldots, d_n] \in \mathcal{F}_L$. The length of the list $\mathcal{S}_L(d)$ must also be n, so that we can write $\mathcal{S}_L(d) = [s_1, \ldots, s_n]$. In addition, it must be the case that if $k < n$, then we have $[d_1, \ldots, d_k] \in \mathcal{F}_L$ and $\mathcal{S}_L([d_1, \ldots, d_k]) = [s_1, \ldots, s_k]$.

It follows from the above definition that $[\,] \in \mathcal{F}_L$ for all function definition languages L. There exists a unique language L where $[\,]$ is the only member of $\mathcal{F}_L$. This is called the *empty function definition language*, and it corresponds to the case where no new functions can be defined.

Very little has been assumed about function definition languages, but a simple counting argument can show that no fixed function definition language can be expressive enough for all purposes. If U is infinite, then the number of all possible functions $\textsc{Func}(U)$ is uncountable. However, the set $\textsc{List}(\textsc{List}(\mathbb{S}))$ is only countably infinite, so the number of different functions that a function definition language can represent is always countable.

If textbooks use different languages in an uncoordinated way, this can lead to a situation where the same notation is used for subtly different things. This can cause unnecessary confusion to the readers. Traditional textbooks are already guilty of this: as a trivial example, some authors use the symbol $\mathbb{N}$ to refer to the set of nonnegative integers $\{0, 1, 2, \ldots\}$, and others use it refer to the set of positive integers $\{1, 2, 3, \ldots\}$. We could argue that in an ideal world mathematicians would standardize their notation to prevent this.

However, even in an ideal world we cannot assume that all textbooks would use exactly the same function definition language, because no fixed language can be expressive enough for all mathematics. Despite this, there is no reason why different textbooks would have to use incompatible languages.

Definition 11. Suppose that L and K are function definition languages, where $\mathcal{F}_L \subseteq \mathcal{F}_K$ and $\mathcal{S}_L(d) = \mathcal{S}_K(d)$ for all $d \in \mathcal{F}_L$. Then L is a *sublanguage* of K, or alternatively, K is an *extension* of L.

As a thought experiment, we could imagine a standardization organization that is in charge of maintaining a function definition language. This organization would occasionally release new versions of the language in such a way that the new function definition language L_{n+1} is always an extension of the previous version L_n. This would lead to a sequence of versions $(L_1, L_2, L_3, \ldots)$. New versions can add useful new features to the language, but they can also add complexity. However, because of compatibility, different authors can safely choose different sublanguages of the latest language version L_n.

4.2 Conditional statement languages

For the purposes of this thesis it is assumed that theorems in mathematical textbooks are *conditional statements* that consist of three parts: the introduction of variables, zero or more hypotheses that the variables must satisfy, and a conclusion that follows from the hypotheses for all values of the variables. To make the formal definitions easier to follow we start with an example from the conditional statement language SD that is defined in Chapter 7.

A *range declaration* is a string that, informally speaking, introduces one or more variables, and gives them a range of possible values. The string "`x, y in Int`" = '$x, y \in \mathbb{Z}$' is a range declaration in SD, and it introduces variables x and y that can have any integer values. Zero or more range declarations specify a *statement language*, which consists of all syntactically valid *statements*. These statements can contain those free variables that have been declared by the range declarations. As before, the statements of the object language are called *formulas* in order to distinguish them from the statements of the metalanguage. As an example, the range declaration '$x, y \in \mathbb{Z}$' specifies a particular statement language, and the string "`x + y = 8`" = '$x + y = 8$' is a formula of this language. Formulas such as this can be used as both hypotheses and the conclusion in a conditional statement. Finally, the function $\{`x' \mapsto 3, `y' \mapsto 5\}$ is a particular *assignment* of integer values to these variables, and this particular assignment *satisfies* the formula, because $x + y = 8$ is true when $x = 3$ and $y = 5$. With this example in mind, it should be easier to follow the formal definitions.

Before we can represent statements as strings, we need some way of representing variables. Variables in mathematics are often denoted by a single letter, but this would impose an upper limit on the number of variables. Such a limit is undesirable for theoretical, and sometimes even for practical reasons. In traditional mathematical notation a potentially unlimited supply of variable names can be achieved in various ways, such as by using subscripted variables ($x_1, x_2, x_3, \ldots$) or the *prime symbol* ($x', x'', x''', \ldots$). In programming languages, the usual answer is to allow variable names to consist of multiple letters, and this is the approach that will be followed here. Certain words are reserved for special use in the languages PL, FL, and SD, and are disallowed as variable names.

Definition 12. The set of *variable names*, denoted by $\mathbb{V} \subset \mathbb{S}$, consists of all nonempty strings that contain only letters (as defined in Section 2.2), except for the following *reserved words*:

"`in`"	"`Bool`"	"`and`"	"`or`"	"`not`"
"`Set`"	"`forall`"	"`exists`"	"`Nat`"	"`List`"

Definition 13. Suppose that $\mathcal{V} \subseteq \mathbb{V}$. An *assignment on* $\mathcal{V}$ is a function $A : \mathcal{V} \to U$, where U is an arbitrary set.

Definition 14. A *statement language* $L = (\mathcal{V}_L, \mathcal{F}_L, \mathcal{A}_L, \models_L)$ consists of:

1. a finite set of *free variables* $\mathcal{V}_L \subseteq \mathbb{V}$;
2. a computable set of *formulas* $\mathcal{F}_L \subseteq \mathbb{S}$;
3. a nonempty set $\mathcal{A}_L$ of assignments on $\mathcal{V}_L$; and
4. the *satisfiability relation* $\models_L : \mathcal{A}_L \times \mathcal{F}_L \to \mathbb{B}$, where $A \models_L \phi$ can be read as "in the language L, the formula ϕ is true for the assignment A." Alternatively, we may simply say that the assignment A *satisfies* the formula ϕ in L.

The definition given above is more general than Definition 6 in Section 2.8, which did not support the concept of free variables and assignments. Definition 6 can be thought of as a special case where $\mathcal{V}_L = \emptyset$ and the set $\mathcal{A}_L$ is a singleton set, whose only member is the *empty function*: the unique function whose domain is the empty set. In other words, the *empty assignment* is the only possible assignment if the statement language has zero free variables.

The set $\mathcal{V}_L$ is restricted to be finite, because the framework defined in this chapter is meant for studying the formalization of textbooks, and a textbook can only contain a finite number of variables. In other contexts it may be useful to allow for an infinite number of variables, and in Section 5.7 we give one example of this.

Definition 14 assumes so little that we cannot prove many strong theorems about all statement languages. We can, however, introduce some common terminology.

Definition 15. Suppose that L is a statement language and let $\phi \in \mathcal{F}_L$. If $A \models_L \phi$ for all $A \in \mathcal{A}_L$, then ϕ is *L-valid.* If $A \models_L \phi$ for at least one $A \in \mathcal{A}_L$, then ϕ is *L-satisfiable.* If ϕ is not valid, it is *L-invalid.* If ϕ is not satisfiable, it is *L-unsatisfiable.* If $\phi_1, \phi_2 \in \mathcal{F}_L$ are formulas that are satisfied by exactly the same assignments (that is,

$A \models_L \phi_1$ if and only if $A \models_L \phi_2$), then ϕ_1 and ϕ_2 are *L-equivalent*. If the statement language L is clear from context, we can drop the L-prefix, and simply speak of *validity*, *satisfiability*, *invalidity*, *unsatisfiability*, and *equivalence* instead.

The satisfiability relation $A \models_L \phi$ can be naturally extended to cases where instead of a single formula ϕ we have a list of formulas or a set of formulas—possibly even an infinite set of formulas. For any $X \in \textsc{List}(\mathcal{F}_L)$ or $X \in \textsc{Set}(\mathcal{F}_L)$, $A \models_L X$ is defined as $\forall \phi \in X : A \models_L \phi$. In addition to the satisfiability relation, the same symbol $\models_L$ is customarily used for another relation as well.

Definition 16. Suppose that L is a statement language. The *consequence relation* $\models_L : \textsc{Set}(\mathcal{F}_L) \times \mathcal{F}_L \to \mathbb{B}$ is defined so that if $S \in \textsc{Set}(\mathcal{F}_L)$ and $\phi \in \mathcal{F}_L$, then $S \models_L \phi$ (read *ϕ is a L-consequence of S*, or *ϕ follows from S in L*) if and only if all assignments that satisfy the formulas of S also satisfy ϕ. Symbolically, this means that $S \models_L \phi$ is equivalent to

$$\forall A \in \mathcal{A}_L : \big((\forall \phi \in S : A \models_L \phi) \to A \models_L \phi\big).$$

The relation $\models_L : \textsc{List}(\mathcal{F}_L) \times \mathcal{F}_L \to \mathbb{B}$, which works for lists instead of sets, is defined analogously.

We are now ready to give the definition of a *conditional statement language*.

Definition 17. A *conditional statement language* L consists of:

1. a function definition language $\mathcal{D}_L$;
2. for every list of function definitions $d \in \mathcal{D}_L$, a computable set $\mathcal{R}_{L_d} \subseteq \textsc{List}(\mathbb{S})$ of lists of *range declarations*; and
3. for each list of range declarations $r \in \mathcal{R}_{L_d}$, a statement language $L_d(r)$.

Definition 18. For a fixed a conditional statement language L and a list of function definitions $d \in \mathcal{D}_L$, a *conditional statement* is a triple (r, h, c) that consists of a list of *range declarations* $r \in \mathcal{R}_{L_d}$, a list of *hypotheses* $h \in \textsc{List}(\mathcal{F}_{L_d(r)})$, and a *conclusion* $c \in \mathcal{F}_{L_d(r)}$. A conditional statement (r, h, c) is *L_d-true* if $h \models_{L_d(r)} c$. If L_d is clear from context, we can simply say that it is *true*. The set of all conditional statements is denoted by $\mathcal{C}_{L_d}$ and the set of all true conditional statements is denoted by $\mathcal{T}_{L_d} \subseteq \mathcal{C}_{L_d}$.

As an example, let $d = [\,]$. This is a valid list of function definitions for SD, so we have $d \in \mathcal{D}_{\mathsf{SD}}$. The list of strings $r = [\text{‘}x, y \in \mathbb{Z}\text{’}]$ is a valid list of range declarations, so we have $r \in \mathcal{R}_{\mathsf{SD}_d}$. This list of range declarations specifies the statement language

$$\mathsf{SD}_d(r) = (\mathcal{V}_{\mathsf{SD}_d(r)}, \mathcal{F}_{\mathsf{SD}_d(r)}, \mathcal{L}_{\mathsf{SD}_d(r)}, \models_{\mathsf{SD}_d(r)}).$$

Here $\mathcal{V}_{\mathsf{SD}(r)} = \{\text{‘}x\text{’}, \text{‘}y\text{’}\}$. If we define $\phi = \text{‘}x + y = 8\text{’}$ and $A = \{\text{‘}x\text{’} \mapsto 3, \text{‘}y\text{’} \mapsto 5\}$, then ϕ is a formula of $\mathsf{SD}_d(r)$ (that is, $\phi \in \mathcal{F}_{\mathsf{SD}_d(r)}$), A is an assignment of $\mathsf{SD}_d(r)$ (that is, $A \in \mathcal{A}_{\mathsf{SD}_d(r)}$), and A satisfies ϕ (that is, $A \models_{\mathsf{SD}_d(r)} \phi$). The conditional statement

$$([\text{‘}x, y \in \mathbb{Z}\text{’}], [\text{‘}x \geq 2\text{’}, \text{‘}y \geq 3\text{’}], \text{‘}x + y \geq 5\text{’})$$

is SD_d-true. In a traditional printed textbook, this might have been stated as follows:

Suppose that $x, y \in \mathbb{Z}$. If $x \geq 2$ and $y \geq 3$, then $x + y \geq 5$.

The conditional statement languages PL and FL do not support function definitions, so they use the empty function definition language. In these cases d is always equal to $[\,]$, so we can simplify notation by writing simply L instead of L_d.

Note that conditional statements have a much simpler classification than the formulas of a statement language. A formula in a statement language can be valid, invalid, satisfiable, or unsatisfiable. In contrast, conditional statements are simply either true or false.

Finally, we can define the notion of extensibility, which was defined for function definition languages in Definition 11, to statement languages and conditional statement languages. The need for extensibility in conditional statement languages follows directly from the fact that each conditional statement language uses a fixed function definition language. An extension of the function definition language also extends the conditional statement language.

Definition 19. Let L and K be statement languages, and suppose that $\mathcal{F}_L \subseteq \mathcal{F}_K$, $\mathcal{A}_L \subseteq \mathcal{A}_K$, and for all $\phi \in \mathcal{F}_L$ and $A \in \mathcal{A}_L$ we have $A \models_L \phi$ if and only if $A \models_K \phi$. Then L is a *sublanguage* of K, or alternatively, K is an *extension* of L.

Definition 20. Let L and K be conditional statement languages, and suppose that:

1. $\mathcal{D}_L$ is a sublanguage of $\mathcal{D}_K$;
2. $\mathcal{R}_{L_d} \subseteq \mathcal{R}_{K_d}$ for all $d \in \mathcal{D}_L$;
3. $L_d(r)$ is a sublanguage of $K_d(r)$ for all $d \in \mathcal{D}_L$ and $r \in \mathcal{R}_{L_d}$.

Then L is a *sublanguage of* K, or alternatively, K is an *extension of* L. If $L \neq K$, then L is a *proper sublanguage of* K.

It is in the sense of Definition 20 that PL is a proper sublanguage of FL and FL is a proper sublanguage of SD.

4.3 Proof languages

Formal proofs are often defined as sequences of steps, where each step can be encoded as a formula. For the abstract definition below there is no need to make detailed assumptions about the internal structure of proofs. We only assume that the proofs are members of the set $\mathbb{D}$. This minimal assumption is necessary in order to make the requirement that the proof checking predicate is a computable function.

Definition 21. A *proof language P for a conditional statement language L* is a pair $P = (\mathcal{P}_P, \vdash_P)$, where $\mathcal{P}_P \subseteq \mathbb{D}$ is a computable set of *proofs* and the *proof checking predicate* $\vdash_P \colon \mathcal{P}_P \times \mathcal{C}_L \to \mathbb{B}$ is a computable function. If $p \vdash_P (r, h, c)$, then it must be the case that $h \models_{L(r)} c$. We read $p \vdash_P (r, h, c)$ as "in the language P, p is a proof of (r, h, c)."

Definition 22. If P is a proof language for the conditional statement language L, then $\textsc{Prov}(P)$ is the set of all conditional statements that have a proof in P. Symbolically,

$$\textsc{Prov}(P) = \{(r, h, c) \in \mathcal{C}_L \mid p \vdash_P (r, h, c) \text{ for some } p \in \mathcal{P}_P\}.$$

The symbol $\vdash$ is borrowed from logic, but it is used in a slightly different sense. Usually $\vdash$ refers to the *provability relation* with a set of formulas on the left-hand-side and a single formula on the right-hand-side. This is related to using $\models$ as a consequence relation. In fact, in first-order logic the relations $\models$ and $\vdash$ are equivalent ($S \models \phi$ if and only if $S \vdash \phi$), but they are defined in different ways, and the equivalence of the two is nontrivial to prove. In Definition 21 the relation $\vdash_P$ is instead used with a *proof* on the left-hand-side and a *conditional statement* on the right-hand-side. The relation $\vdash$ in the sense of being a provability relation is not needed in this work.

In addition to function definition languages and conditional statement languages, the notion of extensibility can be extended to proof languages also.

Definition 23. Let L_1 be a conditional statement language that is a sublanguage of another conditional statement language L_2. Suppose that P_1 is a proof language for L_1, and that P_2 is a proof language for L_2. If $\mathcal{P}_{P_1} \subseteq \mathcal{P}_{P_2}$, and for all $p \in \mathcal{P}_{P_1}$ we have $p \vdash_{P_1} (r, h, c)$ if and only if $p \vdash_{P_2} (r, h, c)$, then P_1 is a *sublanguage* of P_2, or alternatively, P_2 is an *extension* of P_1.

The need for extensibility follows directly from the fact that every proof language is defined for a particular conditional statement language, and this conditional statement language may have to be extended. However, in the case of proofs there is also a more subtle reason why extensibility is necessary. By Definition 21, every conditional statement that is provable in P has to be true. It is also desirable that every conditional statement that is true can be proven in P. However, depending on the conditional statement language L, there might not exist such a proof language P.

Theorem 2. *Let P be a proof language for a conditional statement language L. Then the set* $\textsc{Prov}(P)$ *is computably enumerable.*

Proof. The set of proofs $\mathcal{P}_P$ and the set of conditional statements $\mathcal{C}_L$ are both computable. Therefore, it is possible to write a program that enumerates all possible pairs of proofs p and conditional statements (r, h, c). For each pair, we can computably check if $p \vdash_P (r, h, c)$. If it is, we output the conditional statement (r, h, c). This nonterminating programs eventually outputs a conditional statement if and only if that statement is provable in P, and by definition, the output set of such a nonterminating program is computably enumerable. $\square$

As a corollary, if the set of true conditional statements $\mathcal{T}_L$ is not computably enumerable, then we cannot hope to find a proof language P that completely captures the notion of truth in L. The converse also holds.

Theorem 3. *If L is a conditional statement language, then the following are equivalent:*

(1) *There exists a proof language P such that* $\textsc{Prov}(P) = \mathcal{T}_L$.

(2) *The set of true conditional statements $\mathcal{T}_L$ is computably enumerable.*

Proof. The set $\textsc{Prov}(P)$ is always computably enumerable. If there exists a proof language P such that $\textsc{Prov}(P) = \mathcal{T}_L$, then $\mathcal{T}_L$ must also be computably enumerable. Therefore, (1) implies (2).

To prove that (2) implies (1), assume that $\mathcal{T}_L$ is computably enumerable. In the trivial case $\mathcal{T}_L = \emptyset$ we can define a proof language P where $p \vdash_P (r, h, c)$ is always false. If $\mathcal{T}_L \neq \emptyset$, then by Definition 5 there exists a computable function $f : \mathbb{N} \to \mathcal{C}_L$ whose range is

$$f(\mathbb{N}) = \{f(0), f(1), f(2), \ldots\} = \{(r_0, h_0, c_0), (r_1, h_1, c_1), (r_2, h_2, c_2), \ldots\} = \mathcal{T}_L.$$

Now we can simply use the natural number n as the “proof” of the conditional statement $f(n) = (r_n, h_n, c_n)$. Specifically, define the proof language $P = (\mathbb{N}, \vdash_P)$ so that for $p \in \mathbb{N}$ and $(r, h, c) \in \mathcal{C}_L$ the statement $p \vdash_P (r, h, c)$ is true if and only if $f(p) = (r, h, c)$. Now the set of provable conditional statements $\textsc{Prov}(P)$ equals the range of f, which is $\mathcal{T}_L$. □

As will be shown in later chapters, the set $\mathcal{T}_{\mathsf{PL}}$ is computable, and the set $\mathcal{T}_{\mathsf{FL}}$ is computably enumerable, but not computable. The set $\mathcal{T}_{\mathsf{SD}}$ not even computably enumerable. To see what consequences this has, suppose that that $(r, h, c) \in \mathcal{C}_L$ for $L \in \{\mathsf{PL}, \mathsf{FL}, \mathsf{SD}\}$. If $L = \mathsf{PL}$, then there exists an algorithm that can tell if $r \models_{\mathsf{PL}(r)} c$. If $L = \mathsf{FL}$, then no such algorithm exists, but there does exist a proof language P so that $\textsc{Prov}(P) = \mathcal{T}_{\mathsf{FL}}$. If $L = \mathsf{SD}$, then there does not even exist a proof language P such that $\textsc{Prov}(P) = \mathcal{T}_{\mathsf{SD}}$. The fact that there are conditional statement languages whose notion of truth cannot be captured by any fixed proof language explains why the idea of extensibility can be necessary for proofs even if the conditional statement language is fixed.

5 Propositional logic

In this chapter we formalize *propositional logic* as the conditional statement language PL. Propositional logic is an example of a language that mathematicians are much more likely to study as an object language than to use as a metalanguage. It is far too weak to formalize any nontrivial textbook of mathematics. However, because of its simplicity, it can serve as a motivating example for later chapters.

In Section 5.1 we define the syntax of PL in input notation, and in Section 5.2 we describe the corresponding display notation. In Section 5.3 we define the semantics of PL by defining *truth assignments* and *satisfiability.* In Section 5.4 we make it possible to omit superfluous parentheses by defining the conditional statement language PLE, which is an extension of PL. In Section 5.5 we describe the *Boolean satisfiability problem*, which has many applications. One of them, the *Erdős discrepancy problem*, is discussed in more detail in Section 5.6. Finally, in Section 5.7 we look at a more theoretical application of propositional logic, and use the *compactness theorem of propositional logic* to prove that if there is an infinite counterexample to the four color theorem, then there has to be a finite counterexample as well.

Logic is the study of valid reasoning, and to make reasoning visible we have to present it in some language. Classically, this is done using natural languages, but in this work we are mostly concerned with reasoning that happens in the symbolic language of mathematics. However, we start with a brief overview of logic in natural languages, since much of the terminology comes from there. In philosophical logic, a *statement* is a declarative sentence, or part of a sentence, that is capable of having a truth value [31]. The English sentences

1. "If it is raining, then the sky is cloudy."
2. "The sky is not cloudy."

are statements, but the part "it is raining" is also by itself a statement. If we assume that both of the statements above are true, then we can conclude that the statement "it is not raining" also has to be true. This conclusion follows logically from the assumptions, since it does not depend on anything else we might happen to know about rain and clouds. The rules of the English language are rather complex, so it is convenient to study some simplified model of it instead. In the case of *propositional logic* we ignore everything about the statements that is not expressible in terms of $\vee$, $\wedge$, $\neg$, $\rightarrow$, and $\leftrightarrow$. In this case we might introduce two Boolean variables *raining* and *cloudy.* Now, if the statements $\mathit{raining} \rightarrow \mathit{cloudy}$ and $\neg\mathit{cloudy}$ are true, then we can conclude that the statement $\neg\mathit{raining}$ is also true.

5.1 Syntax

Let us start with an example of a theorem that we should be able to formalize using the language PL. This theorem is only intended to serve as an example. It has no interesting consequences, and its proof is trivial. It also contains more parentheses than we would normally use, but we return to this issue in Section 5.4.

Theorem 4. *Suppose that $x, y, z \in \mathbb{B}$. If $(x \vee (y \wedge \textsc{False}))$ and $(\neg(z) \wedge y)$, then $(x \wedge y)$.*

Proof. Since $(x \vee (y \wedge \textsc{False}))$ is true even though $(y \wedge \textsc{False})$ is always false, x must be true. And since $(\neg(z) \wedge y)$ is true, we also know that y must be true. Therefore, $(x \wedge y)$ is true. □

Theorem 4 might have also been labeled a *proposition* instead of a *theorem.* According to Timothy Gowers this is indicates a theorem that is slightly "boring" in the sense that what is proved is not surprising, and the proof is not particularly difficult [16, p. 73]. However, in the context of logic the word *proposition* is already used in at least two other senses. Some writers use it synonymously with the word *statement* [40, p. 12]. Others say that every statement *expresses* a proposition, but different statements may express the same proposition [47, p. 8]. To avoid confusion we do not use the term *proposition* at all. In particular, we use the word *theorem* as a neutral technical term with no connotations of importance or nontriviality.

Since the character set for strings $\mathbb{S}$ (defined in Section 2.2) does not contain the symbols $\in$, $\mathbb{B}$, $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$, we will replace them with "`in`", "`Bool`", "`and`", "`or`", "`not`", "`->`", and "`<->`", respectively, in the input notation. Thus, the main parts of Theorem 4 will be formalized as

$$\begin{aligned} r &= [\text{“}\texttt{x, y, z in Bool}\text{”}] \\ h_1 &= \text{“}\texttt{(x or (y and False))}\text{”} \\ h_2 &= \text{“}\texttt{(not(z) or y)}\text{”} \\ c &= \text{“}\texttt{(x and y)}\text{”}. \end{aligned} \tag{7}$$

Therefore, to comply with this example, we need to define the range declarations of PL in such a way that $r \in \mathcal{R}_{\mathsf{PL}}$ and the formulas of PL in such a way that $h_1, h_2, c \in \mathcal{F}_{\mathsf{PL}(r)}$. In addition, we have to define the satisfiability relation $\models_{\mathsf{PL}}$ in such a way that $[h_1, h_2] \models_{\mathsf{PL}(r)} c$. Our first step is to define the syntactically valid range declarations.

Definition 24. The set $\mathcal{R}_{\mathsf{PL}} \subseteq \textsc{List}(\mathbb{S})$ is defined so that $r \in \mathcal{R}_{\mathsf{PL}}$ if and only if:

1. Each string in the list r consists of one or members of $\mathbb{V}$ separated by "`, `" and followed by "` in Bool`".
2. The variable names found in r are distinct.

It is straightforward to check if a given list of strings belongs to $\mathcal{R}_{\mathsf{PL}}$. For example, the strings "`x`", "`y`", and "`z`" are nonempty, distinct, not reserved words, and consist of nothing but letters. Therefore, if r is defined as in (7), then $r \in \mathcal{R}_{\mathsf{PL}}$. In fact, it is straightforward to write a program that checks for these properties. This implies that the set $\mathcal{R}_{\mathsf{PL}}$ is computable, just as Definition 17 requires it to be. The reason why Definition 24 requires variable names to be distinct is mainly stylistic. There is no advantage in duplicating a variable declaration, and doing so could only cause unnecessary confusion to the reader.

Definition 25. If $r \in \mathcal{R}$, then the set $\mathcal{V}_{\mathsf{PL}(r)}$ contains all variable names that are listed in the range declarations.

For example, if r is defined as in (7), then $\mathcal{V}_{\mathsf{PL}(r)} = \{$“`x`”, “`y`”, “`z`”$\}$. Based on this we can now define the set of all formulas that can contain members of $\mathcal{V}_{\mathsf{PL}(r)}$ as free variables.

Definition 26. Supposet that $r \in \mathcal{R}_{\mathsf{PL}}$. The set $\mathcal{F}_{\mathsf{PL}(r)} \subseteq \mathbb{S}$ of *propositional formulas* is the smallest set of strings that satisfies the following:

1. $\mathcal{V}_{\mathsf{PL}(r)} \cup \{$“`True`”, “`False`”$\} \subseteq \mathcal{F}_{\mathsf{PL}(r)}$.
2. If $a, b \in \mathcal{F}_{\mathsf{PL}(r)}$, then “`(`$\textcircled{a}$ `or` $\textcircled{b}$`)`” $\in \mathcal{F}_{\mathsf{PL}(r)}$.
3. If $a, b \in \mathcal{F}_{\mathsf{PL}(r)}$, then “`(`$\textcircled{a}$ `and` $\textcircled{b}$`)`” $\in \mathcal{F}_{\mathsf{PL}(r)}$.
4. If $a \in \mathcal{F}_{\mathsf{PL}(r)}$, then “`not(`$\textcircled{a}$`)`” $\in \mathcal{F}_{\mathsf{PL}(r)}$.
5. If $a, b \in \mathcal{F}_{\mathsf{PL}(r)}$, then “`(`$\textcircled{a}$ `->` $\textcircled{b}$`)`” $\in \mathcal{F}_{\mathsf{PL}(r)}$.
6. If $a, b \in \mathcal{F}_{\mathsf{PL}(r)}$, then “`(`$\textcircled{a}$ `<->` $\textcircled{b}$`)`” $\in \mathcal{F}_{\mathsf{PL}(r)}$.

If we are given some string $s \in \mathbb{S}$ and a list of range declarations $r \in \mathcal{R}_{\mathsf{PL}}$, it is straightforward to check if $s \in \mathcal{F}_{\mathsf{PL}(r)}$. We give one example of such an argument, and omit them in the future.

Theorem 5. *If r and h_1 are defined as in* (7)*, then $h_1 \in \mathcal{F}_{\mathsf{PL}(r)}$.*

Proof. We have $\mathcal{V}_{\mathsf{PL}(r)} = \{$“`x`”, “`y`”, “`z`”$\}$. If $a =$ “`y`” and $b =$ “`False`”, then we have $a, b \in \mathcal{F}_{\mathsf{PL}(r)}$, and now “`(`$\textcircled{a}$ `and` $\textcircled{b}$`)`” $=$ “`(y and False)`” $\in \mathcal{F}_{\mathsf{PL}(r)}$. Similarly, if we define $c =$ “`x`” and $d =$ “`(y and False)`”, then we have $c, d \in \mathcal{F}_{\mathsf{PL}(r)}$. Consequently, $h_1 =$ “`(`$\textcircled{c}$ `or` $\textcircled{d}$`)`” $=$ “`(x or (y and False))`” $\in \mathcal{F}_{\mathsf{PL}(r)}$. $\square$

5.2 Display notation

All of the range declarations and propositional formulas thus far have been written in input notation, but we will now define a display notation for them. Intuitively, we want to replace “`in`” by ‘$\in$’, and do other similar replacements, but only when the strings appear “on their own”. In particular, the variable “`raining`” should not become ‘$ra{\in}ing$’ in display form.

Definition 27. In Boolean range declarations and propositional formulas, a *token* is a nonempty substring that:

1. belongs to the set $\{$“`->`”, “`<->`”, “`(`”, “`)`”, “`,`”$\}$; or
2. consists of letters only and is maximally long.

Under this definition, we can represent a formula or a range declaration as a list of tokens instead of a string of characters. The formula “`(p and (q or False))`” can be represented as the list of tokens

$$[\text{“}\texttt{(}\text{”}, \text{“}\texttt{p}\text{”}, \text{“}\texttt{and}\text{”}, \text{“}\texttt{(}\text{”}, \text{“}\texttt{q}\text{”}, \text{“}\texttt{or}\text{”}, \text{“}\texttt{False}\text{”}, \text{“}\texttt{)}\text{”}, \text{“}\texttt{)}\text{”}].$$

In general, a program that transforms a string to a list of tokens is called a *lexer*. The reason why this is a better representation is that it allows us to ignore the fact that variable names may contain multiple characters, since each of them is represented by a single token.

Some tokens have a special role in the statement language PL. In display form, the tokens are replaced by symbols as follows:

Token	Symbol	Token	Symbol	Token	Symbol
“`in`”	‘$\in$’	“`Bool`”	‘$\mathbb{B}$’	“`and`”	‘$\wedge$’
“`or`”	‘$\vee$’	“`not`”	‘$\neg$’	“`->`”	‘$\rightarrow$’
“`<->`”	‘$\leftrightarrow$’	“`True`”	‘TRUE’	“`False`”	‘FALSE’
“`(`”	‘(’	“`)`”	‘)’	“`,`”	‘,’

The remaining tokens are used as variable names. Most of them will be simply displayed using an italic font, so that we have

$$\text{“}\texttt{(x and (y or False))}\text{”} = \text{‘}(x \wedge (y \vee \textsc{False}))\text{’}.$$

In addition, we can choose any number of variable names that should be replaced by special symbols in display notation. This is where can easily add support for all Greek letters: “`alpha`” = ‘α’, “`beta`” = ‘β’, ... A complete list of display substitutions performed by the proof explorer is unimportant for the purposes of this thesis, so it is omitted.

Now that the definition of a token has been given, it is possible to compare the definition given in this chapter to the standard definition of propositional logic. Propositional formulas are usually defined as strings of *symbols* rather than strings of characters [40, p. 12]. The essential difference is that the set of different symbols can be countably infinite, but our set of characters is finite. Since it is possible to have a unique symbol for each variable, it is not necessary to construct variable names from several characters, and tokens become unnecessary. In fact, we could think of symbols as the abstract counterpart of tokens, or as tokens whose internal string representation has been forgotten. It is meaningful to ask what is the first character of a token, but not meaningful to ask what is the first character of a symbol. The only thing we are allowed to know about symbols is that there is a countably infinite supply of them, and that we can compare two symbols to see if they are equal or not.

The decision to define propositional formulas as strings of characters instead of strings of symbols reflects a different focus than the usual one. Propositional formulas are often used for theoretical study, and in such cases it makes little difference if the infinite supply of variables is constructed by using subscripts $(x_1, x_2, x_3, \ldots)$, prime notation $(x', x'', x''', \ldots)$, or multi-letter variable names. Therefore, it is better to bypass characters and tokens, and just use symbols directly. However, from the point of view of human readability this choice is far from irrelevant.

5.3 Semantics

To define the semantics of the conditional statement language PL we need, for every list of range declarations $r \in \mathcal{R}_{\mathsf{PL}}$, a set of assignments $\mathcal{A}_{\mathsf{PL}(r)}$ and the satisfiability relation $\models_{\mathsf{PL}(r)} : \mathcal{A}_{\mathsf{PL}(r)} \times \mathcal{F}_{\mathsf{PL}(r)} \to \mathbb{B}$ between assignments and propositional formulas.

Definition 28. Let $r \in \mathcal{R}_{\mathsf{PL}}$. The set of assignments $\mathcal{A}_{\mathsf{PL}(r)}$ for the statement language $\mathsf{PL}(r)$ is the set of predicates $\mathcal{V}_{\mathsf{PL}(r)} \to \mathbb{B}$. These are called *truth assignments*.

Definition 29. Suppose that $r \in \mathcal{R}_{\mathsf{PL}}$ and $A \in \mathcal{A}_{\mathsf{PL}(r)}$. The satisfiability relation $\models_{\mathsf{PL}(r)} : \mathcal{A}_{\mathsf{PL}(r)} \times \mathcal{F}_{\mathsf{PL}(r)} \to \mathbb{B}$, which is abbreviated to just $\models$ below, is defined recursively as follows:

$$(T \models \phi) = \begin{cases} \textsc{True}, & \text{if } \phi = \text{`}\textsc{True}\text{'}; \\ \textsc{False}, & \text{if } \phi = \text{`}\textsc{False}\text{'}; \\ T(\phi), & \text{if } \phi \in \mathcal{V}_{\mathsf{PL}(r)}; \\ (T \models a) \wedge (T \models b), & \text{if } \phi = \text{`}(\text{ⓐ} \wedge \text{ⓑ})\text{' for some } a, b \in \mathcal{F}_{\mathsf{PL}(r)}; \\ (T \models a) \vee (T \models b), & \text{if } \phi = \text{`}(\text{ⓐ} \vee \text{ⓑ})\text{' for some } a, b \in \mathcal{F}_{\mathsf{PL}(r)}; \\ \neg(T \models a), & \text{if } \phi = \text{`}\neg(\text{ⓐ})\text{' for some } a \in \mathcal{F}_{\mathsf{PL}(r)}; \\ (T \models a) \to (T \models b), & \text{if } \phi = \text{`}(\text{ⓐ} \to \text{ⓑ})\text{' for some } a, b \in \mathcal{F}_{\mathsf{PL}(r)}; \\ (T \models a) \leftrightarrow (T \models b), & \text{if } \phi = \text{`}(\text{ⓐ} \leftrightarrow \text{ⓑ})\text{' for some } a, b \in \mathcal{F}_{\mathsf{PL}(r)}. \end{cases}$$

This definition is slightly different from the usual definitions of propositional logic. Usually, the logical symbols '$\vee$', '$\wedge$', '$\neg$', '$\to$', and '$\leftrightarrow$' only exist in the object language, and their meaning is defined by using ordinary English words, such as "and", "or", and "not". In this thesis we do not think of propositional logic as a simplified model of English, or any other natural language; we think of it as a sublanguage of set theory. Therefore, all symbols of the object language have to exists in the metalanguage as well. When the symbol '$\wedge$' is used in a propositional formula, it follows the definition above. When the same symbol is used outside a propositional formula, it follows the definition given in Section 2.7. The fact that our object language is a sublanguage of our metalanguage is the reason why the use quotation marks is so crucial for the current purposes: they are the only thing that lets us distinguish statements of the metalanguage from formulas of the object language.

Now that the satisfiability relation has been defined, Definition 16 automatically gives us a consequence relation as well. As a sanity check, we now prove that the formalization of Theorem 4 given in (7) works as expected.

Theorem 6. *If r, h_1, h_2, and c are defined as in* (7)*, then* $[h_1, h_2] \models_{\mathsf{PL}(r)} c$.

Proof. Let us abbreviate $\models_{\mathsf{PL}(r)}$ to simply $\models$. We have $\mathcal{V}_{\mathsf{PL}(r)} = \{\text{`}x\text{'}, \text{`}y\text{'}, \text{`}z\text{'}\}$. Let $T : \mathcal{V}_{\mathsf{PL}(r)} \to \mathbb{B}$ be an arbitrary truth assignment. By definition of $\models_{\mathsf{PL}}$, we need to prove that if $T \models h_1$ and $T \models h_2$, then $T \models_{\mathsf{PL}(r)} c$. Let us "mirror" the variables of the object language by introducing helper variables $x, y, r \in \mathbb{B}$ such that $x = T(\text{‘}x\text{’})$, $y = T(\text{‘}y\text{’})$, and $z = T(\text{‘}z\text{’})$. This means that we have

$$\begin{aligned}(T \models h_1) &\leftrightarrow (T \models \text{‘}(x \vee (y \wedge \textsc{False}))\text{’})\\ &\leftrightarrow ((T \models \text{‘}x\text{’}) \vee (T \models \text{‘}(y \wedge \textsc{False})\text{’}))\\ &\leftrightarrow ((T \models \text{‘}x\text{’}) \vee ((T \models \text{‘}y\text{’}) \wedge (T \models \text{‘}\textsc{False}\text{’})))\\ &\leftrightarrow (T(\text{‘}x\text{’}) \vee (T(\text{‘}y\text{’}) \wedge \textsc{False}))\\ &\leftrightarrow (x \vee (y \wedge \textsc{False})).\end{aligned}$$

Thus, the relation $\models_{\mathsf{PL}}$ effectively translates the formula h_1 into a statement of set theory by replacing symbols ‘$\wedge$’ and ‘$\vee$’ with the operations $\wedge$ and $\vee$ and the symbol ‘False’ with the constant False. If we do the same for h_2 and c we find that $(T \models h_2) \leftrightarrow (\neg(z) \vee y)$ and $(T \models c) \leftrightarrow (x \wedge y)$. Since $(T \models h_1)$ and $(T \models h_2)$ are true by assumption, the equivalent statements $(x \wedge (y \vee \textsc{False}))$ and $(\neg(z) \vee y)$ must also be true. By Theorem 4, this implies that $(x \wedge y)$ is true, which is equivalent to $(T \models c)$, and is what we set out to prove. □

In the end, defining the formal semantics of PL as in Definition 29 is merely a long way of saying that the PL inherits its semantics directly from the language of set theory. The languages to be defined later on follow the same principle, and from now on, explicit definitions of semantics will be omitted.

5.4 Reducing the number of parentheses

We would like to reduce the number of parentheses in the propositional formula ‘$((p \wedge q) \wedge r)$’. The usual solution is quite simple: we do not change the definition of a propositional formula, but introduce a shorthand notation under which we can omit parentheses. In this case we could say that the notation ‘$p \wedge q \wedge r$’ is simply an abbreviation of ‘$((p \wedge q) \wedge r)$’.

This approach is slightly problematic for our current purposes. If we do so, then ‘$p \wedge q \wedge r$’ and ‘$(p \wedge q) \wedge r$’ are abbreviations of the same formula, which means that the mapping between displayed formulas and their internal representations is not one-to-one. For the usual purposes this would not matter, but in textbooks the author might occasionally choose to insert superfluous parentheses, and in those cases and the system should preserve them. For example, if the author is trying to state and prove the formula ‘$(p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$’, then some of its natural symmetry is lost if it is instead displayed as ‘$p \wedge q \wedge r \leftrightarrow p \wedge (q \wedge r)$’.

Therefore, for the purposes of textbooks wish to maintain that ‘$p \wedge q \wedge r$’ and ‘$(p \wedge q) \wedge r$’ are different—but of course, equivalent—formulas. We leave the definition of PL as is, and instead, define a new conditional statement language PLE as an extension of PL, where some parentheses can be left out.

Definition 30. The conditional statement language PLE is defined so that the range declarations and assignments are the same as in PL: $\mathcal{R}_{\mathsf{PLE}} = \mathcal{R}_{\mathsf{PL}}$ and $\mathcal{A}_{\mathsf{PLE}(r)} = \mathcal{A}_{\mathsf{PL}(r)}$ for all $r \in \mathcal{R}_{\mathsf{PLE}}$. The set of formulas $\mathcal{F}_{\mathsf{PLE}(r)}$ and the satisfiability relation $\models_{\mathsf{PLE}(r)}$ is defined so that the PLE is an extension of PL with the following additions:

1. Redundant parentheses may be omitted. This includes the operations $\land$ and $\lor$ when used associatively or inside the logical relations $\to$ and $\leftrightarrow$. Also, the outermost parentheses wrapping the whole formula can be omitted.
2. The logical symbols $\to$ and $\leftrightarrow$ can be chained as relations.

For example, suppose that $r = [\text{‘}x, y, z \in \mathbb{B}\text{’}]$ and define

$$\phi = \text{‘}z \leftrightarrow (x \lor y) \to x \land y \land z\text{’}.$$

We now have $\phi \in \mathcal{F}_{\mathsf{PLE}(r)}$. Moreover, if $T : \mathcal{V}_{\mathsf{PLE}(r)} \to \mathbb{B}$ is a truth assignment, $x = T(\text{‘}x\text{’})$, $y = T(\text{‘}y\text{’})$, and $z = T(\text{‘}z\text{’})$, then $T \models_{\mathsf{PLE}(r)} \phi$ if and only if $z \leftrightarrow (x \lor y) \to x \land y \land z$, or equivalently, $\big((z \leftrightarrow (x \lor y)) \land \big((x \lor y) \to ((x \land y) \land z)\big)\big)$.

The chaining of ‘$\to$’ and ‘$\leftrightarrow$’ as relations is the most nonstandard aspect of PLE; most commonly these symbols would not chainable at all. However, if the author of the textbook does not use this particular feature, then the reader does not need to be aware of it. From the reader's perspective, chaining of logical relations does not increase the complexity of the language unless the author chooses to use it.

5.5 Boolean satisfiability problem

Suppose that $(r, h, c) \in \mathcal{C}_{\mathsf{PL}}$, and let us investigate how we could find out if $h \models_{\mathsf{PL}(r)} c$. If the number of different variables is $k = |\mathcal{V}_{\mathsf{PL}(r)}|$, then there are 2^k ways to assign the value False or True to each variable. It is straightforward to write a computer program that enumerates every assignment, and checks if all assignments that satisfy hypotheses h also satisfy the conclusion c. The problem with this approach is that if the number of variables k is large, then the number of truth assignments 2^k can become so large that even the fastest computer cannot enumerate them all. There is clearly a need for better algorithms.

We can simplify problem a bit. If $n \in \mathbb{Z}^+$, then the statement

$$\forall h_1, \ldots, h_n, c \in \mathbb{B} : (h_1 \land \ldots \land h_n) \to c$$

is equivalent to

$$\lnot\big(\exists h_1, \ldots, h_n, c \in \mathbb{B} : h_1 \land \ldots \land h_n \land \lnot(c)\big).$$

Using a similar transformation we can, for a given $h \in \textsc{List}(\mathcal{F}_{\mathsf{PL}(r)})$ and $c \in \mathcal{F}_{\mathsf{PL}(r)}$, easily produce a formula that is unsatisfiable if and only $h \models_{\mathsf{PL}(r)} c$. For example, if $h_1, h_2, c \in \mathsf{PL}(r)$, then $[h_1, h_2] \models_{\mathsf{PL}} c$ if and only if the formula ‘$\boxed{h_1} \land \boxed{h_2} \land \lnot(\textcircled{c})$’ is unsatisfiable. Therefore, if we can find an efficient algorithm that can tell if a single formula is satisfiable or not, this algorithm can also be used to tell if a conclusion follows from a list of hypotheses. The problem of telling if a formula is satisfiable is known as the *Boolean satisfiability problem* or *SAT*, and the programs that are used for solving it are known as *SAT solvers* [4].

None of the current SAT solvers are guaranteed to work efficiently for all possible formulas. Boolean satisfiability is known to be *NP-complete*, and if $P \neq NP$ (a major unsolved problem in theoretical computer science), then there is no algorithm that can solve the problem efficiently in the worst case. However, this theoretical obstacle has not prevented practical progress. The propositional formulas that typically arise in applications do not necessarily represent the worst case, and modern SAT solvers are routinely able to solve problems with over a million variables.

5.6 Erdős discrepancy problem

SAT solvers can often be applied to problems that on the surface do not seem to involve Boolean variables at all. There are numerous practical examples from the industry, but the example application given here is a purely mathematical one. Let us consider finite sequences $(x_1, \ldots, x_N)$, where $x_1, \ldots, x_N \in \{-1, 1\}$. For example, let $N = 11$ and take the finite sequence

$$\begin{array}{ccccccccccc} 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & x_{11} \end{array} \tag{8}$$

For a parameter $d \in \mathbb{Z}^+$, we define the cumulative sum $s_d(k) = \sum_{i=1}^{k} x_{id}$ for all $k \in \mathbb{N}$ where $kd \leq N$. The *discrepancy* of a ± 1-sequence is defined as

$$\max_{d,k} |s_d(k)| \,.$$

The discrepancy of sequence (8) is 1. This can be seen by calculating $|s_d(k)|$ for all 29 values of $k, d \in \mathbb{Z}^+$ for which $kd \leq 11$, and seeing its maximum value is 1. This is visualized in Figure 3 for $d \in \{1, 2, 3\}$. The sequence (8) cannot be extended by $x_{12} \in \{-1, 1\}$ if we still want to keep the same discrepancy. If we choose $x_{12} = 1$, we get $|x_6 + x_{12}| = 2$, and if we choose $x_{12} = -1$, we get $|x_3 + x_6 + x_9 + x_{12}| = 2$. An exhaustive search will show that there are in fact no sequences of length 12 and

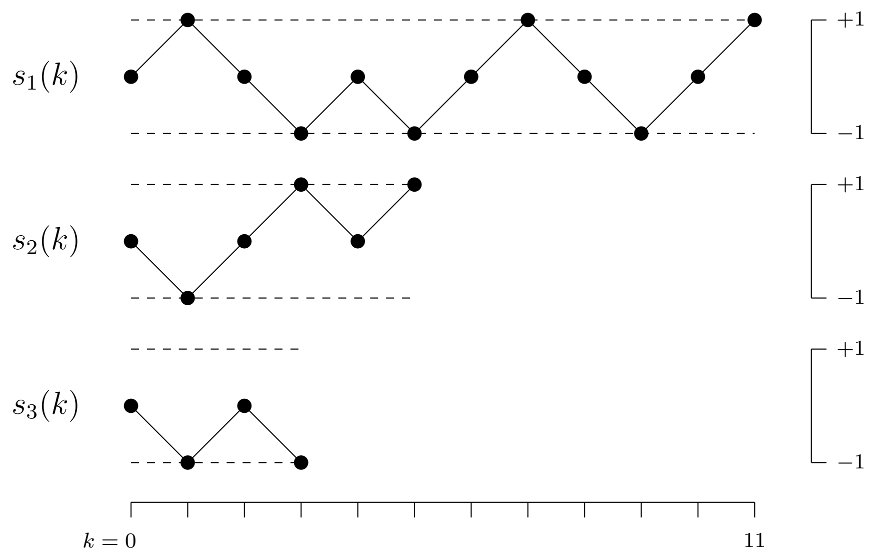


Figure 3: The discrepancy of the sequence (8) is 1, since $|s_d(k)| \leq 1$ for all $kd \leq 11$. This is illustrated here for $d \in \{1, 2, 3\}$.

discrepancy 1. Thus, our inability to extend the sequence is not just a consequence of an unfortunate choice of $(x_1, \ldots, x_{11})$; the length of ±1-sequences of discrepancy $C = 1$ is bounded, and the maximum length is 11. Erdős conjectured that the length of ±1-sequences of discrepancy C would be bounded for any $C \in \mathbb{Z}^+$. However, even the case $C = 2$ was a long-standing open problem. This was resolved in 2014 by Konev and Lisitsa. Using state-of-the-art SAT solvers, they showed that there exists a discrepancy 2 sequence of length 1160, but no sequence of length 1161. [32]

It remains to explain how SAT solvers can be applied to this problem. Assume that the length N and the maximum discrepancy C are fixed. It is easy to encode the variables $x_1, \ldots, x_N$ as Boolean variables: introduce variables $b_1, \ldots, b_N \in \mathbb{B}$ such that $b_i = \textsc{True}$ for $i \in \{1, \ldots, N\}$ if and only if $x_i = 1$. The harder part is to encode all constraints that these variables have to satisfy as propositional formulas.

To do so, we can introduce auxiliary variables in such a way that the discrepancy is at most C if and only if some set of formulas we is satisfiable. Let us introduce a variable $p_{d,k,c} \in \mathbb{B}$ for every $d, k \in \mathbb{Z}^+$ where $dk \leq N$ and $-C \leq c \leq C$. We want $p_{d,k,c}$ to be true if and only if $s_d(k) = c$ and $|s_d(i)| \leq C$ for all $i < k$. If $k \geq 1$, then $s_d(k) = s_d(k-1) + x_k$, meaning that

$$p_{d,k,c} = (b_k \wedge p_{d,k-1,c-1}) \vee (\neg(b_k) \wedge p_{d,k-1,c+1})$$

If $|c| > C$ then we let $p_{d,k,c} = \textsc{False}$. If $k = 0$, then we let $p_{d,k,c} = (c = 0)$. Finally, for every k and d we add the requirement $p_{d,k,-C} \vee \ldots \vee p_{d,k,C}$.

We have now found an alternative formulation of the problem where all variables are Boolean, and every constraint can be represented by propositional formula. This means that SAT solvers can now be used to attack the problem. It should be noted that the encoding given above was intentionally simplistic. It is not as efficient as the one given in [32], which uses a binary encoding for the values of $s_d(k)$.

Showing that none of the 2^{1161} possible ±1-sequences have a discrepancy of 2 is clearly beyond the reach of the simple brute force methods. More interestingly, it had also been beyond the reach of previous specialized programs, tailored specifically to the Erdős discrepancy problem. This a powerful testimony of the level of sophistication general purpose SAT solvers have achieved.

5.7 Compactness theorem

Definition 14 in Chapter 4 requires that the set of free variables $\mathcal{V}_L$ in a statement language is finite. This differs from the standard definition of propositional logic, which allows for an infinite number of variables [40]. Even though a single propositional formula can only contain a finite number of variables, an infinite set of formulas can, as a whole, contain an infinite number of variables. To explain why this can be useful, we give one application of the *compactness theorem of propositional logic.* That said, the framework of Chapter 4 is meant for discussing languages that are used for formalizing textbooks, and a textbook cannot contain an infinite number of variables. Therefore, this is the only section in this thesis where statement languages with an infinite number of variables are discussed.

Theorem 7 (*Compactness theorem of propositional logic*, Gödel, 1930)**.** *A set of propositional formulas for a countable set of variables is satisfiable if and only if every finite subset of it is satisfiable.*

Proof. See [40, p. 44]. □

As promised in Section 3.1, this theorem can be used to show that if there is an infinite counterexample to the four-color theorem, then there has to be a finite counterexample as well [15, p. 48]. This is a fairly representative example of the applications of the compactness theorem of propositional logic, so we prove it below. To use Theorem 7 we will first have to prove that the number of regions is only countably infinite. The compactness theorem for propositional logic can be extended to an uncountable number of variables, as proven by Maltsev in 1936, but this makes it impossible to encode variable names as members of the countable set of strings $\mathbb{S}$.

Theorem 8. *The number of regions in a simple planar map is countable.*

Proof. Let $R \subseteq \textsc{Set}(\mathbb{R}^2)$ be the set of regions of a map. Since regions are open subsets of $\mathbb{R}^2$, every region contains a point whose coordinates are rational numbers, and since the regions are pairwise disjoint, no point is contained in two different regions. Therefore, the number of regions cannot be larger than the number points with rational coordinates, and we have $|R| \leq |\mathbb{Q}^2| = \aleph_0$. □

Theorem 9. *If all simple planar maps with a finite number of regions can be four-colored, then all maps can be four-colored.*

Proof. Take some map with a countably infinite number of regions. Denote the set of all of its regions by $R \subseteq \textsc{Set}(\mathbb{R}^2)$. We are looking for a four-coloring, which can be represented by a function $c : R \to \{1, 2, 3, 4\}$, which satisfies the condition that for any pair of adjacent regions $(x, y) \in R^2$ we have $c(x) \neq c(y)$.

For every region r, define the auxiliary variables $r_1, r_2, r_3, r_4 \in \mathbb{B}$ in such a way that $r_i \leftrightarrow (c(r) = i)$. Exactly one of these four variables has the value True, or equivalently, the following has to hold:

$$\begin{aligned}&(r_1 \wedge \neg r_2 \wedge \neg r_3 \wedge \neg r_4) \vee (\neg r_1 \wedge r_2 \wedge \neg r_3 \wedge \neg r_4) \vee \\ &(\neg r_1 \wedge \neg r_2 \wedge r_3 \wedge \neg r_4) \vee (\neg r_1 \wedge \neg r_2 \wedge \neg r_3 \wedge r_4).\end{aligned} \tag{9}$$

Now, for every pair of adjacent regions $(x, y) \in R^2$ the requirement that $c(x) \neq c(y)$ is equivalent to

$$\neg((x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3) \vee (x_4 \wedge y_4)). \tag{10}$$

The four-colorability of the infinite map is equivalent to the satisfiability of a certain infinite set of propositional formulas. This set contains a formula like (9) for each region, and a formula like (10) for each pair of adjacent regions. Therefore, it is enough to prove that this infinite set of formulas is satisfiable.

Every finite subset of this set of formulas can only refer to a finite number of variables, and therefore only to a finite number of regions. By our assumption, the submap that contains only these regions is four-colorable, and this coloring gives one possible truth assignment that satisfies all formulas in the finite subset. Now the compactness theorem of propositional logic tells us that since every finite subset of the original set of formulas is satisfiable, the whole set is satisfiable, which is exactly what we had to prove. $\square$

6 First-order logic

First-order logic is an extension of propositional logic. Where propositional logic only supports Boolean variables, first-order logic supports three new kinds of variables: *set-valued variables*, whose values can be arbitrary sets; *function-valued variables*, whose values can be arbitrary functions; and *element variables*, which can range over one of the set variables. First-order logic supports universal and existential quantification for element variables, but not for set-valued or function-valued variables.

In this chapter we formalize *first-order logic* as the conditional statement language FL. More specifically, the variant of first-order logic that the conditional statement language FL is based on is called *many-sorted first-order logic with equality.* It supports the comparison of two element variables of the same *sort* or *type* using the equality relation $=$. It can be contrasted with *single-sorted first-order logic*, where every element variable ranges over the same set.

One motivation behind this chapter is the fact that first-order logic is closely related to abstract algebra. Many classes of algebraic structures, such as *rings*, *fields*, *groups*, *vector spaces*, and *Boolean algebras* are defined by giving a list of first-order *axioms* that the structure has to satisfy. Therefore, support for first-order logic is necessary for writing textbooks that talk about such things.

In Section 6.1 we discuss rings, which are an important example of a class of algebraic structures whose axioms are *equational laws.* Boolean algebras, introduced in Section 6.2, are another example of such a structure. In addition, Boolean algebras are involved in the Robbins conjecture, which serves as another example of successes of automated theorem proving. In Section 6.3 we look at examples of axioms that go beyond equational laws, and in Section 6.4 we define the syntax and semantics of the conditional statement language FL. In Section 6.5 we distinguish between two types of axioms for later purposes. Finally, in Section 6.6 we look at some of the theoretical results related to first-order logic. In particular, these results are needed to understand the limitations of what can be expressed in terms of first-order logic.

6.1 Algebraic structures and rings

In this section we define what is meant by an *algebraic structure*, and look at *rings* as a typical example of a class of algebraic structures.

Definition 31. An *algebraic structure* S consists of

1. an *underlying set* denoted by $|S|$,
2. zero or more *operations* $|S|^n \to |S|$, and
3. zero or more *named elements* that belong to $|S|$.

Formally, algebraic structures can be defined as a k-tuples whose first element is the underlying set, and the remaining elements are the operations and named elements.

Algebraic structures appear both in abstract algebra and in mathematical logic, but the terminology is slightly different. This chapters emphasizes the algebraic perspective, and therefore, uses the terminology that is more common in algebra. In more logically oriented texts the underlying set would more likely be called the *universe* or the *domain of discourse*. Similarly, the operations might be called *functions*, and the named elements might be called *constants*.

In abstract algebra it is common to use a relaxed notation, where an algebraic structure and its underlying set are both denoted by the same symbol S. For the current purposes it is better to keep them notationally separate. The use of $|S|$ for the underlying set is unrelated to other uses this notation, such as absolute value or cardinality. If we wish to combine the second and third items of Definition 31, it is possible to think of the named elements as operations $|S|^0 \to |S|$ that take zero arguments.

Several subclasses of algebraic structures can be defined by stating some *axioms* that the structure must satisfy. It is assumed that the named elements of a structure may be equal, unless the axioms say otherwise. All axioms considered before Section 6.3 are *equational laws*, which we define as statements of the form

$$\forall x_1, \ldots, x_k \in |S| : a(x_1, \ldots, x_k) = b(x_1, \ldots, x_k)$$

where the functions $a, b : |S|^k \to |S|$ are composed from the operations and named elements of the structure. Classes of algebraic structures that are characterized by equational laws are studied from slightly different perspectives in *equational logic* and *universal algebra*.

A ring R, as we will soon define, is an algebraic structure with three operations $+_R$, $\cdot_R$, and $-_R$, and two named elements 0_R and 1_R. Before proceeding with the definition we need to establish one notational convention. The elements 0_R and 1_R can be arbitrary members of $|R|$, and are not necessarily equal to the numbers 0 and 1. Similarly, $+_R$, $\cdot_R$, and $-_R$ can be arbitrary functions. However, if $x, y \in |R|$, it seems better to write $x + y$ than $x +_R y$, since this reduces notational clutter, and it is clear from the arguments that the symbol $+$ in the expression $x + y$ must refer to $+_R$. However, if we also dropped the subscripts from the named elements, this could sometimes lead to ambiguity. It would no longer be clear whether $1 + 1$ refers to the number $1 + 1 = 2$, or to ring element $1_R + 1_R$. We adopt a convention that is intended to make the notation more readable, but still keep it unambiguous.

Convention. *Named elements, such as 0_R and 1_R, are always written with subscripts. Operations are written without subscripts when they are used as a part of some expression, and with subscripts when they appear outside of an expression.*

Definition 32. A *ring* $R = (|R|, +_R, \cdot_R, -_R, 0_R, 1_R)$ is an algebraic structure with two binary operations $+_R, \cdot_R : |R|^2 \to |R|$, one unary operation $-_R : |R| \to |R|$, and two named elements $0_R, 1_R \in |R|$. The operation $\cdot_R$ has a higher precedence than $+_R$. The following axioms must hold for all $a, b, c \in |R|$:

(R_1)	$(a + b) + c = a + (b + c)$	(R_2)	$a + b = b + a$
(R_3)	$0_R + a = a$	(R_4)	$-a + a = 0_R$
(R_5)	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	(R_6)	$1_R \cdot a = a$
(R_7)	$a \cdot 1_R = a$	(R_8)	$a \cdot (b + c) = a \cdot b + a \cdot c$
(R_9)	$(a + b) \cdot c = a \cdot c + b \cdot c$		

The axioms above are all equational laws. For brevity, the universal quantifier is not repeated in front of every axiom.

Definition 33. A ring R is *commutative* if

$$(\mathrm{R}_{10}) \quad a \cdot b = b \cdot a$$

for all $a, b \in |R|$.

Since the operations $+_R$ and $\cdot_R$ are associative, they can be chained without parentheses. For notational convenience, it is customary to define some additional operations for rings. Ring elements can be subtracted: for $a, b \in |R|$, we define $a - b$ as $a + (-b)$. They can also be multiplied by an integer or raised to a nonnegative integer power, as defined below.

Definition 34. Suppose that R is a ring, $a \in |R|$, and $n \in \mathbb{Z}$. Then

$$n \cdot a = \begin{cases} 0_R, & \text{if } n = 0; \\ a + (n-1) \cdot a, & \text{if } n > 0; \\ -((-n) \cdot a), & \text{if } n < 0. \end{cases}$$

Furthermore, we define n_R as an abbreviation of $n \cdot 1_R$. Since $0_R = 0 \cdot 1_R$ and $1_R = 1 \cdot 1_R$, this agrees with the notation that is used for the two named elements in a ring. Every ring R includes the elements $\{\ldots, -2_R, -1_R, 0_R, 1_R, 2_R, \ldots\}$, but these elements are not necessarily all distinct.

Definition 35. Suppose that R is a ring, $a \in |R|$, and $n \in \mathbb{N}$. Then

$$a^n = \begin{cases} 1_R, & \text{if } n = 0; \\ a \cdot a^{n-1}, & \text{if } n > 0. \end{cases}$$

One example of a ring is the ring of complex numbers $(\mathbb{C}, +, \cdot, -, 0, 1)$. It is a commutative ring, since it satisfies all axioms from (R_1) to (R_{10}). To show that some result holds in all rings, and not just on particular examples, we can derive it as a consequence of the axioms. To show that $0_R \cdot a = 0_R$ for all $a \in |R|$ in an arbitrary ring R, it is enough to note that

$$\begin{aligned} 0_R \cdot a &\overset{(\mathrm{R}_3)}{=} 0_R + 0_R \cdot a \overset{(\mathrm{R}_4)}{=} (-(0_R \cdot a) + 0_R \cdot a) + 0_R \cdot a \\ &\overset{(\mathrm{R}_1)}{=} -(0_R \cdot a) + (0_R \cdot a + 0_R \cdot a) \\ &\overset{(\mathrm{R}_9)}{=} -(0_R \cdot a) + (0_R + 0_R) \cdot a \\ &\overset{(\mathrm{R}_3)}{=} -(0_R \cdot a) + 0_R \cdot a \overset{(\mathrm{R}_4)}{=} 0_R. \end{aligned} \tag{11}$$

This proof is a simple *chain of equalities*, where every step is justified directly by some axiom. Sometimes simple mathematical statements require sophisticated proofs, but this is not the case in ring theory, as guaranteed by the *completeness theorem of equational logic*, which was proven by Birkhoff in 1935. Roughly speaking, the theorem says that if all axioms are equational laws, then all of their consequences can be proven by a simple chain of equalities such as above. In particular, since the axioms of a ring are equational laws, every equational law that holds in all rings can be proven in such a way. See [41] for a more formal statement and proof of the theorem.

Definition 36. Suppose that S and R are rings where $|S| \subseteq |R|$, $0_S = 0_R$, $1_S = 1_R$, and that the operations $+_R$, $\cdot_R$, $-_R$ are equal to the operations $+_S$, $\cdot_S$, $-_S$ when restricted to the set $|S|$. Then S is called a *subring* of R.

This definition is a special case of a more general notion of *substructures* that can be naturally defined for all classes of algebraic structures whose axioms are equational laws. In the special case of rings, the definition can be weakened. It is unnecessary to demand that the restriction of $-_R$ equals $-_S$ and that $0_S = 0_R$, since these follow automatically if the other conditions are met [7].

The rings of integers, rational numbers, and real numbers are subrings of the ring of complex numbers. As a more advanced example, we may take the ring of *algebraic reals*. A real number—or more generally, a complex number—is *algebraic* if it is a root of some nonzero polynomial with integer coefficients. For instance, the number $\sqrt[3]{2}$ is algebraic since it is one of the roots of the polynomial $t^3 - 2$. Since all algebraic reals are real numbers, the axioms from (R_1) through (R_{10}) automatically hold for them as well. The hard part is to show that the sum and product of two algebraic reals are also algebraic [27].

In addition to subrings, *rings of functions* provide another way of obtaining new rings from existing ones. Again, the idea can be naturally generalized for other classes algebraic structures as long as all of the axioms are equational laws.

Definition 37. Suppose that R is a ring and X is an arbitrary nonempty set. The *ring of functions from X to R* is a ring T with the underlying set $|T| = (X \to |R|)$. The elements $0_T, 1_T \in (X \to |R|)$ are defined as the constant functions $0_T(x) = 0_R$ and $1_T(x) = 1_R$. The operations $+_T, \cdot_T : |T|^2 \to |T|$ and $-_T : |T| \to |T|$ are defined pointwise so that for every $a, b \in |T|$ we have

$$\begin{aligned} a + b &= (x \mapsto a(x) + b(x)) \\ a \cdot b &= (x \mapsto a(x) \cdot b(x)) \\ -a &= (x \mapsto -a(x)). \end{aligned}$$

Theorem 10. *If R is a ring, then the ring of functions T from X to R is in fact a ring.*

Proof. We must show that every ring axiom holds. Let us look at the axiom (R_9) as an example, and show that $(a + b) \cdot c = a \cdot c + b \cdot c$ for all $a, b, c \in |T|$. Using repeatedly the pointwise definition of the operations on T we see that

$$(a + b) \cdot c = (x \mapsto (a(x) + b(x)) \cdot c(x)).$$

The axiom (R_9) for the ring R tells us that

$$(x \mapsto (a(x) + b(x)) \cdot c(x)) = (x \mapsto a(x) \cdot c(x) + b(x) \cdot c(x)).$$

Finally, if we use the pointwise definition of the operations on T in reverse, we get

$$(x \mapsto a(x) \cdot c(x) + b(x) \cdot c(x)) = a \cdot c + b \cdot c.$$

Since the other axioms are also equational laws, they can be proved in essentially the same way. Therefore, T is a ring. $\square$

Finally, we define the concepts of *homomorphism* and *isomorphism*.

Definition 38. Suppose R and S are rings. A function $f : |R| \to |S|$ is called a *ring homomorphism* if for all $a, b \in |R|$ we have $f(a+b) = f(a)+f(b)$, $f(a\cdot b) = f(a)\cdot f(b)$, $f(-a) = -f(a)$, $f(0_R) = 0_S$, and $f(1_R) = 1_S$.

The definition was written in the most straightforward way possible, but can be weakened. It is only necessary to verify that $f(a + b) = f(a) + f(b)$, $f(a \cdot b) = f(a) \cdot f(b)$, and $f(1_R) = 1_S$, since the facts that $f(-a) = -f(a)$ and $f(0_R) = 0_S$ automatically follow [7].

Definition 39. Two rings R and S are *isomorphic*, denoted $R \cong S$, if there exists a bijective homomorphism from R to S.

In the case of isomorphisms it is not necessary to verify the property $f(1_R) = 1_S$ of homomorphisms. If f is bijective and $f(a \cdot b) = f(a) \cdot f(b)$ for all $a, b \in |R|$, then

$$f(1_R) = f(1_R) \cdot 1_S = f(1_R) \cdot f(f^{-1}(1_S)) = f(1_R \cdot f^{-1}(1_S)) = f(f^{-1}(1_S)) = 1_S.$$

6.2 Boolean algebras and the Robbins conjecture

In this section we look at *Boolean algebras*. Just like rings, Boolean algebras are a class of algebraic structures that can be defined by using only equational laws as axioms. There are several alternative notations for the operations of a Boolean algebra. The notation used here is the same as in [36].

Definition 40. A *Boolean algebra* $B = (|B|, \cup_B, \cap_B, \overline{\square}_B, 0_B, 1_B)$ is an algebraic structure that includes two binary operations $\cup_B$ and $\cap_B$, one unary operation $\overline{\square}_B$, and two elements 0_B and 1_B. The following axioms must hold for all $x, y, z \in |B|$:

(B$_1$)	$(x \cup y) \cup z = x \cup (y \cup z)$	(B$_2$)	$(x \cap y) \cap z = x \cap (y \cap z)$
(B$_3$)	$x \cup y = y \cup x$	(B$_4$)	$x \cap y = y \cap x$
(B$_5$)	$x \cup (x \cap y) = x$	(B$_6$)	$x \cap (x \cup y) = x$
(B$_7$)	$x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$	(B$_8$)	$x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$
(B$_9$)	$x \cup \overline{x} = 1_B$	(B$_{10}$)	$x \cap \overline{x} = 0_B$

The use notation $+$ and $\cdot$ within the context rings suggested a connection with ordinary addition and multiplication. Similarly, the use of notation $\cup$ and $\cap$ here suggests a connection with the set-theoretic union and intersection. This in fact so. If X is a nonempty set and $\overline{S}$ is the set complement $X \setminus S$, then $(\textsc{Set}(X), \cup, \cap, \overline{\square}, \emptyset, X)$ is a Boolean algebra. We can generate more Boolean algebras if we replace $\textsc{Set}(X)$

by some nonempty subset of $\textsc{Set}(X)$ that is closed under finite unions, intersections, and the complement.

It is straightforward to show that $x \cap y = \overline{\overline{x} \cup \overline{y}}$ for all $x, y \in |B|$. Therefore, if we replace all instances of $x \cap y$ by $\overline{\overline{x} \cup \overline{y}}$ in the axioms, we can axiomatize Boolean algebras without using the operation $\cap$ at all. We can also get rid of the named elements 0_B and 1_B if we replace axioms (B_9) and (B_{10}) by

$$(B_9^*) \quad x \cup \overline{x} = y \cup \overline{y} \qquad\qquad (B_{10}^*) \quad x \cap \overline{x} = y \cap \overline{y},$$

and take 1_B to be a notational abbreviation for the value of $x \cup \overline{x}$, which does not depend on x, and 0_B as an abbreviation for the value of $x \cap \overline{x}$, which, again, does not depend on x. This gives an axiomatization of Boolean algebras that has only two operations $\cup$ and $\overline{\Box}$, and no named elements.

At this point it is natural to ask if there is a simpler axiomatization than the 10 axioms given above. In 1930s Herbert Robbins asked if the three axioms

$$\begin{aligned}
&(H_1) \quad x \cup (y \cup z) = (x \cup y) \cup z \\
&(H_2) \quad x \cup y = y \cup x \\
&(H_3) \quad \overline{\overline{x \cup y} \cup \overline{x \cup \overline{y}}} = x
\end{aligned}$$

would suffice. It is easy to show that these axioms follow from the axioms of a Boolean algebra, but the question if the axioms of a Boolean algebra followed from these three axioms remained an open problem until 1996. This question is known as the Robbins conjecture.

The axioms of a Boolean algebra and the three axioms proposed by Robbins are all equational laws. Therefore, if the Robbins conjecture is true, it follows that all of the axioms of a Boolean algebra can be proven using simple chains of equalities, where every step is justified by one of the axioms (H_1) through (H_3). In other words, long before the Robbins conjecture was proved it was known that if the conjecture is true, then it has a proof that is technically very straightforward.

Despite this, the problem remained open for decades. The Robbins conjecture was proven in 1996 by McCune using the automated theorem proved *EQP* and several days of computer time [38]. The proof, even when presented for a human reader [36], has a distinctly mechanical flavor.

6.3 First-order axioms

Up to this point, all of the axioms we have worked with have been equational laws, and all elements have been of the same type. This is not always enough. *Fields* require existential quantification, *ordered rings* require other relations than just equality, and *vector spaces* require multiple types of elements. For the most part, the definitions given in this section will not be needed later in this thesis, but they serve as motivating examples of situations where equational laws with a single underlying set are not expressive enough.

A full *first-order axiom*, as opposed to an equational law, can contain logical operations and relations ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$) and universal and existential quantifiers over the underlying set $|S|$.

Definition 41. A *nontrivial ring* is a ring where $0_R \neq 1_R$. Formally, this is an abbreviation of $\neg(0_R = 1_R)$.

Definition 42. A *field* F is a nontrivial commutative ring with the additional axiom

$$\forall x \in |F| : x \neq 0_F \rightarrow \exists y \in |F| : x \cdot y = 1_F.$$

In other words, a field is a nontrivial commutative ring where every nonzero element is *invertible.*

Another use of first-order axioms is to reduce the number of operations and named elements in a structure. For example, it is possible to define rings in such a way that they only have two binary operations ($+_R$ and $\cdot_R$) and no named elements [7]. This can be done if we replace the axioms (R_3), (R_4), (R_6), and (R_7) by the following variations:

$$\begin{array}{ll} (R_3^*) & \exists e \in |R| : \forall a \in |R| : e + a = a \\ (R_4^*) & \forall e, x, a \in |R| : e + x = x \rightarrow \exists b \in |R| : b + a = e \\ (R_6^*) & \exists u \in |R| : \forall a \in |R| : u \cdot a = a \\ (R_7^*) & \exists u \in |R| : \forall a \in |R| : a \cdot u = a. \end{array}$$

One benefit of doing so is that it simplifies notation, since we can talk about the ring $(|R|, +_R, \cdot_R)$ instead of the ring $(|R|, +_R, \cdot_R, -_R, 0_R, 1_R)$. In this work we stick to using $(|R|, +_R, \cdot_R, -_R, 0_R, 1_R)$, because there are several theoretical advantages in using equational laws whenever possible. Having access to the completeness theorem of equational logic is just one these advantages.

If F is a field and X is a nonempty set, then the ring of functions $X \rightarrow F$ is not necessarily a field. As a simple example, take the ring of functions $\mathbb{R} \rightarrow \mathbb{R}$. The function $x \mapsto x$ is not identically zero, so it should have an inverse $x \mapsto g(x)$ such that $x \cdot g(x) = 1$ for all $x \in \mathbb{R}$. But this is impossible for $x = 0$, which means that the ring of functions from $\mathbb{R}$ to $\mathbb{R}$ is not a field. The kind of reasoning that was used in the proof of Theorem 10 worked for equational laws, but it does not work for full first-order axioms. If we use full first-order axioms for defining rings, the we lose sight of the reason why a ring of functions is a ring, but a ring of fields is not usually a field. This is one the reasons why it is desirable to use equational laws as axioms whenever possible. Furthermore, the definitions of a subring (Definition 36) and a ring homomorphism (Definition 38) give special treatment to the element 1_R. If rings are defined in a way that does not list 1_R as a named element, it makes these definitions feel less natural.

Equality ($=$) is the only relation we have used so far, but sometimes it is also useful to equip algebraic structures with other relations, or more generally, predicates. We drop the word *algebraic*, and define a *structure* as the appropriate generalization that allows just that.

Definition 43. A *structure* S consists of

1. an *underlying set* denoted by $|S|$,
2. zero or more *operations* $|S|^n \rightarrow |S|$,

3. zero or more *named elements* that belong to $|S|$, and
4. zero or more *predicates* $|S|^n \to \mathbb{B}$.

The *ordered ring*, which we define below, is a simple example of such a structure.

Definition 44. An *ordered ring* R is a ring with an additional relation $\leq_R : |R|^2 \to \mathbb{B}$ where the following hold for all $a, b, c \in |R|$:

(O_1)	$a \leq b \wedge b \leq a \to a = b$	(O_2)	$a \leq b \wedge b \leq c \to a \leq c$
(O_3)	$a \leq b \vee b \leq a$	(O_4)	$a \leq b \to a + c \leq b + c$
(O_5)	$0_R \leq a \wedge 0_R \leq b \to 0_R \leq a \cdot b$		

Finally, it is sometimes useful to have several underlying sets that correspond to elements of different *sorts* or *types*. Perhaps the most common example is the vector space, where we distinguish between vectors and scalars.

Definition 45. A *vector space V over the field F* consists of a field F, and another underlying set $|V|$ with the named element $0_V \in |V|$. In addition to the field operations on F it has the operations $+_V : |V|^2 \to |V|$, $-_V : |V| \to |V|$, and $\cdot_{F,V} : |F| \times |V| \to |V|$ such that the following axioms hold for all $a, b \in |F|$ and $u, v, w \in |V|$:

(V_1)	$u + (v + w) = (u + v) + w$	(V_2)	$u + v = v + u$
(V_3)	$v + 0_V = v$	(V_4)	$-v + v = 0_V$
(V_5)	$a \cdot (b \cdot v) = (a \cdot b) \cdot v$	(V_6)	$1_F \cdot v = v$
(V_7)	$a \cdot (u + v) = a \cdot u + a \cdot v$	(V_8)	$(a + b) \cdot v = a \cdot v + b \cdot v$

Numerous other examples of classes of structures with first-order axioms can be given.

6.4 Syntax and semantics

In the following, the statement $X_1, \ldots, X_n$: SET means that $X_1, \ldots, X_n$ are arbitrary sets. As an example of the syntax of FL, we can take the following nonsensical (and false!) conditional statement:

FOR:	'$p, q, r \in \mathbb{B}$'
	'S, T : SET'
	'$f : S \times T \to S$'
	'$P : T \to \mathbb{B}$'
	'$x, y \in S$'
IF:	'$\forall a \in S : \exists b \in T : P(f(a, b))$'
THEN:	'$x = y$'

It contains five range declarations, one hypothesis, and the conclusion. The conditional statement language FL inherits its semantics directly from set theory, so we only need to describe its syntax. For notational simplicity, the items 1 through 5 below are written for two variables, but the definitions apply—with the obvious generalizations—to any number of variables. Similarly, the items 3 through 6 below are written for binary functions only, but they apply to any number of inputs. An *undeclared variable* is a member of the set $\mathbb{V}$ that has not been declared by any previous range declaration. The word "afterward", when used in the context of a range declaration, refers to all range declarations that appear later, all hypotheses, and the conclusion.

1. If $v_1, v_2 \in \mathbb{V}$ are undeclared variables, then '$\boxed{v_1}, \boxed{v_2} \in \mathbb{B}$' is a range declaration. Afterward, v_1 and v_2 are *formulas.*

2. If $S_1, S_2 \in \mathbb{V}$ are undeclared variables, then '$\boxed{S_1}, \boxed{S_2} : \textsc{Set}$' is a range declaration. Afterward, S_1 and S_2 are *set-valued terms.*

3. If $v_1, v_2 \in \mathbb{V}$ are undeclared variables and S is a set-valued term, then '$\boxed{v_1}, \boxed{v_2} \in \boxed{S}$' is a range declaration. Afterward, v_1 and v_2 are *S-valued terms.*

4. Suppose that $f, g \in \mathbb{V}$ are undeclared variables and $S_1, S_2, T \in \mathbb{S}$ are set-valued terms. Then '$\boxed{f}, \boxed{g} : \boxed{S_1} \times \boxed{S_2} \to \boxed{T}$' is a range declaration. Afterward, f and g is a *function-valued terms of the type $S_1 \times S_2 \to T$.*

5. Suppose that $f, g \in \mathbb{V}$ are undeclared variables and $S_1, S_2 \in \mathbb{S}$ are set-valued terms. Then '$\boxed{f} : \boxed{S_1} \times \boxed{S_2} \to \mathbb{B}$' is a range declaration. Afterward, f and g are *predicate-valued terms of the type $S_1 \times S_2 \to \mathbb{B}$.*

6. Suppose that a is an A-valued term and b is a B-valued term. If f is a function-valued term of type $A \times B \to C$, then '$\boxed{f}(\boxed{a}, \boxed{b})$' is a *$C$-valued term.* If f is a predicate-valued term of type $A \times B \to \mathbb{B}$, then '$\boxed{f}(\boxed{a}, \boxed{b})$' is a *formula.*

7. If a and b are both A-valued terms, then '$\boxed{a} = \boxed{b}$' is a formula.

8. If p and q are formulas, then so are '$(\boxed{p} \vee \boxed{q})$', '$(\boxed{p} \wedge \boxed{q})$', '$\neg(\boxed{p})$', '$(\boxed{p} \to \boxed{q})$', and '$(\boxed{p} \leftrightarrow \boxed{q})$'. Redundant parentheses, as defined in Section 5.4, can be omitted.

9. Let v be an undeclared variable and S be a set-valued term. Assume that ϕ is a formula under the additional assumption that v in ϕ is a S-valued term. Then '$\forall \boxed{v} \in \boxed{S} : \boxed{\phi}$' and '$\exists \boxed{v} \in \boxed{S} : \boxed{\phi}$' are formulas.

As a consequence of defining PL as sublanguage of the language of set theory, the definition given above has some unusual features when compared to a more standard definition, such as the one given in [40]. First of all, the standard definition describes string representations for terms and formulas only, but the definition above also gives string representations for range declarations.

The standard definition of first-order logic is usually given in terms of a single underlying set, but the definition above uses multiple sets. This distinction corresponds to a standard one between *one-sorted* and *many-sorted* variations of first-order logic. In single-sorted syntax, the formula '$\exists x, y \in S : x \neq y$' would be expressed as '$\exists x, y : x \neq y$' instead. Single-sorted syntax is usually more convenient for theoretical study, but it is also less explicit. In the given example, the truth value of the formula depends solely on the size of the set S. If $|S| > 1$, then the formula is true; otherwise it false. In many-sorted syntax the dependency on the set S is visible, and in single-sorted syntax it is hidden.

Finally, in first order logic it is customary to assume that the underlying set is nonempty. This means, for example, that the formula '$\exists x \in S : x = x$' is considered to be valid, although it is not true if $S = \emptyset$. However, FL inherits its semantics directly from set theory, and the notation $S : \textsc{Set}$ does not imply that the set S is nonempty. Therefore, the conditional statement language FL does not guarantee that sets are nonempty, unless explicitly assumed otherwise. In fact, the formula '$\exists x \in S : x = x$' is one possible way to state the assumption that S is nonempty.

6.5 Structural and foundational axioms

Let us make a distinction between *structural axioms* and *foundational axioms*. The axiom

$$\forall a, b \in |R| : a + b = b + a \tag{12}$$

is an example of a *structural axiom*. It is not true or false by itself, because its truth value depends on R. Instead, it is simply one the requirements that R must satisfy before we are entitled to call R a ring. In contrast,

$$\forall a, b \in \mathbb{N} : a + b = b + a \tag{13}$$

is a true statement about a specific set: the set of natural numbers $\mathbb{N}$. Whether (13) can be proved depends on the definitions that have been chosen. We may choose to take the addition of natural numbers as a primitive concept that is not formally defined in terms of anything else. In this case a formal proof of (13) is not possible, but we can at least can give a combinatoric or a geometric explanation of why (13) has to be true, and choose to accept (13) as a *foundational axiom*. Alternatively, we may define addition formally in terms of the *successor function* $S : \mathbb{N} \to \mathbb{N}$ so that for any $x, y \in \mathbb{N}$ we have $x + 0 = 0$ and $x + S(y) = S(x + y)$. In this case we do not have to accept (13) without a formal proof: it can be proved by induction on y.

It is not possible to get rid of all primitive concepts in a language. We can define addition in terms of the successor function, the successor function in terms of higher-order logic, and higher-order logic in terms of set theory. But sooner or later this has to stop, and if we want the textbook to be understandable it may be better to stop sooner rather than later. Otherwise we end up defining an intuitively accessible concept, such as the addition of natural numbers, in terms of something that is far more difficult to understand.

With this in mind, we can think of foundational axioms as true statements that we accept without a formal proof, because we they involve concepts that we have chosen not to define formally.

6.6 Theoretical results

This section gives a brief tour of some standard theoretical results related to first-order logic. For concreteness, many results will only be stated in the special case of rings. Where not stated otherwise, the general results and their proofs can be found in [37].

A formula without free variables is called a *sentence*, and in this section the sentences are written using single-sorted syntax, meaning that the underlying set is implicit. The usual notational conveniences are allowed: the operations are written in infix, associative operations do not require parentheses, and multiplication by an integer and the exponentiation to a nonnegative integer power are allowed. Therefore, '$\exists x : x^2 = 2_R$' is a sentence in the language of rings. This particular sentence is true in the ring $(\mathbb{R}, +, \cdot, -, 0, 1)$, but false in the ring $(\mathbb{Q}, +, \cdot, -, 0, 1)$.

A *theory* is an arbitrary set of such sentences. A structure that satisfies a theory T is called a *model*. If T is a theory, then $\mathrm{Con}(T)$ is the set of *consequences* of T: the set of sentences that hold in all models of T. Some theories, such as the theory of rings, consist of a finite set of axioms. In a more theoretical setting a theory can be an infinite set and its structure can be arbitrarily complex. As a case in point, if R is some particular ring, then let the *full theory of* R, denoted by $\mathrm{Th}(R)$, be the set of all sentences that are true in R. For example, $\mathrm{Th}(\mathbb{Z}, +, \cdot, -, 0, 1)$ is the full theory of the ring of integers. This theory includes the sentence '$\forall x, y : x \cdot y = y \cdot x$', but not the sentence '$\exists x : 2 \cdot x = 1_R$'.

Definition 46. A theory T is *decidable* if $\mathrm{Con}(T)$ is a computable set.

Theorem 11. *If R is a ring, then $\mathrm{Th}(R)$ is decidable if and only if there exists an algorithm that can, for a given sentence, decide if it is true in R.*

Proof. It is enough to show that the set $\mathrm{Con}(\mathrm{Th}(R))$ contains exactly those sentences that are true in R. A sentence that is true in R belongs to $\mathrm{Th}(R)$, and by definition, is true in every model of $\mathrm{Th}(R)$. Therefore, it belongs to $\mathrm{Con}(\mathrm{Th}(R))$. Conversely, a sentence that is false is R cannot belong to $\mathrm{Con}(\mathrm{Th}(R))$, because it would have to be true in all models of $\mathrm{Th}(R)$, and R itself is a model of $\mathrm{Th}(R)$. $\square$

Gödel proved in 1931 that $\mathrm{Th}(\mathbb{Z}, +, \cdot, -, 0, 1)$ is undecidable [13], and Robinson proved in 1949 that $\mathrm{Th}(\mathbb{Q}, +, \cdot, -, 0, 1)$ is also undecidable [45]. Tarski proved in 1951 that $\mathrm{Th}(\mathbb{R}, +, \cdot, -, 0, 1)$ is decidable [51]. See [42] for a more comprehensive list of results of this type. It may be surprising that real numbers are in this sense easier than integers or rationals. For a simple plausibility argument we can note that if we can find at least one positive and one negative value for a multivariate real polynomial, then it has to have a zero as well. A similar argument does not hold for integers or rationals.

Gödel's completeness theorem, stated in the terminology of this thesis, says that there exists a proof language P that completely captures the notion of truth for conditional statements in FL: a conditional statement is true in FL if and only if it is provable in P. This is useful as a pure existence result even if we do not say anything about the proof language P. According to Theorem 3 it implies that the set $\mathcal{T}_{\mathsf{FL}}$ of true conditional statements in FL is computably enumerable.

It is reasonably straightforward to show that if two rings isomorphic, then they satisfy the same sentences. However, the converse does not hold. The ring of reals and the ring of algebraic reals satisfy the same sentences, but they cannot be isomorphic, since the set of algebraic reals is countable and the set of real numbers is uncountable. Generally, we say that two rings R_1 and R_2 are *elementarily equivalent* if they satisfy exactly the same sentences. The situation concerning the real numbers and algebraic real numbers is an example of a more general result. Every infinite ring R has a countable subring R that is elementarily equivalent to R. Even more generally, if a first-order theory has an infinite model, then it has a model of all infinite cardinalities. In other words, first-order axioms cannot control the cardinality of infinite models in any way.

In practice, this means that while first-order axioms are useful for defining classes of structures, such as rings, they are not as well suited for defining specific examples of such structures. For example, suppose that we would use first-order axioms to describe the ring of real numbers. Since the ring of real numbers and the ring of algebraic reals are elementarily equivalent, any first-order axiomatization is unable to distinguish between these two.

This is not a problem as long as we continue to use first-order logic, because of first-order logic by definition is unable to distinguish between elementarily equivalent models. However, in the spirit of extensibility we should require that our definitions never become obsolete when the conditional statement language is extended. The conditional language FL only supports two quantifiers ("for all" and "there exists"), but an extension of FL could add support for news quantifiers, such as "for countably many", and using this quantifier it would be possible to distinguish between the ring of reals and the ring of algebraic reals.

The problem becomes even worse if we try to axiomatize the ring of integers. In other words, we want to choose a theory $T \subseteq \mathrm{Th}(\mathbb{Z}, +, \cdot, -, 0, 1)$ that is a subset of the full theory of integers, but in such a way that $\mathrm{Con}(T) = \mathrm{Th}(\mathbb{Z}, +, \cdot, -, 0, 1)$. Preferably, we would want T to be a finite set. If this is not possible, then we would want T to be a computable set, or by the very least, computably enumerable. Without such an assumption we cannot use the theory T in a proof checker. However, it follows from Gödel's completeness theorem that if T is computably enumerable, then $\mathrm{Con}(T)$ is also computably enumerable. Since the full theory $\mathrm{Th}(\mathbb{Z}, +, \cdot, -, 0, 1)$ is not computably enumerable, a set T that satisfies our requirements does not exist.

The semantics of first-order logic is defined by quantifying over all possible sets, all possible operations, and all possible predicates. The number of all sets is much larger $2^{\aleph_0}$, which makes this potentially problematic in the light of assumption (4) in Section 2.9. Therefore, let us return to the fact that if a first-order theory has an infinite model, then it also has a countable model. If the model is countably infinite, then the cardinality of all operations and predicates is just $2^{\aleph_0}$. Since there is no need to quantify over sets that are larger than $2^{\aleph_0}$, we can conclude that according to assumption (4) the semantics of first-order logic is unambiguously defined.

7 A second-order language

In the previous chapter we saw two limitations of first-order logic. First, arbitrary sets of axioms cannot specify infinite structures up to isomorphism; they can only be specified up to elementary equivalence. Second, even specifying a structure up to elementary equivalence can require a set of axioms that is not computably enumerable. This happens, for example, for the ring of integers and the ring of rational numbers.

These problems can be fixed by using *higher-order logic*, which is discussed in Section 7.1. Higher-order logic is much more expressive than first-order logic, but brings with it two major problems. First of all, there does not exist a proof language that fully captures the notion of truth for conditional statements in higher-order logic. Even worse, the semantics of higher-order logic is rather vague: it is unclear what does it even *mean* for a conditional statement in higher-order logic to be true. We argue that the first problem is unavoidable, but the second can be avoided by choosing another way to extend first-order logic. This leads to the definition of the conditional statement language SD in Section 7.2. This language is closely related to *second-order arithmetic*, but the underlying set in second-order arithmetic is the set of integers, and the underlying set in SD is the set $\mathbb{D}$. The language SD contains a number of built-in functions. This is complemented by the ability to define new functions by using a simple programming language, which we describe in Section 7.3.

Using the language SD for expressing theorem statements is a major difference between this work and most of the currently existing systems. There exists a fairly large number of proof systems, and the differences between them can be subtle, so it is beyond the scope of this thesis to give a detailed account of the other systems. However, Wiedijk gives a comparison of 17 commonly used proof assistants, and 15 of them—the exceptions 2 being ACL2 and Minlog—are listed as being based on either higher-order logic of ZFC set theory [52]. The two exceptions are not very close to the language SD either: ACL2 specializes in large scale software verification, and it uses a quantifier-free logic [30]. Minlog is based on minimal rather than classical logic [48].

7.1 Second-order and higher-order logics

First-order logic supports several different types of variables: elements, functions, predicates, sets, and Booleans. However, it only supports quantification over element variables. This is extended in higher order logics. If S is a set, then *second-order logic* allows quantification over the power set $\textsc{Set}(S)$. Going further, *third-order logic* allows quantification over $\textsc{Set}(\textsc{Set}(S))$. However, even though second-order logic is more expressive than first-order logic, third-order logic is not more expressive than second-order logic. For any sentence in third-order or even higher-order logics it is possible to construct a second-order sentence that is valid if and only if the original sentence is valid [10]. Because of this, it is not particularly important to keep track of different orders, and we shall simply talk about higher-order logic.

One advantage of higher-order logic is that it makes it possible to define infinite structures up to isomorphism. As an example, we shall look at how to define the ring of integers up to isomorphism by adding a second-order induction axiom.

Theorem 12. *An ordered ring R is isomorphic to the ring of integers if it satisfies the following second-order axiom:*

$$\forall S \in \textsc{Set}(|R|) : (0_R \in S \wedge \forall s \in S : s+1 \in S) \to (\forall x \in |R| : x \geq 0_R \to x \in S). \quad (14)$$

Proof. Define the function $f : \mathbb{Z} \to |R|$ so that $f(n) = n \cdot 1_R$. To verify that this is a ring homomorphism we first note that $f(1) = 1_R$. Also, for all $a, b \in \mathbb{Z}$ we have $f(a+b) = (a+b) \cdot 1_R = a \cdot 1_R + b \cdot 1_R$ and $f(a \cdot b) = (a \cdot b) \cdot 1_R = (a \cdot 1_R) \cdot (b \cdot 1_R)$. The function f is an injection. To see why, assume that $a, b \in \mathbb{Z}$ and $a \neq b$. Without loss of generality, we can assume that $a < b$. Since R is an ordered ring, the fact that $a < b$ implies that $a \cdot 1_R < b \cdot 1_R$, so the elements $f(a) = a \cdot 1_R$ and $f(b) = b \cdot 1_R$ cannot be equal.

Finally, we show that $f(\mathbb{Z})$ is in fact equal to $|R|$. By contradiction, suppose that there is such an element k that $k \in |R|$ but $k \notin f(\mathbb{Z})$. The set $f(\mathbb{Z})$ is closed under negation, so we also have $-k \in |R|$, but $k \notin f(\mathbb{Z})$. Therefore, without loss of generality we can assume that $k \geq 0_R$. Let $N = \{n \in f(\mathbb{Z}) \mid n \geq 0_R\}$ be the set of nonnegative elements in the image of f. If we set $S = N$ and $x = k$ in the axiom (14), then we see that $k \in N$, which implies that $k \in f(\mathbb{Z})$. We have found a contradiction. □

Being able to describe infinite structures up to isomorphism is a clear advantage of higher-order logic, but it has also two major disadvantages. The first disadvantage is that unlike first-order logic, higher-order logic does not admit a complete proof theory.

Theorem 13. *There does not exist a complete proof language for the conditional statements of higher-order logic.*

Proof. In Theorem 12 it was shown that the ring of integers can be defined up to isomorphism, and therefore, up to elementary equivalence, using a finite number of higher-order axioms. If a complete proof language for higher-order logic were to exist, then this proof language should be able to prove every first-order sentence that is true in the ring of integers. According to Theorem 3 this would imply that the set of true sentences in the ring $(\mathbb{Z}, +, \cdot, -, 0, 1)$ is computably enumerable. However, as noted in Section 6.6, this is not so, and therefore, a complete proof language for higher-order logic cannot exist. □

The second disadvantage of higher-order logic is that it is, in a sense, so expressive that even its semantics is no longer clear. It is possible to define the semantics of higher-order logic by using set theory, but this makes it subject to the ambiguities that concern quantification over very large sets. As discussed in Section 2.9, the assumption that is made in this thesis is that quantification is unproblematic as long as the sets are not larger than the set of real numbers. There exists a sentence in higher-order logic that is valid if and only if the continuum hypothesis is true

[10]. If we regard the continuum hypothesis as something that does not necessarily have a definite truth value, then we must adopt the same attitude with the notion of validity in higher-order logic. In fact, Quine famously criticized higher-order logic as being “set theory in sheep’s clothing” [44].

We have now listed two disadvantages of higher-order logic: it does not admit a complete proof language, and its semantics is ambiguous. By examining the proof of Theorem 12 we can see that the first disadvantage is practically unavoidable if we want a language that can talk about integers. Since the full theory of the ring of integers is undecidable, there cannot exist a language that is simultaneously simple enough to have a complete proof language, but also expressive enough to be able to describe the ring of integers up to isomorphism, or even just elementary equivalence.

The second disadvantage, however, can be avoided. It is possible to have a language “in the middle ground” that only has the first disadvantage, but not the second. Perhaps the simplest answer is to choose a language that is not, properly speaking, a logic. When a language is called a *logic*, no assumptions are usually made about the domain of discourse. However, we can just as well choose to study a language whose domain of discourse is, by definition, some particular set. As long as we are working with sublanguages of set theory, we only need to choose some *built-in* sets, and functions, and constants to define such a language.

Definition 47. The conditional statement language FA (*first-order arithmetic*) has the built-in set $\mathbb{Z}$, the constants $0, 1 \in \mathbb{Z}$, and the functions $+, \cdot : \mathbb{Z}^2 \to \mathbb{Z}$ and $- : \mathbb{Z} \to \mathbb{Z}$.

First-order arithmetic is often defined for natural numbers instead of integers. This work uses integers because integers are a ring, but the difference is fairly minor. The above definition means that, for example

$$([\text{‘}a \in \mathbb{Z}\text{’}], [\,], \text{‘}\exists b \in \mathbb{Z} : a = b + b\text{’}) \tag{15}$$

is a conditional statement of FA. In this language the symbol ‘$\mathbb{Z}$’, by definition, refers to the set of integers, and the symbol ‘$+$’ refers to the addition of integers. Therefore, if a is odd, then such an integer b does not exist, which means that (15) is a false conditional statement.

According to assumption (4) we can safely quantify over sets as large $2^{\aleph_0}$. This means that FA is safe, since it only quantifies over the countably infinite set $\mathbb{Z}$. In fact, we can extend FA by adding support for the set $\textsc{Set}(\mathbb{Z})$ and the relation $\in : \mathbb{Z} \times \textsc{Set}(\mathbb{Z}) \to \mathbb{B}$. This results in a language that can be called *second-order arithmetic*.

7.2 Syntax and semantics of SD

The theoretical benefits of second-order arithmetic do not require that the underlying set would have to be $\mathbb{Z}$. The same properties hold as long as it is a countably infinite set. In the language SD, which we describe in this section, that set is a subset of $\mathbb{D}$. In the name SD the letter S refers to the fact that it is a *second-order* language, and the letter D refers to the set $\mathbb{D}$ (“data”).

We distinguish between first-order sets, which have cardinality $\aleph_0$, and second-order sets, which have cardinality $2^{\aleph_0}$. The built-in sets are the following:

1. $\mathbb{Z}$ is a first-order set.
2. If X is a first-order set, then $\textsc{List}(X)$ is a first-order set.
3. If X is a first-order set, then $\textsc{Set}(X)$ is a second-order set.
4. If $X_1, \ldots, X_n$ and Y are first-order sets, then $X_1, \ldots, X_n \to Y$ is a second-order set.

It is possible to quantify over both first-order and second-order sets. The language has a constant symbol for every integer $(\ldots, -2, -1, 0, 1, 2, \ldots)$. The built-in functions are the following:

1. It is possible to write lists of elements of that consists of elements of the same type. For example, we can write $[[2,5],[\,],[3]]$ where all elements are of the type $\textsc{List}(\mathbb{Z})$. Technically, if τ is a type and $n \in \mathbb{N}$, then there exists a function $[\,] : \tau^n \to \textsc{List}(\tau)$.
2. It is possible to construct finite sets by enumerating their elements. For example, we can write $\{[2,5],[\,],[3]\}$ where all elements are of the type $\textsc{List}(\mathbb{Z})$. Technically, if τ is a type and $n \in \mathbb{N}$, then there exists a function $\{\,\} : \tau^n \to \textsc{Set}(\tau)$.
3. The following built-in functions exist for integers: $+, \cdot : \mathbb{Z}^2 \to \mathbb{Z}$, $- : \mathbb{Z} \to \mathbb{Z}$, $\mathit{range} : \mathbb{Z} \to \textsc{List}(\mathbb{Z})$, $<, >, \leq, \geq : \mathbb{Z}^2 \to \mathbb{B}$. Here $\mathit{range} : \mathbb{Z} \to \textsc{List}(\mathbb{Z})$ is defined so that $\mathit{range}(a,b) = [a, \ldots, b]$ when $a \leq b$, and $\mathit{range}(a,b) = [\,]$ when $a > b$.
4. For every type τ there exists the list concatenation function $\diamond : \textsc{List}(\tau)^2 \to \textsc{List}(\tau)$. For example, $[2,5] \diamond [6,7,1] = [2,5,6,7,1]$.
5. For every type τ there exists the membership relation $\in : \tau \times \textsc{Set}(\tau) \to \mathbb{B}$, and the union and intersection operators $\cup, \cap : \textsc{Set}(\tau)^2 \to \textsc{Set}(\tau)$.
6. It is possible to define new functions, as defined in Section 7.3.

7.3 Defining new functions

The function definition language of `SD` is a simple programming language, which means that all definable functions are automatically computable functions. This language can evaluate expressions, assign values to variables, and it has the standard `if/else` statement for conditional evaluation and the `for/in` statement for looping over a list. The source code in Figure 4 demonstrates this language by defining the absolute value function, the "is divisible by" relation, and a primality test.

```
abs : Int -> Int
abs(x):
    if x >= 0:
        return x
    else:
        return -x

divides : Int * Int -> Bool
divides(a, b):
    if b == 0:
        return True
    else:
        foundDivisor = False
        for k in range(1, abs(b)):
            foundDivisor = foundDivisor or abs(a) * k == abs(b)
        return foundDivisor

IsPrime : Int -> Bool
IsPrime(n) -> Bool:
    if n < 2:
        return False
    else:
        foundDivisor = False
        for k in range(1, n):
            foundDivisor = foundDivisor or (1 < k and k < n and divides(k, n))
        return foundDivisor
```

Figure 4: Three computable functions defined in the function definition language of SD.

The language has some important limitations. First of all, it is a *strongly typed* language, meaning that we can statically rule out all cases where a function is given an argument that is outside of its domain. For example, let us look how the argument `n` (which is integer-valued) is used in the body of `IsPrime`. It is used in four places: twice as an argument of `<`, once as the second argument of `range`, and once as the second argument of `divides`. The system is automatically able to determine that all of these are safe.

In addition, one noteworthy aspect of this programming language is that it does not allow recursion. A function definition can only call functions that appear earlier; it cannot call itself or any of the functions that are defined later. On the positive side, this means that all functions are *total*, meaning that their execution always terminates after a finite number of steps. With recursive definitions this is not always guaranteed. One downside of this is that there are computable functions, such as the Ackermann function, that cannot be defined at all with a language like this. Of course, some future extension of SD can make the Ackermann function definable. Dealing with limitations such as this is one of the reasons why the framework for extensible languages in Chapter 4 was defined in the first place.

8 A proof explorer

In this chapter we look at the more practical side of producing textbooks that contain interactively explorable proofs. The interactive textbook is written in a text-based file format that is understood by two programs: the *proof checker* and the *proof explorer*. The proof checker is an noninteractive program that reads the input file, and checks that all definitions and theorems statements in the textbook are syntactically valid, and that all proofs are correct. If they are not, it produces an error message. The proof explorer handles the interactive part. It is implemented as a web page, with the interactive parts created using JavaScript. This makes it possible to read the textbooks on a wide variety of hardware, such as personal computers and tablets.

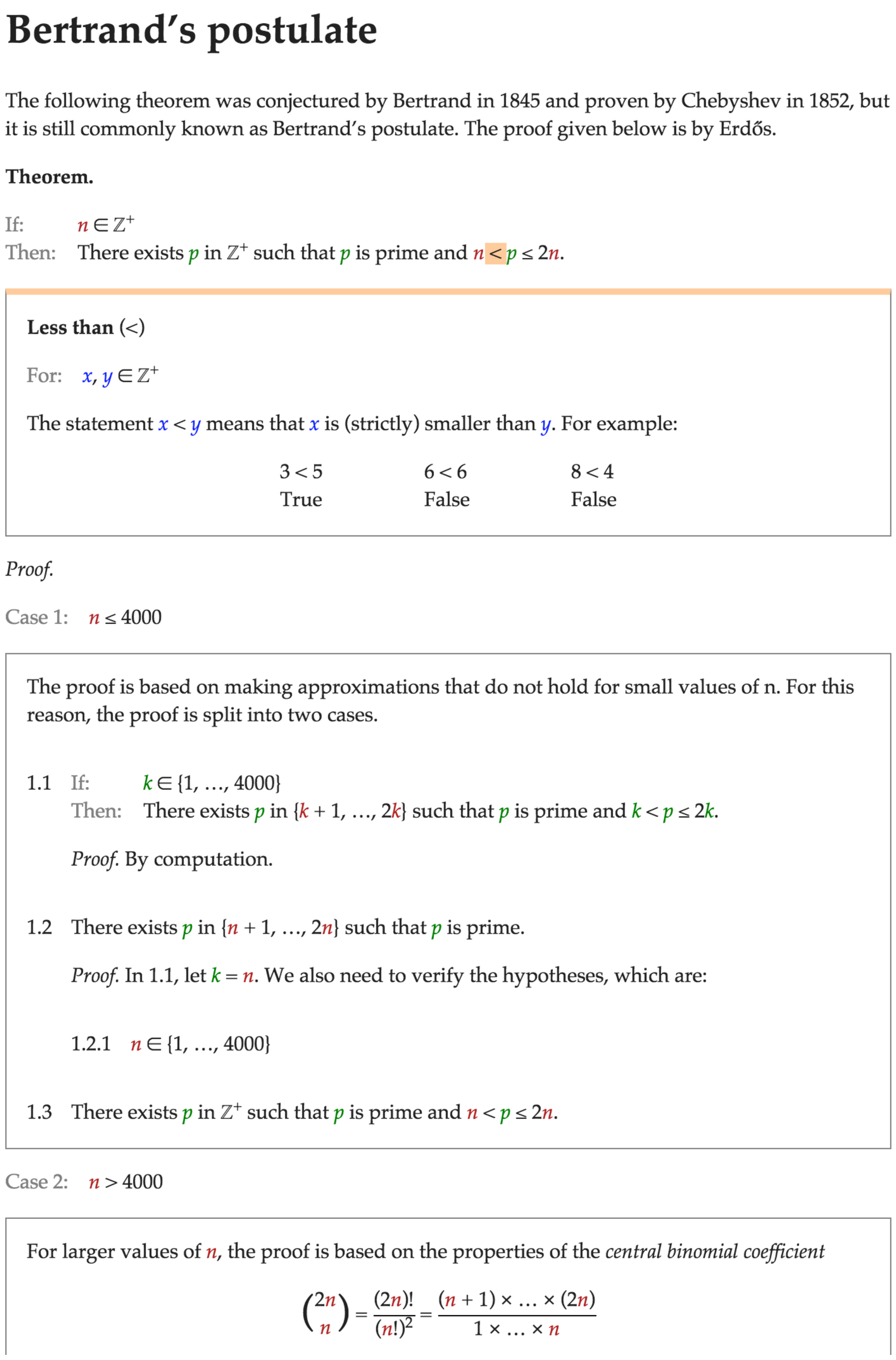
To illustrate the capabilities of the system, a formal proof of *Bertrand's postulate* was written. Bertrand conjectured in 1845 that for all positive integers n, there exists a prime number p such that $n < p \leq 2n$. This was originally proven by Chebyshev in 1850, but we use a later proof by Erdős instead [2]. This proof is divided into two cases: $n \leq 4000$ and $n > 4000$. The first case is handled by computation, and the second case is handled by studying the central binomial coefficient $\binom{2n}{n} = ((n+1) \cdot \ldots \cdot (2n))/(n!)$. This theorem was chosen because the proof as a whole is difficult enough to be interesting, and because the first case relies on a nontrivial amount of computation.

8.1 User interface

The paragraph at the top of Figure 5 ("The following theorem...") is a comment intended to provide historical information about the theorem. Comments like this can appear at all levels of the proof, and both cases in the proof begin with such a paragraph ("The proof is based on..." and "For larger values of n, ..."). The comments are not a part of the formal contents of the textbook, and they are ignored by the proof checker.

The statements in the proof are presented in English, rather than using a completely symbolic notation. For example, the conclusion "There exists p in $\mathbb{Z}^+$ such that p is prime and $n < p \leq 2n$" is a pretty-printed version of the symbolic statement '$\exists p \in \mathbb{Z}^+ : \text{IsPrime}(p) \wedge n < p \leq 2n$'. Similarly, step 1.1 internally uses the function $\mathit{range} : \mathbb{Z}^2 \to \textsc{Set}(Z)$, where $\mathit{range}(a, b) = \{k \in \mathbb{Z} \mid a \leq k \leq b\}$, but it is displayed as $\{a, \ldots, b\}$.

The reader can click any symbol to look up its meaning. In Figure 5 the reader has clicked the symbol '$<$' in the conclusion of the theorem statement, and in response, the proof explorer has displayed an explanation of its meaning. For a function or a relation, the explanation to be shown can depend on the types of its arguments. In this case, the proof explorer can infer that n and p both belong to $\mathbb{Z}^+$, and the explanation reflects this. In another context, the meaning of the symbol '$<$' could be different.



Bertrand's postulate

The following theorem was conjectured by Bertrand in 1845 and proven by Chebyshev in 1852, but it is still commonly known as Bertrand's postulate. The proof given below is by Erdős.

Theorem.

If: $n \in \mathbb{Z}^+$
Then: There exists p in $\mathbb{Z}^+$ such that p is prime and $n < p \leq 2n$.

Less than ($<$)

For: $x, y \in \mathbb{Z}^+$

The statement $x < y$ means that x is (strictly) smaller than y. For example:

$3 < 5$	$6 < 6$	$8 < 4$
True	False	False

Proof.

Case 1: $n \leq 4000$

The proof is based on making approximations that do not hold for small values of n. For this reason, the proof is split into two cases.

1.1 If: $k \in \{1, \ldots, 4000\}$
Then: There exists p in $\{k + 1, \ldots, 2k\}$ such that p is prime and $k < p \leq 2k$.

Proof. By computation.

1.2 There exists p in $\{n + 1, \ldots, 2n\}$ such that p is prime.

Proof. In 1.1, let $k = n$. We also need to verify the hypotheses, which are:

1.2.1 $n \in \{1, \ldots, 4000\}$

1.3 There exists p in $\mathbb{Z}^+$ such that p is prime and $n < p \leq 2n$.

Case 2: $n > 4000$

For larger values of n, the proof is based on the properties of the *central binomial coefficient*

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2} = \frac{(n+1) \times \ldots \times (2n)}{1 \times \ldots \times n}$$

Figure 5: An explorable proof of Bertrand's postulate.

The hierarchical numbering of steps is strongly influenced by Lamport's hierarchical proofs [33, 34]. The reader can choose to show or hide the proofs of the steps. In Figure 5 the proofs of steps 1.1 and 1.2 are visible, but the proofs of steps 1.2.1 and 1.3 are hidden. The numbering is useful for referring to earlier steps. For example, the proof of the step 1.2 refers to the step 1.1.

Suppose that v is a variable in a conditional statement, and one of its hypotheses is of the form '$v \in \ldots$' or of the form '$v = \ldots$'. In these cases it is unnecessary to explicitly show the range declaration for the variable v. For example, the conditional statement in step 1.1 should technically contain the range declaration '$k \in \mathbb{Z}$', but this is unnecessary, because it is implied by the hypothesis '$k \in \{1, \ldots, 4000\}$'. The proof explorer recognizes these situations, and can hide the range declaration.

8.2 Substitution

The *Hilbert system* is one of the oldest and best known systems for formal proof. It views a formal proof as a sequence of steps, where each step must be directly justifiable by one of the axioms, or one of the few *inference rules*. As an example, one of the inference rules, which is known by its Latin name *modus ponens*, says that if we have deduced both P and $P \to Q$, then we can infer Q. Hilbert style proofs are well suited for theoretical study because of their simple structure, but they do not closely reflect the way humans usually think about proofs. The system used here is much closer to an alternative system called *natural deduction*. The proofs are not thought of as linear sequences of steps; instead they have a hierarchical, tree-like structure.

Suppose that (r, h, c) is a conditional statement in SD_d. The basic method of proving such a theorem is by means of *conditional proof*. The variables that are introduced in the range declaration r are added to the list of known variables, and the hypotheses in the list h are added to the list of known statements. After this, we need to justify the conclusion.

Suppose that we wish to justify the conclusion c by appealing to some other conditional statement (r', h', c') that is already established to be true. We start by choosing expressions to be substituted to the variables declared in r'. The proof checker automatically performs those substitutions to the hypotheses h' and the conclusion c'. The author has to prove that the hypotheses hold, but the system automatically verifies that the conclusion matches exactly the original conclusion c. For example, suppose that we wish to prove the conditional statement

$$(\text{‘}x \in \mathbb{Z}\text{’}, [\text{‘}x > 1\text{’}], \text{‘}x^2 \geq 4\text{’}) \tag{16}$$

by appealing to the conditional statement

$$(\text{‘}a, b, c \in \mathbb{Z}\text{’}, [\text{‘}a \geq b\text{’}, \text{‘}b \geq 0\text{’}, \text{‘}b^2 = c\text{’}], \text{‘}a^2 \geq c\text{’}). \tag{17}$$

At this point we choose to substitute 'x' for 'a', '2' for 'b', and '4' for 'c'. The proof checker automatically performs the substitutions to (17), producing the hypotheses ['$x \geq 2$', '$2 \geq 0$', '$2^2 = 4$'] and the conclusion '$x^2 \geq 4$'. The proof checker automatically verifies that the substitution is type-safe, and that the conclusion '$x^2 \geq 4$'

exactly matches the conclusion in (16). To conclude the proof, the author must still prove that the hypotheses '$x \geq 2$', '$2 \geq 0$', and '$2^2 = 4$' hold under the assumption '$x > 1$'. This is also where the explorable nature of these formal proofs come in. These subproofs are hidden by default, but the reader can ask the proof explorer to show any of them.

8.3 Automatic computation

When discussing the proof of (16) in Section 8.2, we were left with three hypotheses to justify. Two of these, '$2 \geq 0$' and '$2^2 = 4$', seem particularly straightforward, since they do not contain any variables, and all of the functions involved are computable. In cases like this, the proof checker can simply perform the computation, and check that the result is true. This also applies to user-defined functions.

Beyond trivial statements like this, there are also more complicated situations where the ability to automatically verify computations is useful. Step 1.1. in Figure 5 states that if $k \in \{1, \ldots, 4000\}$, then there exists p in $\{k+1, \ldots, 2k\}$ such that p is prime and $k < p \leq 2k$. This conditional statement contains quantification, but the quantification happens only over finite sets, and therefore, it can be proven by computation. The proof checker enumerates all possible values of k, and for each k, checks that the statement "p is prime and $k < p \leq 2k$" evaluates to true for at least one $p \in \{k+1, \ldots, 2k\}$.

8.4 Automating trivial simplifications

The goal of the proof system is that the formal proofs correspond as closely as possible to how humans think of proofs. For example, suppose that the substitution $p =$ '$x < y$' is made into the formula '$\neg p$'. Technically speaking, the resulting formula would be '$\neg(x < y)$', but it is highly unlikely that such an expression would ever be written down as a part of a conventional proof. More likely, the author of the proof would perform the obvious simplification, and write '$x \geq y$' instead.

For this reason, the proof checker automatically performs certain trivial simplifications when doing substitutions. Every built-in relation in the language SD has a corresponding negated operator: $(=, \neq)$, $(<, \geq)$, and $(>, \leq)$. A similar simplification is performed whenever any one of these relations is substituted inside a negation. In addition, double negation elimination is performed automatically: if $p =$ '$\neg q$' is substituted into '$\neg p$', then the result is automatically simplified into q.

In addition, the substitution engine is aware of the associativity of the built-in functions '$+$', '$\cdot$', and '$\diamond$'. If $x =$ '$a + b$' is substituted into $2 + x$, then technically the resulting expression should be '$2 + (a + b)$'. However, since the substitution engine knows that addition is associative, it can automatically leave the parentheses out, resulting in '$2 + a + b$'.

The substitution engine is not—and is not supposed to be—a powerful system for automatically simplifying expressions. It does not try to make proofs easier to write; it tries to make them easier to read by hiding some of the more mechanical parts of the proof.

8.5 Chains of equalities

A *chain of equalities*, such as (11) in Section 6.1, is a common and straightforward style of writing proofs. Let us take a look at technical details that are hidden inside such a proof. The first step is to prove that $x_1 = \ldots = x_n$. By definition, this is equivalent to $x_1 = x_2 \wedge \ldots \wedge x_{n-1} = x_n$, so we can proceed by proving the equations $x_1 = x_2, \ldots, x_{n-1} = x_n$ separately. When proving one of these equations (say, $x_k = x_{k+1}$), it is often the case that the expressions x_k and x_{k+1} are mostly the same, but differ in some small part. For example, to prove that $a + b \cdot (2 \cdot 3 + c) = a + b \cdot (6 + c)$, it is enough to prove that $2 \cdot 3 = 6$. Technically, this works by using the general principle that whenever $x = y$, we also have $P(x) = P(y)$. If we substitute $x = 2 \cdot 3$, $y = 6$, and $P(z) = a + b \cdot (z + c)$, then we find that $2 \cdot 3 = 6$ implies that $a + b \cdot (2 \cdot 3 + c) = a + b \cdot (6 + c)$. After all the equations $x_1 = x_2, \ldots, x_{n-1} = x_n$ have been proven, we can show that $x_1 = x_n$ by repeatedly using the transitive property of equality.

This description may seem complicated, but this is only because after years of training all users of mathematics have internalized the ideas behind chains of equalities to such a level that they do not have to consciously think about "transitivity of equality" or other such the details. To take advantage of this, the system has special supports for chains of equalities. It combines the features presented above into a single building block. From a proof-theoretic point of view this is redundant, since all proofs could also be written without this feature. However, if we want to write proofs in human-readable form, features like this are indispensable.

Since the logical relation $\leftrightarrow$ can be seen as a special case of equality, these proofs work entirely analogously for chains of equivalences as well.

8.6 Working with function definitions

Since the conditional statement language SD makes it possible to define new functions, it should also make it possible to prove theorems about those functions. The functions are defined using a simple programming language, and there are existing fully fledged proof systems, such as Hoare logic, that can be used to prove properties of complicated programs.

The approach taken here is lighter. It is based on recognizing certain simple patterns in function definitions, and providing customized support for them. To illustrate this, consider the following functions:

```
funcA : Int * Int -> Int
funcA(a, b):
    return a * b - 2

funcB : Int * Int -> Int
funcB(a, b):
    c = a + b
    return c * c

abs : Int -> Int
abs(n):
    if n >= 0:
        return n
    else:
        return -n

sum : List(Int) -> Int
sum(list):
    ans = 0
    for k in list:
        ans = ans + k
    return ans
```

The first function ($\mathit{funcA} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$) is the easiest, since its definition consists of a single return statement. From this function definition the system can infer that if $a, b \in \mathbb{Z}$, then $\mathit{funcA}(a, b) = a \cdot b - 2$. The second function ($\mathit{funcB} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$) is slightly trickier, since it contains an auxiliary variable c. One option would be to automatically expand this variable. In this case the system would infer from this definition that if $a, b \in \mathbb{Z}$, then $\mathit{funcB}(a, b) = (a+b) \cdot (a+b)$. However, expanding all auxiliary variables can sometimes make the theorem harder to read. For this reason, the system keeps the variable as is, and produces the following theorem instead:

- Suppose that $a, b, c \in \mathbb{Z}$. If $c = a + b$, then $\mathit{funcB}(a, b) = c \cdot c$.

The next function definition ($\mathit{abs} : \mathbb{Z} \to \mathbb{Z}$) consists of an if/else statement with a simple return statement in each branch. In this case the system will produce two theorems:

- Suppose that $n \in \mathbb{Z}$. If $n \geq 0$, then $\mathit{abs}(n) = n$.
- Suppose that $n \in \mathbb{Z}$. If $n < 0$, then $\mathit{abs}(n) = -n$.

The second theorem takes advantage of the automatic simplification of negations from Section 8.4. This is why it contains the hypothesis '$n < 0$' instead of '$\neg(n \geq 0)$'.

The last function definition ($sum : \textsc{List}(\mathbb{Z}) \to \mathbb{Z}$) is the most involved. The system is able to detect that it is a specific instance of this common pattern:

```
f : List(A) -> B
f(list):
    ans = init
    for elem in list:
        ans = update(ans, elem)
    return ans
```

There are two distinct cases of interest here. First, if $list = [\,]$, then the loop is never executed, and we have $f([\,]) = init$. Second, let us compare what happens if we evaluate $f(list)$ versus $f(list \diamond [elem])$. The value of the parameter *list* does not affect the initial value or the updates, so the execution starts in the same way. The only difference is that when $f(list)$ has looped over the entire list, $f(list \diamond [elem])$ will run the update function one more time. Therefore, we can conclude that $f(list \diamond [elem]) = update(f(list), elem)$. In the case of the function $sum : \textsc{List}(\mathbb{Z}) \to \mathbb{Z}$ this means that we get the following two theorems:

- $sum([\,]) = 0$.
- Suppose that $list \in \textsc{List}(\mathbb{Z})$ and $k \in \mathbb{Z}$. Then $sum(list \diamond [k]) = sum(list) + k$.

In its current form, the system is limited to function definitions that strictly follow one of these patterns. Consequently, the system is unable to prove any properties of functions whose definitions are more complicated. This includes, for instance, the definitions of $divides : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ and $IsPrime : \mathbb{Z} \to \mathbb{B}$ given in Section 7.3. One possible future extension is to add support for Hoare logic, or some other system like it. In the meantime, it is always possible to write all functions definitions in a way that fits these patterns. However, this may require new auxiliary functions to be defined. As an example, here is an alternative definition of the function $IsPrime : \mathbb{Z} \to \mathbb{B}$ that follows these patterns.

```
anyElementDivides : List(Int) * Int -> Bool
anyElementDivides(list, n):
    foundDivisor = False
    for elem in list:
        foundDivisor = foundDivisor or divides(elem, n)
    return foundDivisor

IsPrime : Int -> Bool
IsPrime(n) -> Bool:
    return anyElementDivides(range(2, n), n)
```

9 Conclusions

By using formal proofs it is possible to develop interactive textbooks of mathematics, where the student can request a more detailed explanation of any part of a proof that he or she does not understand. Such a system would be useful for research articles as well, but at the current time textbooks are a more realistic goal. Formal, computer-verifiable proofs are also helpful for verifying proofs that are unusually complicated, and therefore difficult for humans to verify. In the context of textbooks, however, formal proofs would be used not so much to avoid errors, but to ensure that the proof explorer is able to explain every part of the proof.

As a first step, the author of such a textbook should choose appropriate formal languages for representing definitions, theorem statements, and proofs. Any fixed choice of languages cannot be expressive enough for all mathematics, so the choices have to be made by considering the particular requirements of the textbook. For example, if the textbook needs to talk about integers, then it either needs a language where integers are a primitive concept, or a language that is expressive enough to adequately define integers in terms of the primitive concepts it does support.

In addition to giving a general overview of formal proofs, this thesis has three main contributions. The first contribution is the theoretical framework for extensible languages defined in Chapter 4. The fact that propositional logic can be extended to first-order logic is, of course, well known. The importance of defining a framework like this lies in the fact that it says explicitly what kind of future extensions beyond first-order logic are allowed. In particular, definitions should never become obsolete in future extensions of a conditional statement language. For example, let $T = \mathrm{Th}(\mathbb{R}, +, \cdot, -, 0, 1)$ be the full first-order theory of the ring of reals. One of the models of T is, by definition, the ring $(\mathbb{R}, +, \cdot, -, 0, 1)$, but it also has models that are not isomorphic to it, such as the ring of algebraic reals. First-order logic is unable to distinguish between these two models, but there are extensions of first-order logic that can make this distinction. For this reason, the theory T is not an adequate definition of the ring of real numbers in general, although it would be adequate for the purposes of first-order logic.

The second contribution is the conditional statement language SD. Because it is a sublanguage of set theory, its notation should be immediately recognizable for mathematicians. However, unlike the full language of set theory or the language of higher-order logic, it does not allow unrestricted quantification over sets whose cardinality is larger than $2^{\aleph_0}$. Therefore, it manages to avoid some conceptually difficult questions, such as the question of whether the continuum hypothesis has a definite truth value or not. By expressive power, it corresponds to second-order arithmetic, but it has a broader computational emphasis. At its current form the language is more suited to textbooks whose subject belongs to discrete rather can continuous mathematics: elementary number theory, combinatorics, graph theory, or abstract algebra, but not, for example, analysis.

The third contribution is the prototype of the proof explorer, and the accompanying discussion of its design goals. The proof explorer displays proofs hierarchically, and can answer two kinds of questions the reader might have when examining a statement that is a part of a proof: what is the meaning of some symbol in that statement, and why is that statement true.

Possible future work might include extending the language SD so that it can conveniently work with real and complex numbers. On the more practical side, it would be useful to improve the textbook authoring tools. The prototype proof explorer is mostly concerned with the experience of the reader, and the interface that the author has to use is still rather crude. Bertrand's postulate is a nontrivial theorem, but it is still just one theorem. With slightly more sophisticated authoring tools it should be feasible to write a complete textbook instead. Writing a complete textbook, using it on some university course, and gathering feedback from the teacher and the students would undoubtedly be valuable for further development.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of mathematics*, pages 781–793, 2004.

[2] M. Aigner, G. M. Ziegler, K. H. Hofmann, and P. Erdős. *Proofs from the Book*, volume 274. Springer, 2010.

[3] K. Appel, W. Haken, et al. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82(5):711–712, 1976.

[4] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS Press, 2009.

[5] R. Boyer et al. The QED manifesto. *Automated Deduction–CADE*, 12:238–251, 1994.

[6] P. J. Cameron. *Introduction to algebra*. Oxford University Press Oxford, 2008.

[7] A. Clark. *Elements of abstract algebra*. Courier Corporation, 1984.

[8] G. Dowek. *Proofs and Algorithms: An Introduction to Logic and Computability*. Springer Science & Business Media, 2011.

[9] E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 255–264. ACM, 2008.

[10] H. B. Enderton. Second-order and higher-order logic. In *The Stanford Encyclopedia of Philosophy*. Fall 2015 edition.

[11] S. Feferman. Is the continuum hypothesis a definite mathematical problem. *Draft of paper for the lecture to the Philosophy Dept., Harvard University*, pages 2011–2012, 2011.

[12] M. Ganesalingam and W. T. Gowers. A fully automatic problem solver with human-style output. *CoRR*, abs/1309.4501, 2013.

[13] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

[14] K. Gödel. The consistency of the axiom of choice and of the generalized continuum-hypothesis. *Proceedings of the National Academy of Sciences of the United States of America*, 24(12):556, 1938.

[15] G. Gonthier. A computer-checked proof of the four colour theorem. *Microsoft Research Cambridge, Research report*, 148, 2005.

[16] T Gowers, J. Barrow-Green, and I. Leader. *The Princeton companion to mathematics*. Princeton University Press, 2010.

[17] T. Granlund and the GMP development team. The GNU multiple precision arithmetic library, edition 6.0.0, 25 march 2014.

[18] T. Hales. *Dense sphere packings: A blueprint for formal proofs*, volume 400. Cambridge University Press, 2012.

[19] T. C. Hales. Introduction to the Flyspeck project. *Mathematics, Algorithms, Proofs*, 5021:2006, 2005.

[20] T. C. Hales. A proof of the Kepler conjecture. *Annals of mathematics*, pages 1065–1185, 2005.

[21] T. C. Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, 2008.

[22] T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, et al. A formal proof of the Kepler conjecture. *arXiv preprint arXiv:1501.02155*, 2015.

[23] G. H. Hardy, E. M. Wright, D. R. Heath-Brown, and J. H. Silverman. *An introduction to the theory of numbers*, volume 4. Clarendon press Oxford, 1979.

[24] J. Harrison. Formalized mathematics. Technical Report 36, Turku Centre for Computer Science (TUCS), 1996.

[25] J. Harrison. Formal proof—theory and practice. *Notices of the AMS*, 55(11):1395–1406, 2008.

[26] J. Harrison. Formalizing an analytic proof of the prime number theorem. *Journal of Automated Reasoning*, 43(3):243–261, 2009.

[27] E. T. Hecke. *Lectures on the theory of algebraic numbers*, volume 77. Springer Science & Business Media, 2013.

[28] S. K. Jakobsen. Cauchy's functional equation, 2010.

[29] T. J. Jech. *Set theory*, volume 79. Academic press, 1978.

[30] M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of Nqthm. In *Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on*, pages 23–34. IEEE, 1996.

[31] K. C. Klement. Propositional logic. In *The Internet Encyclopedia of Philosophy.*

[32] B. Konev and A. Lisitsa. A SAT attack on the Erdős discrepancy conjecture. *CoRR*, abs/1402.2184, 2014.

[33] L. Lamport. How to write a proof. *American Mathematical Monthly*, pages 600–608, 1995.

[34] L. Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, 2012.

[35] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[36] A. L. Mann. *A Case Study in Automated Theorem Proving: Otter and EQP*. PhD thesis, University of Colorado, 2003.

[37] D. Marker. *Model theory: an introduction*. Springer Science & Business Media, 2002.

[38] W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[39] N. Megill. Metamath: A computer language for pure mathematics. 1997.

[40] A. Nerode and R. A. Shore. *Logic for applications, Graduate Texts in Computer Science*. Springer-Verlag, New York,, 1997.

[41] D. Pigozzi. Equational logic and equational theories of algebras. *Purdue University Computer Science Technical Reports*, 1975.

[42] B. Poonen. Hilbert's tenth problem over rings of number-theoretic interest. *Note from the lecture at the Arizona Winter School on "Number Theory and Logic*, 2003.

[43] B. Poonen. Undecidable problems: a sampler. Technical report, 2012.

[44] W. V. O. Quine. *Philosophy of logic*. Harvard University Press, 1986.

[45] J. A. Robinson. Definability and decision problems in arithmetic. *The Journal of Symbolic Logic*, 14(02):98–114, 1949.

[46] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[47] J. B. Rosser. *Logic for mathematicians*. McGraw-Hill New York, 1953.

[48] H. Schwichtenberg. Minimal logic for computable functions. In *Logic and Algebra of Specification*, pages 289–320. Springer, 1993.

[49] R. I. Soare. Computability and recursion. *Bulletin of Symbolic Logic*, 2(03):284–321, 1996.

[50] R. M. Solovay. A model of set-theory in which every set of reals is Lebesgue measurable. *Annals of Mathematics*, pages 1–56, 1970.

[51] A. Tarski. A decision method for elementary algebra and geometry. 1951.

[52] F. Wiedijk. *The seventeen provers of the world: Foreword by Dana S. Scott*, volume 3600. Springer Science & Business Media, 2006.

[53] A. Wiles. Modular elliptic curves and Fermat's last theorem. *Annals of Mathematics*, pages 443–551, 1995.

[54] W. H. Woodin. The continuum hypothesis, part I. *Notices of the AMS*, 48(6):567–576, 2001.

[55] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. *ACM SIGPLAN Notices*, 47(6):283–294, 2012.