2017

Pretending To Be Human: An Automated Theorem Prover To Write Mathematical Proofs

Khoa Nguyen
The College of Wooster, knguyen18@wooster.edu

PRETENDING TO BE HUMAN: AN AUTOMATED THEOREM PROVER TO WRITE MATHEMATICAL PROOFS

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelors of Arts in the
Department of Mathematics and Computer Science
at The College of Wooster

by
Khoa Le Tue Nguyen
The College of Wooster
2017

Advised by:

Dr. James Hartman

Dr. Sofia Visa

Abstract

This project explores the theories of automated theorem proving and lambda calculus, and seeks to improve upon a current implementation of an automated theorem prover that produces human-like output. This software component is based on a published article titled "A Fully Automatic Theorem Prover with Human-Style Output" by M. Ganesalingam and W.T. Gowers [7], which outlines a program called `robotone` containing a first-order logic system (with a few modifications) able to automatically write simple direct proofs. It is designed to mimic different strategies that a human mathematician would use when facing a problem, and to write human-style proofs. The current prototype can solve and write eighteen proofs, seven of which are developed from this project with the addition of a mathematics library customized for the course MATH-332 Real-Analysis I at The College of Wooster. Future work includes further investigation on how human mathematicians construct their proofs, and a possible implementation of a more dynamic tactic mechanism to emulate humans' thinking process better.

Acknowledgments

To my parents and brother for enabling me to have a great education both inside and outside of school.

To many friends who help me get over the various difficulties of being away from home for most of my teenage and studying in the US for the last five and a half years.

To many great teachers and professors who encourage me to do things I never thought about doing.

Thank you.

Contents

List of Listings

CHAPTER 1

Introduction

Automated theorem proving is a relatively new and competitive research area in mathematics and computer science. Many existing systems have made significant progress and compete against each other at international conferences and tournaments [8]. However, many challenges lie ahead in this field, particularly regarding the user-friendliness of such systems, and whether it is possible to integrate the results of recent research into traditional mathematics education in the undergraduate level.

The goal of this project is to create a human-friendly automated theorem prover that produces human-style proofs, and acts as a helper for students in the course MATH-33200 Real Analysis I at The College of Wooster. The idea is based on a published research article titled "A Fully Automatic Theorem Prover with Human-Style Output" by M. Ganesalingam and W.T. Gowers, which outlines a program called `robotone` containing a first-order logic system (with a few modifications) able to automatically write simple direct proofs. We seek to tailor the scope of this program to mostly proofs in Real Analysis, as well as to improve its usability. This means creating a set of Real Analysis problems and a library of Real Analysis definitions and theorems; lowering the software environment requirements to install and run `robotone`; creating a user-friend graphical user interface; as well as understanding and hopefully adding to the proof-writing capabilities of the existing code.

1.1 Proof Techniques

Proof writing is an important skill that all mathematics students at the college level should have, yet many struggle with forming a logical argument to prove a statement from the given definitions and theorems. In a similar sense, we face challenges in creating a software system that can represent mathematical facts in a way that both the human and the machine can understand, to create and make use of the automatically generated proofs.

A theorem is a statement that describes a pattern or relationship among quantities or structures, and a proof is a justification of the truth of a theorem. There are different strategies, or techniques, that human mathematicians employ when constructing a proof. We explore the three most common proof techniques at the undergraduate level: direct, contrapositive, and contradiction.

1.1.1 Direct Proofs

Direct proofs should be attempted whenever possible, since it is often the most straightforward proof. Given a conditional $P \implies Q$ where P represents given facts, the outline of a direct proof is as followed:

Proof. Assume P.

...

(Deduce Q over a sequence of steps).

...

Then Q.

Thus $P \implies Q$. □

Example 1.1. If a sequence (x_n) of real numbers converges, then it is Cauchy.

Proof. (Direct proof) Suppose (x_n) is a convergent sequence, and let its limit be denoted by L, or $\lim x_n = L$. Let $\epsilon > 0$.

Then there exists k such that $n \geq k$ implies $|x_n - L| < \frac{\epsilon}{2}$.

Let $n, m \geq k$ be arbitrary. Then

$$|x_m - x_n| = |(x_m - L) - (x_n - L)| \leq |x_m - L| - |x_n - L| < \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon.$$

Thus (x_n) is a Cauchy sequence as desired. □

Here, P is "a sequence (x_n) of real numbers is convergent," and Q is "[(x_n)] is a Cauchy sequence." We first assume that (x_n) is indeed convergent, and through a series of steps, arrive at the conclusion that (x_n) is a Cauchy sequence, meaning for each $\epsilon > 0$, there exists an $N \in \mathbb{N}$ such that for all $m, n \leq N, |x_m - x_n| < \epsilon$.

1.1.2 Contrapositive Proofs

Contrapositive proofs are based on the tautology (P implies Q) $\Leftrightarrow$ ($\sim Q$ implies $\sim P$). Thus the contrapositive proof of (P implies Q) can be outlined as the direct proof of ($\sim Q$ implies $\sim P$) as followed:

Proof. Assume $\sim Q$.

...

(Deduce $\sim P$ over a sequence of steps).

...

Then $\sim P$.

Thus $P \implies Q$. □

Example 1.2. If a sequence of real numbers (x_n) converges to L, then any subsequence of (x_n) also converges to L.

Proof. (By contraposition). Assume there exists a subsequence (x_{n_i}) of (x_n) such that (x_{n_i}) does not converge to L.

Then for all $k \in \mathbb{N}$ and $\epsilon > 0 \in \mathbb{R}$, there exists some n_{i0} such that $n_{i0} > k$ and $|x_{n_{i0}} - L| \geq \epsilon$.

Thus for all $k \in \mathbb{N}$ and $\epsilon > 0 \in \mathbb{R}$, there exists some $n = n_{i0}$ such that $n > k$ and $|x_n - L| \geq \epsilon$, i.e. (x_n) does not converge to L.

Therefore if a sequence of real numbers (x_n) converges to L, then any subsequence of (x_n) also converges to L. □

1.1.3 Contradiction Proofs

Unlike direct and contrapositive proofs, a proof by contradiction can be used for any proposition, not just those involving conditionals. If P is the statement or theorem to be proved, the outline of a contradiction proof is as followed:

Proof. Assume $\sim P$.

...

(Deduce some fact R and also $\sim R$ over a sequence of steps).

...

Thus R and $\sim R$, which is a contradiction.

Therefore P. □

Example 1.3. Archimedean Property part (i): Given any number $x \in \mathbb{R}$, there exists an $n \in \mathbb{N}$ satisfying $n > x$.

Proof. (By contradiction). Suppose that there exists $x \in \mathbb{R}$ such that $\forall n, n \leq x$, i.e. $\mathbb{N}$ is bounded above.

Thus $sup\ \mathbb{N}$ exists. Let $q = sup\ \mathbb{N}$. Then $n \leq q\ \forall n \in \mathbb{N}$. (*)

Consider $q - 1 < q$ which is not an upper bound for $\mathbb{N}$. By the property of *sup*, there exists $m \in \mathbb{N}$ such that $q - 1 < m \Leftrightarrow q < m + 1$, but $m + 1$ is a natural number which is supposed to be less than or equal to q by (*), so this is a contradiction.

Therefore for any number $x \in \mathbb{R}$, there exists an $n \in \mathbb{N}$ satisfying $n > x$. □

1.2 What is Automated Theorem Proving?

Logical reasoning distinguishes humans from other species. We use it to solve almost anything from everyday matters to mathematical problems. Logical reasoning can be defined to be the process of drawing conclusions from given facts, where these conclusions must inevitably follow from only the given facts without resorting to probability or common sense.

As technology rises, the following question naturally comes with it: Can computers be taught or programmed to reason logically? More or less, yes, in relatively more defined areas such as mathematics. We define *inference rules* for a program to dictate how a conclusion or new facts can be derived from the given facts, how to discard irrelevant facts, and how to rephrase facts into an equivalent but more applicable form [20]. Most theorem provers are built based on the concepts of *artifacts* and *inference rules*.

1.2.1 Artifacts

Artifacts here means the resources that the program is allowed to use in trying to solve or prove the given problems. They include assumptions and axioms, special facts and the special hypothesis, and the negation or denial of the goal or theorem [20].

- Assumptions and axioms: These belong to a set of clauses that describe the problem domain, in particular these are statements regarded as true in the problem domain. This set should be large enough to encompass all necessary concepts in the problem domain that are relevant to the goal of the proof.

- Special facts: These belong to another set of clauses called the *special hypothesis*. It is basically the specifically given facts and assumptions directly related to the proof that the program needs to produce, in addition to the general, broader assumptions and axioms of the problem domain described earlier.

- Negation or denial of the goal or theorem: This is the set of clauses that establish the impossibility of the given problem. It can be used in contraposition or contradiction proofs, or to check for the completion of the proof.

1.2.2 Inference Rules

An *inference rule* is an algorithm yielding a conclusion that follows inevitably and logically from some set of given facts and hypotheses. Inference rules are applied in order to derive new information from the given facts, but the resulting conclusions may or may not be actually new. There are several different types of inference rules defined in automated reasoning theory: unification, binary resolution, UR-resolution, hyperresolution, negative hyperresolution, paramodulation, unit resolution, and factoring. More details about these reference rules can be found in Chapter 6 of [20].

This project's software component breaks the different types of inference rules into even smaller procedures called *tactics*. The tactics listed in Chapter 4 in descending priority are from the existing source code, and the explanations are from Ganesalingam and Gowers' paper [7]. There are other smaller, intermediate steps that do not contribute much to the proving process and thus are not included.

1.3 Current Theorem Provers and Research

There are several existing theorem provers, such as Isabelle/HOL [16], SAT [14], EQP [6], Vampire [19], ML4PG [9], and Coq [4]. These prove to be successful in assisting human mathematicians in solving many difficult, sometimes decades-old problems, such as the Robbins conjecture (EQP), the Kepler conjecture (Isabelle/HOL), and the designing and verification of processor chips [13]. Some of these provers are actually not automated but instead interactive, i.e. the user has to input something other than the problem statement into the prover in order for it to proceed, in some cases because the main purpose of the prover is actually to verify a proof written by a human, not to produce its own proof. The time and effort required to master these systems can be discouraging to most human mathematicians.

The program described in Ganesalingam's and Gowers' paper is one of the most recent attempts at bringing technology closer to the work of human mathematicians. The authors have many reasons behind developing a system with human-style output. They cite evidence that many mathematical proofs in published literature are actually incorrect. In addition, many correct, but complicated proofs are very difficult to verify and thus a formalized proof can be of value. Thus, a system having human-style output, placing less emphasis on correctness and more on explanations, may be able to help mathematicians in their daily work [7]. As pioneers, certainly some shortcomings are present in the software prototype enclosed with the paper, which we seek to make some progress on:

- Proofs by contradiction and contraposition are not yet possible.
- The process of adding or modifying given problems and existing facts requires recompilation of the entire code base.
- The user interface is not exactly friendly to non-technical users.

CHAPTER 2

Formalized Logic

Computers have only been around for a few decades while mathematics was born centuries ago. And yet, computer programs already contribute to finding answers to unsolved mathematical questions and conjectures, such as the four-color theorem, Robbin's conjecture, and the Kepler conjecture [13]. However, a large part of all of these proofs is still the work of human mathematicians, because even though first-order logic (FOL) contains decidable fragments and can power fully automated theorem provers, it is not expressive enough for complex problems. On the other hand, Higher Order Logic (HOL) can express complex problems, but includes many undecidable algorithms and proof methods, rendering automation much more difficult [8]. This chapter gives an overview of different types of formalized logic and its application in current proof assistants.

Logic can be considered the language of mathematics. A natural language, like English, has the vocabulary and grammar rules (syntax) to enable a speaker to express their ideas in a meaningful way (semantics). Analogously, logic has specifically defined elements and rules to correctly represent mathematical ideas.

2.1 Propositional Logic

As suggested by its name, the elements of propositional logic consists of Boolean variables called propositions (evaluating to either true or false), and logical connectives (such as AND, OR, XOR, etc.). Propositions can be combined by Boolean logical connectives into *propositional formulas*. Thus propositional logic has no variables other than Boolean variables, no quantifiers, and no function symbols [17].

The following notions are important in propositional logic:

Definition 2.1 Interpretation (Valuation): *An interpretation (or valuation) I over the propositions* $\{P_1, P_2, ..., P_n\}$ *is a function from the propositions* $\{P_1, P_2, ..., P_n\}$ *to the truth values (true or false). There are* 2^n *interpretations over* $\{P_1, P_2, ..., P_n\}$*, since each* P_i *can be either true or false. We write* $I \models P$ *if* $I(P) = true$ *(I satisfies P) and* $I \not\models P$ *if* $I(P) = false$*. For Boolean formulas A and B, we define their truth in I by the following rules:*

- $I \models \neg A$ *iff* $I \not\models A$
- $I \models A \wedge B$ *iff* $I \models A$ *and* $I \models B$
- $I \models A \vee B$ *iff* $I \models A$ *or* $I \models B$
- $I \models A \supset B$ *iff* $I \models \neg A$ *or* $I \models B$
- $I \models A \equiv B$ *iff* $I \models A \supset B$ *and* $I \models B \supset A$.

Definition 2.2 Satisfiability: *A formula A over* $\{P_1, P_2, ..., P_n\}$ *is satisfiable if there exists an interpretation I over* $\{P_1, P_2, ..., P_n\}$ *such that* $I \models A$*. Consequentially, if a formula A is not satisfiable, A is called unsatisfiable or contradictory, i.e. for all I over* $\{P_1, P_2, ..., P_n\}, I \not\models A$*.*

Definition 2.3 Validity: *A formula A over* $\{P_1, P_2, ..., P_n\}$ *is valid if for all interpretations I over* $\{P_1, P_2, ..., P_n\}, I \models A$*. Consequentially, A is called invalid otherwise, i.e there exists an I such that* $I \not\models A$*.*

Definition 2.4 Equivalence: *Two formulas A and B are equivalent iff the formula $A \equiv B$ is valid.*

Thus, based on propositional logic, the theorem proving problem is to determine whether a given formula A is valid. We notice that A is valid iff $\neg A$ is unsatisfiable. Interestingly, determining whether a Boolean formula A is satisfiable is one of the current NP-complete problems, i.e. at the moment it cannot be solved in polynomial time. Using a truth table means checking each of the 2^n possible interpretations, resulting in an exponential runtime. However, it is possible to "use partial interpretations to obtain a satisfiability testing procedure more efficient than truth tables", which some proof systems attempt to do [17].

2.2 First-Order Logic

2.2.1 Elements of Syntax and Semantics

Even though propositional logic is capable of expressing many problems, the fact that it does not include additional non-Boolean variables, quantifiers, and functions is rather limiting. First-order logic allows a wider variety of problems to be expressed, with non-Boolean *variables*, *function symbols*, and *predicate symbols* in addition to the existing elements propositional logic. Moreover, function and predicate symbols can take a number of arguments. This number is called *arity* of the function or predicate symbol. If the arity is 0, then the function symbol is called a *constant symbol* or *individual constant*. Thus, we can roughly say that propositional logic is a "subset" of first-order logic.

Formally, we also define the following additional elements of first-order logic as below [17] [20]:

1. A *term* is either:

- A variable
- A constant symbol
- An expression $f(t_1, ..., t_n)$ where f is a function symbol of arity n and the t_i are terms.

2. An *atom* is an expression $P(t_1, ..., t_n)$ where P is a predicate symbol of arity n and the t_i are terms.

3. A formula is defined inductively by the following rules:

   - If A is an atom, then A is a formula.
   - If A is a formula, then $\neg A$ is a formula.
   - If A, B are formulas, then $(A \wedge B), (A \vee B), (A \supset B)$, and $(A \equiv B)$ are also formulas.
   - If A is a formula and x is a variable, then $(\forall x)A$ and $(\exists x)A$ are formulas.

   A *quantifier-free* formula is one without quantifiers. In a formula of the form $(\forall x)A$ or $(\exists x)A$, A is called the *scope* of the respective quantifier $(\forall x)$ or $(\exists x)$, and x is said to be *bound* to the quantifier.

4. A *literal* is an atom (positive literal) or the negation of an atom (negative literal). Thus by the definition of formulas, a literal is a formula.

With these extra elements, the definition of an interpretation in first-order logic is slightly different than that in propositional logic. We define interpretation I as the following:

Definition 2.5 Interpretation (first-order logic) [17]: *An interpretation, or structure, I consists of a domain D, where D is a nonempty collection of objects, and assignments of meanings to variables and constants. Let $\alpha^I \rightarrow \alpha$ denote "I assigns $\alpha^I \in D$ to α". Then:*

- *For a variable x: $x^I \rightarrow x$.*
- *For a constant a: $a^I \rightarrow a$.*
- *For a function constant f: $f^I \rightarrow f$, where $f^I : D^n \rightarrow D$ and n is arity of f.*
- *For a predicate constant P: $P^I \rightarrow P$, where $P^I : D^n \rightarrow \{true, false\}$ and n is arity of P.*
- *For a term A of the form $f(t_1, ..., t_n)$ where f is a function symbol of arity n and the t_i are terms: $A^I \rightarrow A$, where $A^I = f(t_1, ..., t_n)^I = f^I(t_1^I, ..., t_n^I) \in D$.*

For interpretations I and J in D, $I \equiv J(mod\ x)$ iff A^I and A^J are identical for all function symbols, predicate symbols, and variables different from x. The definitions of satisfiability, validity, and equivalence in first-order logic are analogously similar to those in propositional logic. The truth value of formulas A, B in an interpretation I is determined with the following rules [17]:

- $I \models P(t_1, ..., t_n)$ if $P^I(t_1^I, ..., t_n^I)$ is true.
- $I \models A \wedge B$ iff $I \models A$ and $I \models B$
- $I \models A \vee B$ iff $I \models A$ or $I \models B$
- $I \models \neg A$ iff $I \not\models A$
- $I \models A \supset B$ iff $I \models \neg A$ or $I \models B$
- $I \models A \equiv B$ iff $I \models A \supset B$ and $I \models B \supset A$.
- $I \models (\forall x)A$ iff for all $J \equiv I(mod\ x)$, $J \models A$.
- $I \models (\exists x)A$ iff there exists a $J \equiv I(mod\ x)$ and $J \models A$.

2.2.2 First-Order Proof Systems

There are many existing first-order proof systems, such as Hilbert-style systems, Gentzen-style systems, etc. [17]; however, they are outside the scope of this project. Therefore, this section is devoted to only describing the system implemented in the software component of this project, which is based on the "waterfall" architecture of the Boyer-Moore provers with a few modifications [7].

Generally speaking, the system attempts to prove a claim by considering it a goal, and applying appropriate heuristics, also called tactics, to transform it. These tactics are ranked by their "attractiveness" based on their effects on the goal when applied. In the software associated with this project, the goal has a list of assumptions and a list of targets to be deduced from them [7].

The Boyer-Moore theorem prover has 7 main proof techniques [3]:

1. Simplification.
2. Destructor elimination.
3. Cross-fertilization.
4. Generalization.
5. Elimination of irrelevance.
6. Induction.

Ganesalingam and Gowers built *robotone* based loosely on these 7 techniques, developing more specific tactics that are applied to goals in order of priority. These tactics are discussed in more details in the next chapter. In addition, the program also utilizes the following newly defined data structures [7]:

1. A *box* is either a *nontrivial box* or the special box $\top$ (which essentially means the truth value *true*.

2. A *nontrivial box* has a list of variables, a list of formulas (assumptions), and a list of *targets*.

3. A *target* is either a formula or a list of boxes.

2.3 Higher-Order Logic (HOL)

We will not go into as much details of higher-order logic (HOL) as we do for propositional and first-order logic, partly because this project only uses a minimal amount of HOL. Simply put, HOL can be written as follows [16]:

$$\text{HOL} = \text{Functional Programming} + \text{Logic}$$

Like first-order logic, HOL also has variables (of different types), terms, and formulas. HOL introduces λ-terms and formulas, which replace first-order terms in first-order logic, and allows for the quantification of predicate symbols [15].

The software component of this project does not utilize much of HOL except for the use of metavariables on the logic part, and of Haskell, a purely functional language, on the programming part. According to Ganesalingam and Gowers, metavariables help with the problem of choosing the right substitutions for existentially quantified variables. In this case, the decision should be delayed until it is clear what would make the argument work [7]. It is difficult to use HOL in automated theorem provers because of undecidable algorithms and proof tactics [8].

CHAPTER 3

Lambda Calculus

Lambda calculus is a formalism developed by Alonzo Church in the 1930s, around the same time as Alan Turing developed another formalism on Turing machines. Turing proved in his thesis that the two systems are equivalent and define the same class of computable functions. Interestingly, uncomputable problems were first described in terms of lambda calculus when discovered [10].

Turing machines form the foundations of modern day computers with the von Neumann architecture (conceptually Turing machines with random access registers) as well as the imperative paradigm of programming languages, for example Fortran and C. Meanwhile, lambda calculus provides the theory for the functional paradigm, for example Lisp and Haskell [1]. Lambda calculus and the functional paradigm emphasize symbolic transformation rules [18] and thus have the potential to represent formal logic and mathematics in a way that is less concerned with the implementation details of the imperative paradigm.

In this chapter, we introduce definitions and axioms central to the foundation of lambda calculus.

3.1 Definitions

3.1.1 λ-Terms

Computable function, without referring to any model of computation, is an informal notion referring to a function where there exists an algorithm to take in that function's input and give the corresponding output [10]. Lambda calculus has "a single transformation rule (variable substitution, also called β-conversion) and a single function definition scheme" that allows any computable function to be expressed and evaluated [18]. Lambda calculus centers around the concept of *λ-term*, which is defined recursively as follows:

- <λ-term> := <name> | <function> | <application>
- <function> := λ<name> . <λ-term>
- <application> := <λ-term> <λ-term>

where *names* (also called *atoms*) are identifiers for some variable or constant, and functions (also called *abstractions*) are represented using the symbol λ to denote its arguments before the period (.) and its body after the (.) where it specifies how the given arguments should be modified. As usual, parentheses around an expression E can be used for better clarity without changing the meaning of that expression, i.e. E is equivalent to (E) or $E \equiv (E)$. Function application associates from the left, i.e. $E_1E_2E_3...E_n \equiv (...((E_1E_2)E_3)...E_n)$.

For example, the identity function is represented by $(\lambda x.x)$ where λx means the function's only argument is x, which is returned unchanged as x in the function's body.

More formally, *λ-terms* is defined as follows:

Definition 3.1 λ-terms [10]: *Assume that there is given an infinite sequence of variables, and a sequence of atomic constants different from the variables. (Note: When the latter* *sequence is empty, the system is considered **pure**, otherwise it is **applied**). The set of expressions called λ**-terms** is defined as follows:*

- *All **atoms** (variables and atomic constants) are λ-terms.*
- ***Abstractions** of the form $(\lambda x.M)$ are λ-terms where M is a λ-term and x is a variable.*
- ***Applications** of the form (MN) are λ-terms where M, N are also λ-terms.*

For convenience, λ-terms are often abbreviated to "terms" in the appropriate contexts.

3.1.2 Free and Bound Variables

Before defining how variables are considered *free* or *bound*, we need to define *occurrence* and *scope*.

Definition 3.2 Occurrence [10]: *For λ-terms P and Q, the relation P **occurs in** Q is defined as:*

- *P occurs in P (itself);*
- *if P occurs in M or $P \equiv x$, then P occurs in $\lambda x.M$;*
- *If P occurs in M or in N, then P occurs in MN.*

Definition 3.3 Scope [10]: *Consider $\lambda x.M$. M is called the **scope** of λx.*

Definition 3.4 Free and bound variables [10]: *An occurrence of a variable x in a term P is called*

- ***bound** if it is in the scope of a λx in P,*
- ***bound and binding** if and only if it is the x in λx,*
- ***free** otherwise.*

If x has at least one binding occurrence in P, then it is a bound variable of P. If x has at least one free occurrence in P, then it is a free variable of P.

A term is *closed* if it is a term without any free variables; otherwise it is an *open* term [12].

Many introductory materials on lambda calculus eliminate the distinction between *bound* and *bound and binding* variables to avoid confusion for the reader [1] [12] [18], since it often does not matter much. We highlight the difference with an example below.

Example 3.1. Consider $(\lambda x.2xy)$. Here, the first x next to the λ is bound and binding, the second x in the middle of $2xy$ is bound, and y is free. The concept of free and bound variables is analogous to multivariate integrals, for instance we evaluate the inner integral of $\iint 2xy\,dx\,dy = \int x^2y\,dy$, "ignoring" y while integrating x because x is bound to dx and y is free in regards to dx. Note that the second x is bound in the context of the whole expression $(\lambda x.2xy)$, but free in the context of just the function body $2xy$, which makes a difference when we move on to substitution in the next section.

3.1.3 Substitution

In this section we give a formal definition for substitution, then discuss two different types of substitution, or reduction, called α-reduction and β-reduction.

Formally, we define substitution as follows:

Definition 3.5 $FV(P)$ [1]: *The set of all free variables of P is written as $FV(P)$, defined* *inductively as follows:*

$$FV(x) = \{x\};$$
$$FV(PQ) = FV(P) \cup FV(Q);$$
$$FV(\lambda x.P) = FV(P) - \{x\}.$$

Definition 3.6 SUBSTITUTION [10]: *For any terms M, N, x, let $[N/x]M$ denote the result of substituting N for every free occurrence of x in M, while avoiding any name clashing with bound variables. Let $y \not\equiv x$ and $z \notin FV(NP)$ for any term P. Then $[N/x]M$ is defined as follows:*

- $[N/x]x \equiv N;$
- $[N/x]a \equiv a$ *for all atoms* $a \not\equiv x;$
- $[N/x](PQ) \equiv ([N/x]P\ [N/x]Q);$
- $[N/x](\lambda x.P) \equiv \lambda x.P;$
- $[N/x](\lambda y.P) \equiv \lambda y.P$ *if* $x \notin FV(P);$
- $[N/x](\lambda y.P) \equiv \lambda y.[N/x]P$ *if* $x \in FV(P)$ *and* $y \notin FV(N);$
- $[N/x](\lambda y.P) \equiv \lambda z.[N/x][z/y]P$ *if* $x \in FV(P)$ *and* $y \in FV(N).$

Example 3.2. Using Definition 3.6, consider the following examples:

- $[(uv)/x](\lambda y.x(\lambda w.vwx)) \rightarrow \lambda y.uv(\lambda w.vw(uv))$ since both occurrences of x in $(\lambda y.x(\lambda w.vwx))$ are free.
- $[(\lambda y.vy)/x](y(\lambda v.xv)) \rightarrow [(\lambda y.vy)/x](y(\lambda t.xt)) \rightarrow y(\lambda t.(\lambda y.vy)t)$ where $t \neq x, y, v$. We rewrite $(y(\lambda v.xv)) \equiv (y(\lambda t.xt))$ to avoid using the same v to represent two different variables in $(\lambda y.vy)$ and $(\lambda v.xv)$.

- $[(uv)/x](\lambda x.vy) \to (\lambda x.vy)$ since there is no free occurrence of x in $(\lambda x.vy)$.

With substitution defined, we can discuss reduction. Informally, *α-reduction* is simply rewriting a function's definition using different names for its arguments, but in the end, it is still the same function since the arguments' names are merely place holders [18].

Example 3.3. Consider the identity function in the previous section. We can write it in different but equivalent ways:

$$(\lambda x.x) \equiv (\lambda y.y) \equiv (\lambda z.z) \equiv (\lambda w.w).$$

More formally, α-reduction is denoted by $\lambda y.[y/x]M$ where $y \notin FV(M)$, and also called *change of bound variable* or *α-conversion* [10].

β-reduction is intuitively the replacement of the "place holders" arguments with real values using the rules of substitution in Definition 3.6. We use the notation $[N/x]E$ to indicate that all free occurrences of x in the term E (for example, a function body) are replaced by N [12] [18].

Example 3.4. By β-reduction, the application of the identity function $\lambda x.x$ to some a is $(\lambda x.x)a \to [a/x]x \to a$.

Example 3.5. Consider a more complicated example, $(\lambda x.x(\lambda x.x)w)y$. We notice that in the term $x(\lambda x.x)w$, we can apply α-reduction to rewrite it as $x(\lambda t.t)w$ for less confusion. Thus, the application of the function $(\lambda x.x(\lambda x.x)w)$ to y is $(\lambda x.x(\lambda x.x)w)y \equiv (\lambda x.x(\lambda t.t)w)y \to [y/x](x(\lambda t.t)w) \to y(\lambda t.t)w$.

Functions of multiple variables can be represented similarly in a way analogous to function composition. Consider $h(x, y) = x + y$. We can write h using the λ-notation as: $\lambda xy.x + y \equiv \lambda x.(\lambda y.x + y)$. Thus, for some $x = a$ and $y = b$, $(\lambda xy.x + y)ab = (\lambda x.(\lambda y.x + y))ab = (\lambda y.a + y)b = a + b = h(a, b)$.

Definition 3.7 Redex [10] [12]: *A reducible expression, or redex for short, is any term of the form*

$$(\lambda x.M)N$$

where M, N are λ-terms, and the corresponding term

$$[N/x]M$$

is called its contractum.

If a term P contains a redex$(\lambda x.M)N$ which is replaced by $[N/x]M$, resulting in P', we say P is contracted or reduced to P' and write $P \rightarrow P'$.

An expression with no redex is called a *β-normal form*, or *normal form* for short.

Example 3.6. (a) $\lambda x.xy$ is not a redex because it is not applied to any argument.

(b) $(\lambda x.xy)N$ is a redex because it can be contracted to Ny using β-reduction. Ny is considered to **be** in normal form since it does not contain any redex; while $(\lambda x.xy)N$ **has** a normal form.

(c) Consider $(\lambda x.(\lambda y.xyz)t)z$. There are two ways to contract this term:

- $(\lambda x.(\lambda y.xyz)t)v \rightarrow (\lambda y.vyz)t \rightarrow vtz.$
- $(\lambda x.(\lambda y.xyz)t)v \rightarrow (\lambda x.xtz)v \rightarrow vtz.$

As shown in Example 3.6, expressions with more than one redex can be reduced in different ways [12].

1. Normal-order reduction: Choose the left-most redex first.

2. Applicative-order reduction: Choose the right-most redex first.

This example also illustrates an important theorem that essentially points out that no matter what path is taken, the end result of reducing a term, i.e. a computation in lambda calculus, is the same [10].

Theorem 3.1 (Church-Rosser theorem [10]).
If $P \to M$ and $P \to N$, then there exists a term T such that $M \to T$ and $N \to T$.

3.1.4 Lambda Calculus as Formal Theory

All of the above definitions allow us to now introduce lambda calculus as formal theory [2].

1. The principal axiom scheme of lambda calculus is

$$(\lambda x.M)N = [N/x]M \text{ for all } \lambda\text{-terms } M, N.$$

2. Logical axioms and rules:

   Equality:

   - $M = M$;
   - $M = N \implies N = M$;
   - $M = N, N = L \implies M = L$;

   Compatibility rules:

   - $M = M' \implies MZ = M'Z$;
   - $M = M' \implies ZM = ZM'$;
   - $M = M' \implies \lambda x.M = \lambda x.M'$.

3. If $M = N$ is provable in lambda calculus, we write $\lambda \vdash M = N$.

3.2 Conditionals

In our natural language, conditionals are essentially "if ... then ... " rules. They require a way to represent the two values True and False, often referred to collectively as the Boolean type in computer science, which can be represented in lambda calculus as functions.

Definition 3.8 TRUE AND FALSE [1]: *The values True and False are defined using λ-terms as:*

- *True:* $T \equiv \lambda xy.x$
- *False:* $F \equiv \lambda xy.y$

Essentially, True is a function that takes two arguments and return the first one. False is a function that takes two arguments and return the second one. Recall Definition 3.1, which notes that a pure system does not have a set of atomic constants. That is why True and False are defined as functions in pure lambda calculus. The next sections illustrate how these definitions can help defining the following logical operations: AND, OR, and NOT.

3.2.1 AND

Definition 3.9 *AND* [18]: *The function AND of two arguments is defined as*

$$\wedge \equiv \lambda xy.xyF$$

Then for any a, b, $\wedge ab$ denotes "a AND b" where the value of $\wedge ab$ is determined as follows:

a	b	$\wedge ab$
T	T	T
T	F	F
F	T	F
F	F	F

Table 3.1: Truth table for *AND*.

Proof. We need to show that for any a, b that can be either $T \equiv \lambda xy.x$ or $F \equiv \lambda xy.y$, the results of applying $\wedge \equiv \lambda xy.xyF$ to a and b are as outlined in Table 3.1. Consider the following four cases:

1. $a \equiv b \equiv T \equiv \lambda xy.x$. We want to show that $\wedge ab \equiv T$ in this case. Using Definition 3.6 of substitution yields

   $\wedge ab \to (\lambda xy.xyF)(ab)$ (substituting $\wedge$ with its definition)

   $\to [b/y]([a/x](\lambda xy.xyF))$ (substituting x with a, y with b in the expression $\lambda xy.xyF$)

   $\to abF$

   $\to (\lambda xy.x)(bF)$ (substituting a with its definition)

   $\to [F/y]([b/x](\lambda xy.x))$ (substituting x with b, y with F in the expression $\lambda xy.x$)

   $\to b$

   $\equiv T$

   as desired.

2. $a \equiv T \equiv \lambda xy.x$, and $b \equiv F \equiv \lambda xy.y$. We want to show that $\wedge ab \equiv F$ in this case. Using Definition 3.6 of substitution yields

   $\wedge ab \to (\lambda xy.xyF)(ab)$

   $\to abF$ (by substituting x with a, y with b in the expression $\lambda xy.xyF$)

   $\to (\lambda xy.x)(bF)$ (substituting a with its definition)

   $\to [F/y]([b/x](\lambda xy.x))$ (substituting x with b, y with F in the expression $\lambda xy.x$)

   $\to b$

   $\equiv F$

   as desired.

3. $a \equiv F \equiv \lambda xy.y$, and $b \equiv T \equiv \lambda xy.x$. We want to show that $\wedge ab \equiv F$ in this case. Using Definition 3.6 of substitution yields

$\wedge ab \to (\lambda xy.xyF)(ab)$

$\to abF$ (by substituting x with a, y with b in the expression $\lambda xy.xyF$)

$\to (\lambda xy.y)(bF)$ (substituting a with its definition)

$\to [F/y]([b/x](\lambda xy.y))$ (substituting x with b, y with F in the expression $\lambda xy.x$)

$\to [F/y](\lambda xy.y)$

$\equiv F$

as desired.

4. $a \equiv F \equiv \lambda xy.y$, and $b \equiv F \equiv \lambda xy.y$. We want to show that $\wedge ab \equiv F$ in this case. Using Definition 3.6 of substitution yields

$\wedge ab \to (\lambda xy.xyF)(ab)$

$\to abF$ (by substituting x with a, y with b in the expression $\lambda xy.xyF$)

$\to (\lambda xy.y)(bF)$

$\to [F/y]([b/x](\lambda xy.y))$ (substituting x with b, y with F in the expression $\lambda xy.x$)

$\to [F/y](\lambda xy.y)$

$\equiv F$

as desired.

Therefore, $\wedge \equiv \lambda xy.xyF$. □

3.2.2 OR

Similarly, we define *OR* as follows.

Definition 3.10 *OR* [18]: *The function OR of two arguments is defined as*

$$\lor \equiv \lambda xy.xTy$$

Then for any a, b, $\lor ab$ denotes "a OR b" where the value of $\lor ab$ is determined as follows:

a	b	$\lor ab$
T	T	T
T	F	T
F	T	T
F	F	F

Table 3.2: Truth table for *OR*.

The proof is very similar to the proof for *AND*, thus we will not elaborate here. In short, if a is equivalent to True, then it picks the first argument from (Tb), which is T. If a is equivalent to False, then it picks the second argument from (Tb), which is b, meaning the value of the whole expression $\lor ab$ now depends on the value of b.

3.2.3 NOT

Negation can be defined as a function of one argument.

Definition 3.11 *NOT* [18]: *The function NOT of one argument is defined as*

$$\sim \equiv \lambda x.xFT$$

Proof. We want to show that $\sim T \equiv F$ and $\sim F \equiv T$. We can see that the negation of True is

$$\begin{aligned}
\sim T &\rightarrow (\lambda x.xFT)T \\
&\rightarrow T(FT) \text{ (substituting } x \text{ with } T \text{ in the expression } \lambda xy.xFT \text{)} \\
&\rightarrow (\lambda xy.x)(FT) \text{ (substituting } T \text{ with its definition)} \\
&\rightarrow F.
\end{aligned}$$

Similarly, the negation of False is

$$\begin{aligned}
\sim F &\rightarrow (\lambda x.xFT)F \\
&\rightarrow F(FT) \text{ (substituting } x \text{ with } F \text{ in the expression } \lambda xy.xFT \text{)} \\
&\rightarrow (\lambda xy.y)(FT) \text{ (substituting } F \text{ with its definition)} \\
&\rightarrow T.
\end{aligned}$$

□

3.3 Recursion

Consider the expression $(\lambda x.xx)(\lambda x.xx)$. Reducing this expression does not result in a normal form, since

$$[(\lambda x.xx)/x](\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow [(\lambda x.xx)/x](\lambda x.xx) \rightarrow \ldots$$

The idea of recursion is similar. Recursion is defined as a special function called the *Y combinator*, or *fixed point operator* [12] [18]:

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Applying Y to a function G yields

$$\begin{aligned} YG &\to [G/f](\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))) \\ &\to (\lambda x.G(xx))(\lambda x.G(xx)) \\ &\to G((\lambda x.G(xx))(\lambda x.G(xx))) \\ &\to G(YG) \end{aligned}$$

Therefore, unless G has a base case to terminate, YG goes on forever in an infinite loop.

3.4 Functional Programming

As we can see in the previous sections of this chapter, lambda calculus can indeed provide the basic elements of programming. While Church's lambda calculus is equivalent to Turing machines, in the early days of computing the first electronic devices' hardware was more closely designed as Turing machines with random access memory. This led to many implementation difficulties regarding memory usage, compilation, and run time for functional programming. Nowadays, however, these problems are not so much of a burden due to better hardware and continuous optimization [1].

Since an expression M in pure lambda-calculus can be reduced in different ways, during implementation functional programming languages seek to find the most optimized way in terms of time and space efficiency to evaluate expressions. Haskell, for example, chooses the left-most redex first (as in normal-order reduction), but only if it is not contained within the body of a lambda abstraction [12]. This is also called *lazy evaluation*, a characteristic of "lazy" functional languages that aim to increase efficiency by reducing a big redex into a potentially smaller normal form in a term before trying to evaluate the whole term. Languages such as Clean and Haskell are state-of-the-art in the functional paradigm, and allow development of many important applications such as AT&T's Network Security division and some internal tools at Google [1].

CHAPTER 4

Software Architecture

In this chapter we discuss three main components of `robotone`: the logic framework, the mathematical contents, and the LaTeX writeup. The logic framework and LaTeX writeup are mainly the work of the original authors, Ganesalingam and Gowers [7], with some improvements regarding fixed bugs and clearer source code, while the mathematical contents for Real Analysis are developed in this project as a new customization of `robotone` for The College of Wooster's MATH-332 Real Analysis I course.

Figure 4.1 gives a general overview of how `robotone` works. The user supplies the mathematical contents (problem statements and mathematical facts) as part of the source code. Then `robotone` parses each problem to form appropriate first-order logic objects, and starts the proof engine by sequentially applying tactics in a fixed order by priority until the problem is either solved, or no more tactics can be used to move forward. All steps taken in a proof are logged and written in TeX format into a `.tex` file, together with the original problem statement and its complete proof (or a partial one if `robotone` fails to solve it). The resulting `.tex` file can be compiled separately with an available TeX distribution.

Figure 4.1 may make the components of `robotone` look as if they worked separately, but in reality their implementations are woven together in certain ways to optimize runtime and source code simplicity.

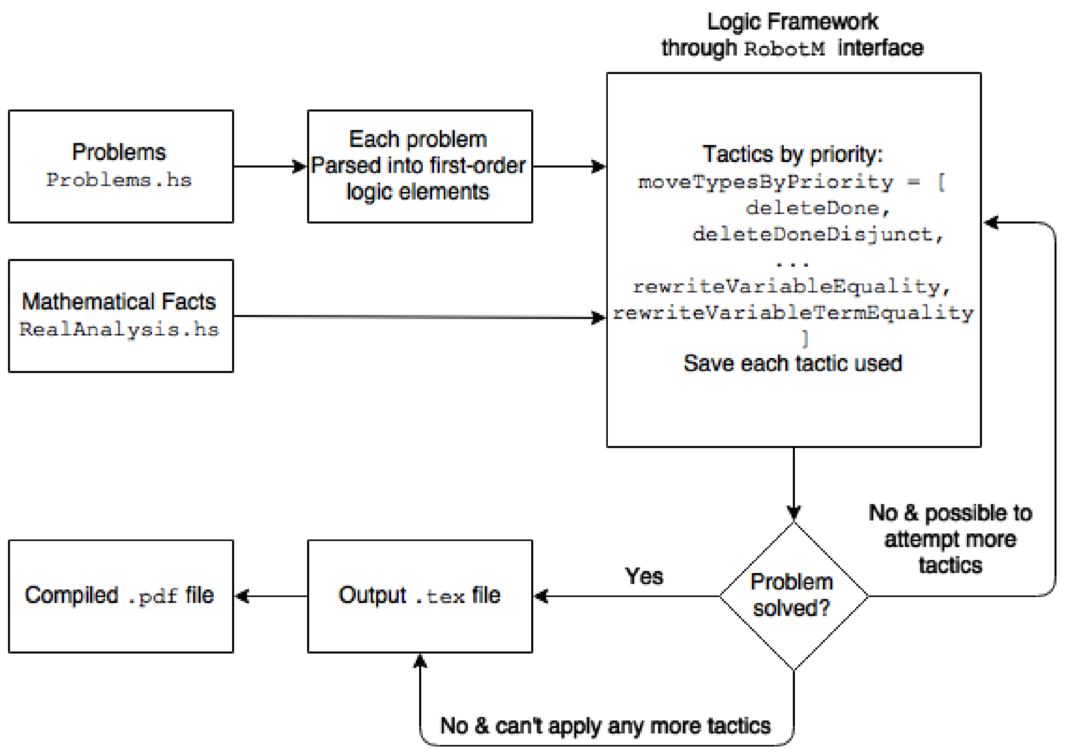


Figure 4.1: Overview of `robotone's` work flow.

4.1 Logic Framework

The program `robotone's` logic framework consists of definitions of the elements in first-order logic and a collection of tactics to manipulate mathematical objects during proofs, as well as an overall "controller" called `RobotM` that keeps track of the proof's progress so far.

4.1.1 Implementing First-Order Logic and Tactics

The theoretical foundation for `robotone's` logic framework is based on first-order logic as laid out in Chapter 2, and the collection of tactics are described in details below.

Using Haskell as the main programming language for `robotone` allows the implementation of types to follow the formal definitions as closely as possible. Of course, some very minor modifications have to be made to the formal definitions during implementation, but this does not affect the overall logic. For example, consider the almost-direct translation from the formal definition of *formula* in first-order logic to the actual definition in the source code in `Types.hs` below.

Example 4.1. Formal definition: A formula is defined inductively by the following rules:

- If A is an atom, then A is a formula.
- If A is a formula, then $\neg A$ is a formula.
- If A, B are formulas, then $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, and $(A \equiv B)$ are also formulas.
- If A is a formula and x is a variable, then $(\forall x)A$ and $(\exists x)A$ are formulas.

In the source code, this definition is interpreted into Haskell as:

```
data Formula = AtomicFormula Predicate [Term]
             | Not Formula
             | And [Formula]
             | Or [Formula]
             | Forall [Variable] Formula
             | Exists [Variable] Formula
             | UniversalImplies [Variable] [Formula] Formula
deriving (Eq, Ord, Show)
```

The only small difference is that the `UniversalImplies` relationship and other helper types indirectly cover $(A \supset B)$ and $(A \equiv B)$ in the formal definition. Everything else comes directly from the formal definition of formula.

Earlier in Chapter 1, inference rules are mentioned as the way new facts or conclusions are derived from given ones. In the implementation, we call them tactics, which are actions that make inferences using the user-provided definitions and theorems in an attempt to prove a given problem. The tactics in `robotone` are designed to mimic the way human mathematicians think, to a certain extent [7]. The bulk of the logic framework lies in the implementation of tactics, which are essentially how proofs progress forward. Tactics are arranged by their priority in

`Main.hs` so that they can be tried out in that order for each step in a proof. Figure 4.2 illustrates how the implementation of tactics is organized in the source code.

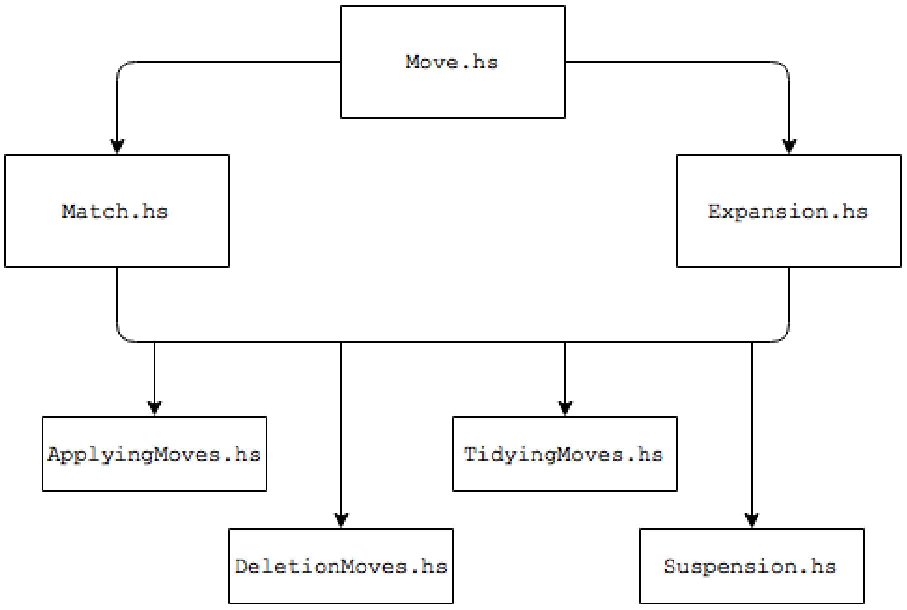


Figure 4.2: The hierarchy of tactics' source code. For the purpose of a well-organized software architecture, tactics are implemented such that the lower level (`ApplyingMoves.hs, DeletionMoves.hs, TidyingMoves.hs, Suspension.hs`) inherits general definitions and methods from the ones above it.

`Match.hs` and `Expansion.hs` contain the necessary manipulations of mathematical objects that all four subcategories of tactics use. Each subcategory's name reflects its tactics' general purpose, as described in details below coming from the original paper of Ganesalingam and Gowers [7].

4.1.2 Deletion/Removal Tactics (`DeletionMoves.hs`)

- deleteDone: if a previous tactic causes a target to be proven and replaced by $\top$, this one removes the target. The program aims to reach a goal with no targets left.

- deleteDoneDisjunct: If a target is disjunctive (in the form "A OR B") and either A or B is $\top$, then that target is automatically true and replaced by $\top$.

- deleteDangling: If a previously used assumption contains a variable that is not involved in any other statements, remove that assumption.

- deleteUnmatchable: If a previously used assumption has no obvious "match" with another unused assumption, remove the used assumption. For example, suppose we have 2 assumptions $x \in A$ and $A \subset B$. $A \subset B$ can be expanded to be $\forall u(u \in A \implies u \in B)$, thus the premise $u \in A$ can be "matched" with the existing assumption $x \in A$.

4.1.3 Tidying Tactics (TidyingMoves.hs)

- peelAndSplitUniversalConditionalTarget: If a target has the form $\forall x(P(x) \implies Q(x))$, then add a variable x, a new assumption $P(x)$ and a new target $Q(x)$. This is similar to how humans would think, "Let x be such that $P(x)$ is true. We want to show $Q(x)$ is true." The new assumption $P(x)$ has to be limited to only be used to prove $Q(x)$.

- splitDisjunctiveHypothesis: This tactic is similar to human splitting a proof into different cases. The paper's authors say that this is a work in progress.

- splitDisjunctiveTarget: If a target is of the form $P \vee Q$, then replace it with boxes of P and Q.

- peelBareUniversalTarget: If the target has the form $\forall x P(x)$ and P is not a conditional statement, then this tactic removes the universal quantifier, which resembles a human mathematician choosing an arbitrary element and proving that P is true for that arbitrary element.

- removeTarget: This tactic removes a target when there is an assumption equal to the target.

- collapseSubtableauTarget: If a target is in the form of a box that has no assumptions and contains no metavariables (i.e. no variable that we need to find or of which to prove the existence), then that target is replaced by the targets in the box.

4.1.4 Applying Tactics (ApplyingMoves.hs)

- forwardsReasoning: If there are two existing assumptions of the forms $\forall u(P(u) \implies Q(u))$ and $P(x)$, then we can obtain the new assumption $Q(x)$. For example, if we have $x \in A$ and $A \subset B$ (which is equivalent to $\forall u(u \in A \implies u \in B)$), we can deduce that $x \in B$.

- forwardsLibraryReasoning: This tactic is similar to the tactic forwardsReasoning above but receives a lower priority, because it requires using a fact stored in the library. Ganesalingam and Gowers speculate that human mathematicians tend to use facts not given explicitly in the problem only after they exhaust all the given assumptions. Moreover, this tactic may result in irrelevant deductions and unused terms, cluttering the final proof, and thus is limited to be used only when not creating any new terms at all. This is one of the tactics noted by Ganesalingam and Gowers to require more study into human mathematicians' behavior and thinking.

- expandPreExistentialHypothesis: This tactic replaces an assumption with its definition which contains existential quantifiers (thus the assumption itself is pre-existential). For example, the assumption "A is bounded" can be replaced with "$\exists M$ such that $\forall x(x \in A \implies |x| \leq M)$".

- elementaryExpansionOfHypothesis: This tactic replaces an assumption with its elementary expansion, which means the expansion does not begin with a quantifier. For example, the assumption “$x \in A \cap B$” can be replaced with two assumptions “$x \in A$” and “$x \in B$”.

- backwardsReasoning: If there is a target $Q(x)$ and an assumption $\forall u(P(u) \implies Q(u))$, then this tactic replaces the target with $P(x)$ (which, by the given assumption, would imply $Q(x)$ as desired). If $P(x)$ is a conjunction of several statements, then the program is limited to apply this tactic only when all but one of these statements are given assumptions, thus creating only one new target to avoid combinatorial growth of targets.

- backwardsLibraryReasoning: This tactic is similar to backwardsReasoning above except that $\forall u(P(u) \implies Q(u))$ is a fact in the library and not a given assumption. It is subjected to the same rule above to avoid adding too many new targets.

- elementaryExpansionOfTarget: This tactic replaces a target with its elementary expansion, which means the expansion does not begin with a quantifier. It is similar to elementaryExpansionOfHypothesis.

- expandPreUniversalTarget: This tactic replaces a pre-universal target (i.e. containing the quantifier $\forall$) by its expansion. This is analogous to expandPreExistentialHypothesis.

- solveBullets: This tactic is used to solve metavariables. It is similar to a human mathematician choosing a specific value or expression for a variable to complete the proof, such as the step of “Choose δ such that...” in an $\epsilon - \delta$ proof.

- automaticRewrite: As its name suggests, this rewrites terms in ways that a human would, to increase the similarity of the final proof to that of a human.

4.1.5 Suspension Tactics (Suspension.hs)

- unlockExistentialUniversalConditionalTarget: If there is a target involving both the universal (∀) and existential (∃) quantifiers, this tactic creates a metavariable, denoted with a ♦ next to the metavariable's name, for example $x♦$ which means x is a metavariable. The role of a metavariable translates to a statement along the line of, "We would like to find x such that ...".

- unlockExistentialTarget: If a target is in the form $\exists x P(x)$, this tactic replaces it with a box containing the variable $x♦$, no assumption, and a single target $P(x♦)$.

- expandPreExistentialTarget: This tactic replaces a pre-existential target (i.e. containing the quantifier ∃) by its expansion, similar to some of the above tactics.

- convertDiamondToBullet: This tactic resembles the human mathematician writing "Assume that..." when dealing with a metavariable. It helps handling the task of keeping track what other variables the metavariable is allowed or prohibited to depend on.

- rewriteVariableVariableEquality: If there is an assumption in the form $x = y$, then this tactic replaces all occurences of y by x and eliminates the original assumption.

- rewriteVariableTermEquality: If there is an assumption of the form $v = t$ or $t = v$ where v is a variable and t is a term, then this tactic replaces all occurences of t by v.

4.1.6 `RobotM`

`RobotM` is the interface through which the entry point to the program communicates with the logic framework. When the program starts from `Main.hs`, a `RobotM` instance is created to pull in the problem statements in `Problem.hs` and the user-provided mathematical contents in `RealAnalysis.hs` as inputs, keep track of tactics used so far for each problem in the order of their appearances, and save all of the active variables, terms, and formulas in the proof. Having `RobotM` helps abstract away the complex mechanisms of tactics, and provide a relatively smooth integration of the different parts of `robotone`.

4.2 Mathematical Contents

`Robotone` uses the problem statements and mathematical facts provided by the user as part of source code itself. This is an advantage coming from using Haskell, a pure functional programming language, because code can be treated as data. On the other hand, this does require recompilation of all the source code every time `robotone` runs, but the compilation time is negligible since Haskell is an extremely fast language. Additionally, if the user is unfamiliar with the language, it can be a rather steep learning curve.

In an attempt to make `robotone` more approachable than other automated theorem proving systems, the problems and mathematical facts are represented using human-like notations as string patterns (described in details in Sections 4.2.1 and 4.2.2). A parser is thus implemented as the bridge between the logic framework and the user-supplied mathematical contents.

4.2.1 Inputting Problems into Robotone

All problems are declared in the file `Problems.hs`. The format of each problem is of the form

```
Problem String [String] String
```

where

- The leftmost `String`: The problem statement in natural language in LaTeX format for the purpose of writing up the proof later.
- `[String]`: The list of given premises specific to the problem.
- The rightmost `String`: The goal of the problem, i.e. what we want to prove from the given premises.

For example, consider the problem definition in `robotone` for the statement, "If g, f are injections then $g \circ f$ is an injection."

```
ifGandFareInjectionsThenGoFisInjection = Problem --New
  "If $g,f$ are injections then $(g \\circ f)$ is an injection."
  ["injection(f)",
   "injection(g)"]
  "injection(compose(g,f))"
```

The first line is the declaration of `ifGandFareInjectionsThenGoFisInjection` as a `Problem` variable. Then the next three lines initialize this variable with a problem statement, the list of given premises (that both g and f are injections), and the goal of proving that the composition of g and f is also an injection.

4.2.2 Inputting Mathematical Facts into Robotone

Mathematical facts consists of definitions and theorems. The definition for a concept is of the form `(String, String)` where the first `String` is the name or form of the concept, and the second `String` is the definition. For example, consider the definition of injection:

```
 ("injection(f)", "forall x y z.(equals(applyfn(f,x),z) & equals(
  applyfn(f,y),z) => equals(x,y))")
```

This translates to natural mathematics language as, “f is an injection iff for all $x, y, z, f(x) = f(y)$ implies $x = y$.”

A theorem, whose type name is `Result` in the source code, is of the form

```
Result String [Formula] Formula
```

where

- `String`: A brief description or the name of the theorem if it has one. This is optional and an empty string can suffice.
- `[Formula]`: A list of premises.
- `Formula`: The conclusion of the given premises above.

For example, consider the Monotone Convergence Theorem in `RealAnalysis.hs`:

```
    Result "Monotone Convergence Theorem" [
      parse formula "bounded(an)",
      parse formula "monotone(an)"]
      (parse formula "converges(an)")
```

This translates to natural mathematics language as, “A sequence a_n that is bounded and monotone converges.” The syntax `parse formula` does exactly what it looks like, parsing the given `String` into a `Formula`.

4.3 LaTeX Writeup

The main files containing the mechanisms for generating the `.tex` file are `Tex.hs`, `TexBase.hs`, and `Writeup.hs`. These files contain the general formatting configurations such as header, footer, and the overall presentation of a proof. However, to optimize for better efficiency, most other source files also contain appropriate LaTeX patterns for the definitions, concepts, and tactics so that the output `.tex` file can be generated while the process of solving a problem is happening.

CHAPTER 5

USING `ROBOTONE` FOR REAL ANALYSIS I

This chapter discusses changes made to the original source code, how to set up `robotone`, the work flow of the graphical user interface, and the analysis of seven successful proofs and two incomplete proofs. We find that using Docker is currently the most viable way to make `robotone's` installation and usage easier for users. The proof writing capability of the software is still rather limited, but the number of working proofs is growing as we address different factors that cause incomplete proofs. A major factor is that often human mathematicians know certain basic mathematical facts after years of studying, which `robotone` could not have known without those facts being explicitly programmed into the library.

5.1 MODIFICATIONS TO THE ORIGINAL SOURCE CODE

5.1.1 CUSTOMIZATIONS FOR MATH-332 REAL ANALYSIS I

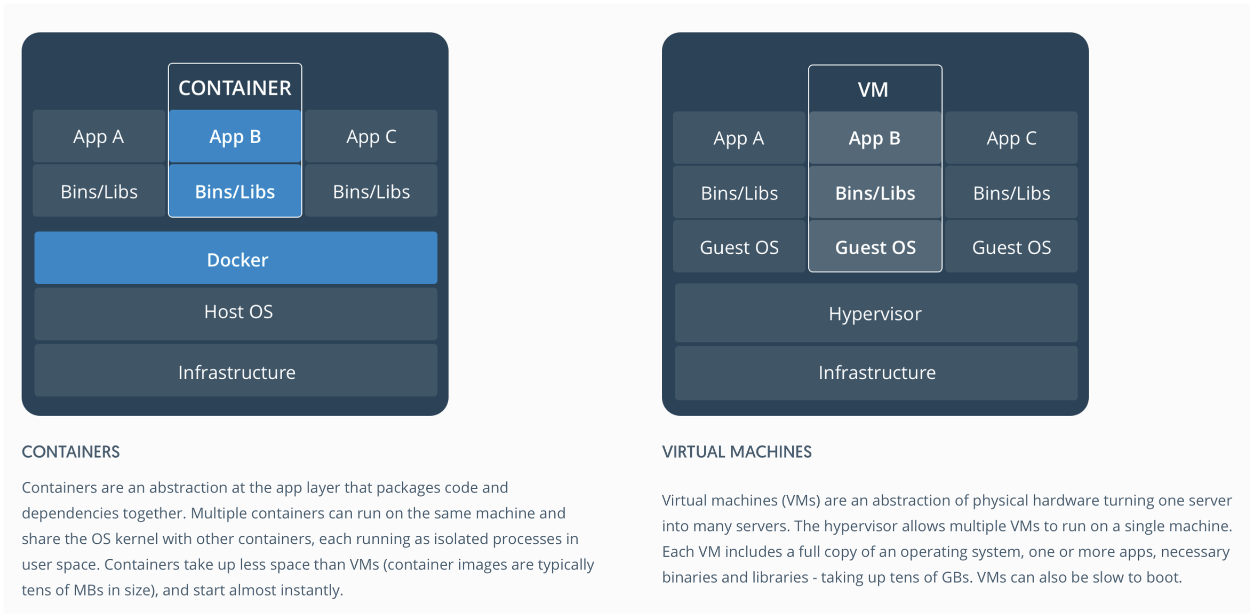
Originally, problems are in the same files as the definitions and theorems for metric spaces, which make up the problem domain from which the original authors, Ganesalingam and Gowers, chose to include examples. Since the set of real numbers $\mathbb{R}$ is a metric space, most of the existing definitions and theorems apply for Real Analysis. We add ten new definitions and three new theorems in the file `src/RealAnalysis.hs`, as well as seven new working problems in the file `src/Problems.hs`. A full list of these are in the Appendices, Chapter A. There are also nine new problems that could not be solved by `robotone`, which are analyzed in a later section of this chapter. Currently, the total number of working proofs is eighteen (and counting), seven of which are new.

5.1.2 Docker

The original source code is meant to be run without any virtualization. Users are left on their own to figure out how to configure their machines to fit the requirements of `robotone`. To compile successfully, the software requires ghc 7.8.3 and Cabal 1.18.0.5, while the latest version of ghc at the time of writing is 8.2.1 and of Cabal is 1.24.2.0, thus no longer compatible with `robotone` and causing build errors. On the other hand, reverting back to using the older version of Haskell (7.8.3 as `robotone` needs) can cause errors in other Haskell software that the users may have on the same machine. Therefore, we decide to integrate the source code with Docker, a software container platform [5].

Docker aims to solve the problem of different development environments on the same or different computers without having to use virtual machines, a more resource-consuming intermediary. An improvement in terms of usability made to Ganesalingam's and Gowers' original software is porting the code into a Docker *image* and thus running it from a Docker *container*. We can think of a Docker image as a static snapshot of a computer or software, and of a corresponding Docker container as an instance of that snapshot (either running or stopped). This means the user only needs to install Docker and then run our provided scripts in the software package to use `robotone` without having to worry about any specific dependencies.

The Docker image associated with `robotone` is configured in the Dockerfile inside the root directory of the software. It is fairly simple and meant to provide the appropriate Haskell environment configurations for `robotone`. Users can easily modify this Dockerfile to include additional Haskell packages and dependencies needed in case they continue to develop `robotone` based on the current source code.



CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Figure 5.1: Comparing Docker containers and virtual machines [5].

Figure 5.1 illustrates the differences between a Docker container and a virtual machine. At the beginning of the project, we install `robotone` inside a virtual machine running the relatively minimalist Linux distribution Debian inside VirtualBox. However, running `robotone` on a virtual machine has many disadvantages compared to Docker and other software container platforms:

- Virtual machines can be slow to boot, since hypervisors require a lot more resources from the host machine in order to emulate the full operating systems and possibly unneeded pre-installed software inside virtual machines. Meanwhile, a running Docker container does not necessarily need the full operating systems, and users can pick and choose what they want to include in the container by specifying their own requirements in the Dockerfile.

- There is a limit on how many running virtual machines can be up at the same time, because they use up a substantial amount (at least 512MB, often 1GB) of RAM on the host machine to maintain a smooth experience inside the virtual operating systems. Most hypervisors also advise users not to allocate more than half of their host machine's RAM to a virtual machine. A running Docker container is much more lightweight, allowing users to run several different containers simultaneously.

- Moving files between virtual machines and host is dependent on the hypervisor, and often has to be done manually. In the case of `robotone`, it means having to copy the `.tex` and `.pdf` files to the host machine every time the program is run. With Docker, a single command in the terminal can copy files from the running container to the host machine, or even write the output directly at the desired directory on the host machine, which can be automated with a bash script as done inside the helper scripts listed in the next section.

5.1.3 Helper Scripts

Using Docker and trying to develop a graphical UI creates the need for simple scripts. The entry point to the program `Main.hs` is rewritten for better modularity, which also helps with the task of writing scripts for a simpler user experience. Currently there are 3 main scripts in the root directory of `robotone`:

1. The script `runfromsource.sh` is for running `robotone` directly without any kind of virtualization as outlined in Section 5.2.3.

2. The script `rundocker.sh` is for running with Docker as outlined in Section 5.2.1. For Docker, users can now specify the path to their desired output file when using the script `rundocker.sh` instead of having the results written inside the `build/` directory by default. It handles the interaction between

the host machine and the Docker container during the process of running `robotone` and producing the proofs without any extra efforts from the user.

3. The script `run.sh` is a helper script that gets run inside a Docker `exec` call for `robotone`'s running Docker container to generate the `.tex` file containing proofs.

All unneeded scripts are saved in the directory `archived_scripts` for reference.

5.2 Setting Up Robotone

There are two ways to set up `robotone` for use, either by building directly from the source code, or through Docker. The former method requires the user to install very specific versions of `robotone's` dependencies or else the build fails, therefore making it very limiting to the user and potentially interfering with the user's existing Haskell programming environment. Thus the latter method is an added improvement to the original source code, in order to make this software more usable and portable.

5.2.1 Installation and Usage with Docker on Command Line Interface

The instructions in this section are tested on macOS Sierra 10.12.6, but should also work for Unix-based systems with very minor modifications if any at all. The Docker image only contains the appropriate environment for running `robotone` and not the software itself, meaning the users can freely modify the source code, the library, and the list of problems without having to rebuild the Docker image. After setting up to run `robotone` the first time, users only need to run the `src/Problems.hs` as detailed below.

1. Make sure that a TeX distribution is available to compile the generated proofs using the `xelatex` command in the terminal.

2. To install Docker, follow the instructions from [5]. After Docker is confirmed to be installed and working properly, proceed to the next steps.

3. Open the terminal. Navigate to the `robotone` root directory with:

```
$ cd /path/to/robotone/directory
```

4. If this is NOT the first time you use `robotone`, skip this step. Otherwise:

   (a) Build the Docker image with the provided Dockerfile with:

   ```
   $ docker build -t robotone .
   ```

   (b) Then create the Docker container named `cont_robotone` with:

   ```
   $ docker run -v $(pwd):/root-robotone --name cont_robotone -it
       robotone /bin/bash &
   ```

   (c) Start the container created above with:

   ```
   $ docker container start cont_robotone
   ```

   If you encounter errors of the form, "Error response from daemon: Container [...] is not running", use the above command.

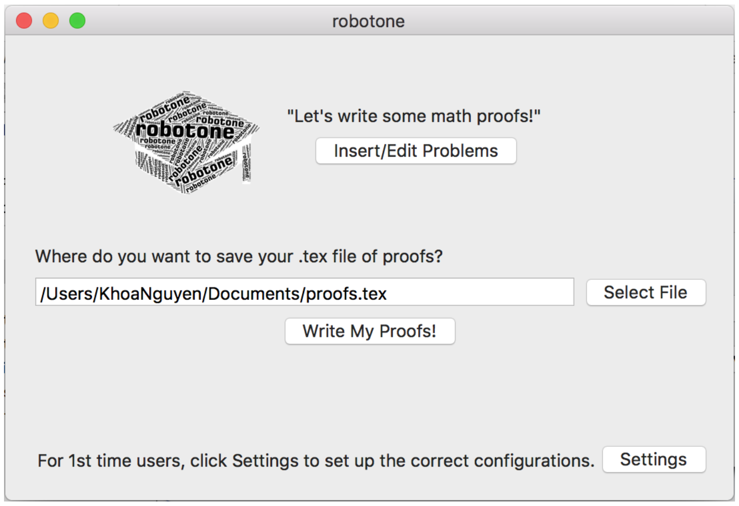
5. Now, whenever you update the problems in `src/Problems.hs` or the theorems and definitions in `src/RealAnalysis.hs`, run the script `rundocker.sh` with:

```
$ bash rundocker.sh <path/to/desired/output/file.tex>
```

from the `robotone` root directory, and retrieve the generated proofs in both `.tex` and `.pdf` files where expected.

5.2.2 Installation and Usage with Docker on Graphical Interface

A graphical user interface (GUI) is implemented in about 300 lines of Swift as a Cocoa application for MacOS. The main window of `robotone` is shown in Figure 5.2.


robotone

"Let's write some math proofs!"

Insert/Edit Problems

Where do you want to save your .tex file of proofs?

/Users/KhoaNguyen/Documents/proofs.tex

Select File

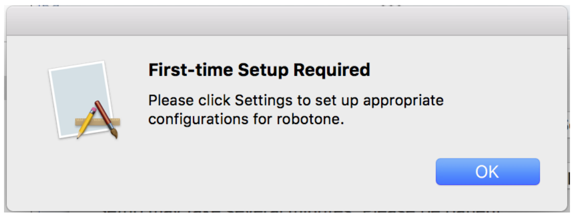
Write My Proofs!

For 1st time users, click Settings to set up the correct configurations.

Settings

Figure 5.2: The main window. From here, the user can change settings, insert or edit problems, and run the proof engine.

For first-time users, a pop-up message as illustrated by Figure 5.3 will appear before the main window to remind them to configure proper settings before attempting to run the proof engine. This pop-up message only appears once the first time; subsequent uses of `robotone` should not have it.


First-time Setup Required

Please click Settings to set up appropriate configurations for robotone.

OK

Figure 5.3: Reminder for first-time users to configure proper settings.

From the main window, clicking the button **Settings** opens up the Settings window as illustrated in Figure 5.4 where the user can specify where the root directory containing all `robotone`-related files is. Without the right path to this directory, `robotone` cannot perform the proof writing capability. Also, first-time users would need to click the button **Yes, run setup now** to build the Docker image and start the corresponding Docker container to enable the proof engine. This first-time setup is required and can take several minutes, but will only need to be done once.

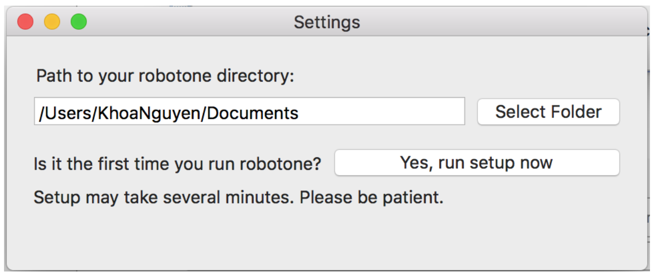

Settings
Path to your robotone directory:
/Users/KhoaNguyen/Documents
Select Folder
Is it the first time you run robotone?
Yes, run setup now
Setup may take several minutes. Please be patient.

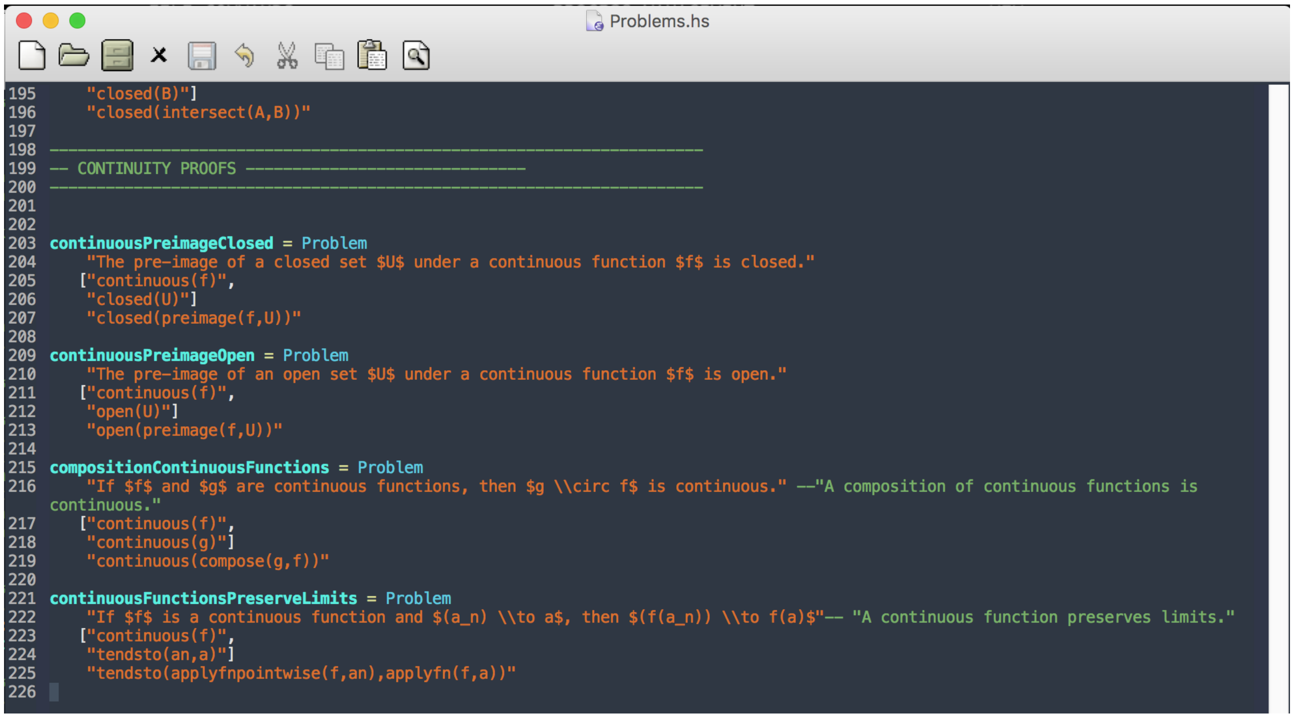
Figure 5.4: The Settings window. Here, the user can input the correct path to the `robotone` directory and run the first-time setup as needed.

Once setup is done, the user can close Settings and come back to the main window. Now, they can click **Insert/Edit Problems** to open `Problem.hs` in their default editor as shown in Figure 5.5.

The user can specify where they want to save the output files by clicking **Select File**. This opens up a file selection window as shown in Figure 5.6. The default place to save the output files is the user's Documents folder.

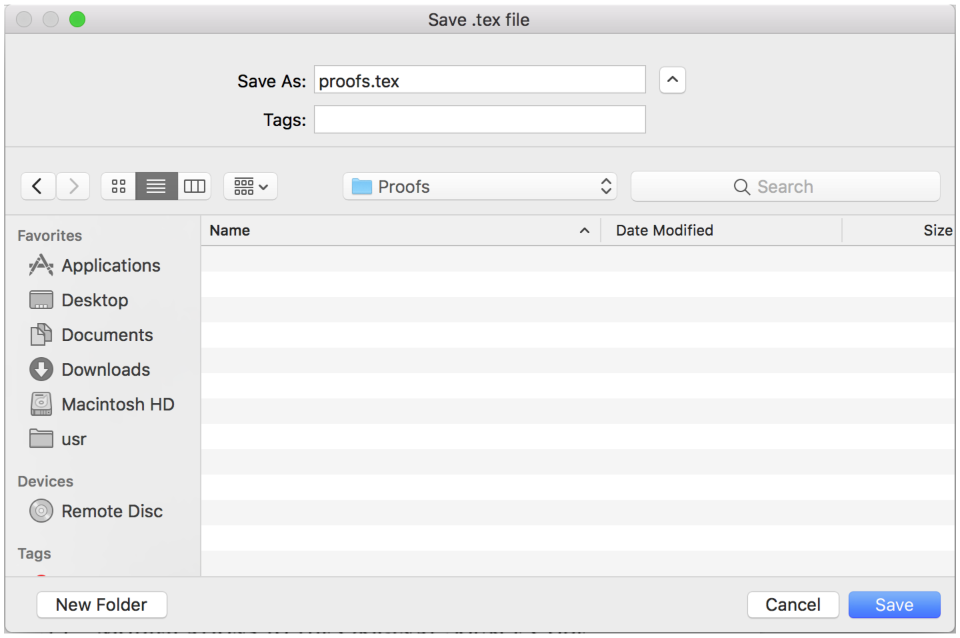
Back to the main window, clicking **Write My Proofs!** starts the proof engine. If everything is set up properly, the resulting `.pdf` file containing all the written proofs will be displayed as illustrated in Figure 5.7. Users can find both the `.tex` and `.pdf` files at the location they have selected previously.

If the settings are not configured properly, running the proof engine results in an error message as shown in Figure 5.8. If this error message pops up, users are advised to check their Settings and/or `Problem.hs` again for any possible mistakes.



```
    "closed(B)"]
    "closed(intersect(A,B))"

-----------------------------------------------------------------------
-- CONTINUITY PROOFS ------------------------------
-----------------------------------------------------------------------


continuousPreimageClosed = Problem
    "The pre-image of a closed set $U$ under a continuous function $f$ is closed."
   ["continuous(f)",
    "closed(U)"]
    "closed(preimage(f,U))"

continuousPreimageOpen = Problem
    "The pre-image of an open set $U$ under a continuous function $f$ is open."
   ["continuous(f)",
    "open(U)"]
    "open(preimage(f,U))"

compositionContinuousFunctions = Problem
    "If $f$ and $g$ are continuous functions, then $g \\circ f$ is continuous." --"A composition of continuous functions is continuous."
   ["continuous(f)",
    "continuous(g)"]
    "continuous(compose(g,f))"

continuousFunctionsPreserveLimits = Problem
    "If $f$ is a continuous function and $(a_n) \\to a$, then $(f(a_n)) \\to f(a)$"-- "A continuous function preserves limits."
   ["continuous(f)",
    "tendsto(an,a)"]
    "tendsto(applyfnpointwise(f,an),applyfn(f,a))"
```

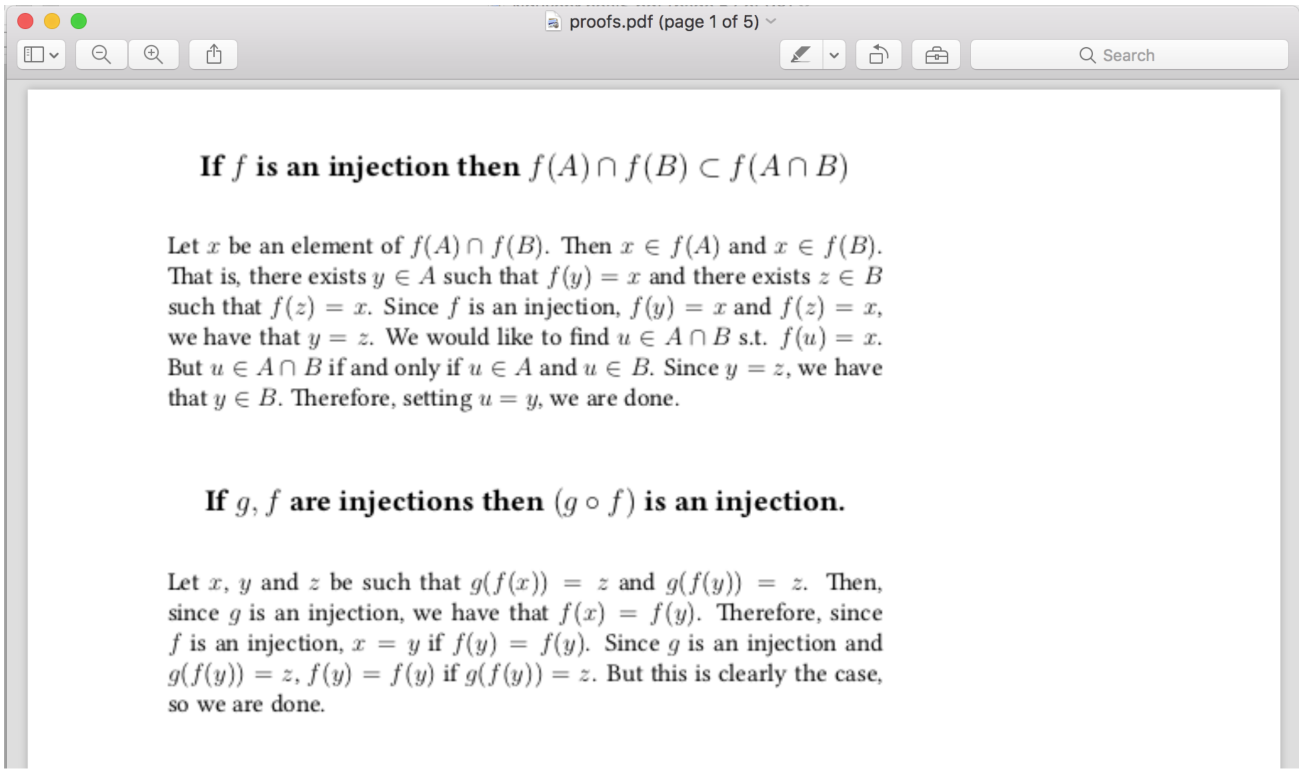
Figure 5.5: Opening `Problems.hs` for editing.


Save .tex file

Save As: proofs.tex

Tags:

Figure 5.6: Choosing where to save the output `.tex` file.



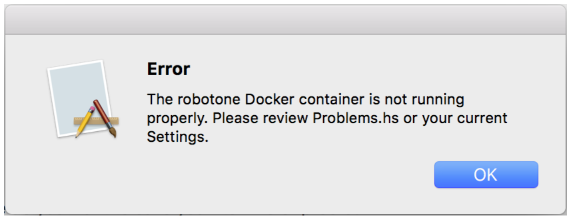
If f is an injection then $f(A) \cap f(B) \subset f(A \cap B)$

Let x be an element of $f(A) \cap f(B)$. Then $x \in f(A)$ and $x \in f(B)$. That is, there exists $y \in A$ such that $f(y) = x$ and there exists $z \in B$ such that $f(z) = x$. Since f is an injection, $f(y) = x$ and $f(z) = x$, we have that $y = z$. We would like to find $u \in A \cap B$ s.t. $f(u) = x$. But $u \in A \cap B$ if and only if $u \in A$ and $u \in B$. Since $y = z$, we have that $y \in B$. Therefore, setting $u = y$, we are done.

If g, f are injections then $(g \circ f)$ is an injection.

Let x, y and z be such that $g(f(x)) = z$ and $g(f(y)) = z$. Then, since g is an injection, we have that $f(x) = f(y)$. Therefore, since f is an injection, $x = y$ if $f(y) = f(y)$. Since g is an injection and $g(f(y)) = z$, $f(y) = f(y)$ if $g(f(y)) = z$. But this is clearly the case, so we are done.

Figure 5.7: Displaying the proofs in the `.pdf` file.



Error

The robotone Docker container is not running properly. Please review Problems.hs or your current Settings.

Figure 5.8: The error message if settings are not correctly configured, or syntax errors are detected in `Problem.hs`.

The GUI is designed to be intuitive and easy to understand. The end result would still be the same as using the command line interface to generate proofs. As many mathematics students and teachers are unfamiliar with programming and using the command line, the graphical user interface can help to make `robotone` more user-friendly.

5.2.3 Direct Installation on Host Machine

Installing `robotone` directly on the user's machine without using some kind of virtual machines or virtual environments is discouraged for many reasons. As mentioned earlier, latest versions of ghc and Cabal break the build, while the older version of ghc that `robotone` needs might interfere with other Haskell programs on the same machine.

The program should be compiled with ghc 7.8.3 and Cabal 1.18.0.5 using version 0.6.0.2 of the `logict` package. A TeX distribution is also needed to compile the generated proofs.

To set up the environment for the program in a Unix environment, ghc and Cabal must be installed first, preferably as one bundle in the Haskell Platform. The official source is at `https://www.haskell.org/platform/`. In the terminal, run the following commands to set up:

1. `$ cd \path\to\robotone\directory`

   To change the current directory to the `robotone` root directory.

2. `$ cabal update`

   To make sure that Cabal has the newest package installation paths.

3. `$ runhaskell Setup.lhs configure --user`

   If prompted that certain packages are missing, install each of them with the command `$ cabal install <package name> --global`. Most likely, the packages needed to be install are QuickCheck, logict, and parsec.

4. `$ runhaskell Setup.lhs build`

   Compile the source code.

5. `$ bash runfromsource.sh`

Run the program on the problems and facts. Results are saved in the `build/` folder.

Steps 3 and 4 are repeated for new additions to the library of definitions and theorems.

5.3 Proof Analysis and Examples

In this section we only discuss the newly added proofs, both working and not working. All eighteen working proofs, including existing ones provided by the original authors, can be found in Appendix C. Two proofs with a complete list of steps that `robotone` uses to reach the conclusion (or not) can be found in Appendix B.

5.3.1 Working Proofs

The table below is a summary of the seven new statements and theorems that `robotone` can prove at the moment. Some work on the first try with existing definitions and theorems while others (e.g. proof (7)) need additional facts before the program can make appropriate progress on them.

Table 5.1: Working proofs.

No.	Statements & Proofs	Analysis
(1)	If g, f are injections then $g \circ f$ is an injection. **Proof.** Let x, y and z be such that $g(f(x)) = z$ and $g(f(y)) = z$. Then, since g is an injection, we have that $f(x) = f(y)$. Therefore, since f is an injection, $x = y$ if $f(y) = f(y)$. Since g is an injection and $g(f(y)) = z$, $f(y) = f(y)$ if $g(f(y)) = z$. But this is clearly the case, so we are done.	This is a pretty straightforward direct proof using the existing definition of injection. However, there are a few questionable wordings in the proofs, such as the place where it needlessly writes "Therefore, since f is an injection, $x = y$ if $f(y) = f(y)$." Otherwise, the logic steps lead to the right conclusion.
(2)	Prove that $f(A \cap B) \subseteq f(A) \cap f(B)$. **Proof.** By definition, since $y \in f(A \cap B)$, there exists $z \in A \cap B$ such that $f(z) = y$. Since $z \in A \cap B$, $z \in A$ and $z \in B$. We would like to show that $y \in f(A) \cap f(B)$, i.e. that $y \in f(A)$ and $y \in f(B)$. We would like to show that $y \in f(A)$. But this is clearly the case, so we are done. Thus $y \in f(B)$ and we are done.	The proof is overall acceptable. The beginning of the proof is a little confusing, since a human mathematician would instead write, "Let $y \in f(A \cap B)$" instead of "since $y \in f(A \cap B)$". It could be a little clearer near the end by stating, "Since $z \in A$ and $y = f(z)$, $y \in f(A)$. Similarly, $z \in B$ and $y = f(z)$, $y \in f(B)$. Thus $y \in f(A) \cap f(B)$ and we are done."

Table 5.1 – continued from previous page

No.	Statements & Proofs	Analysis
(3)	Prove that $f^{-1}(A \cap B) \subseteq f^{-1}(A) \cap f^{-1}(B)$. **Proof.** Since $x \in f^{-1}(A \cap B)$, we have that $f(x) \in A \cap B$. Then $f(x) \in A$ and $f(x) \in B$. We would like to show that $x \in f^{-1}(A) \cap f^{-1}(B)$, i.e. that $x \in f^{-1}(A)$ and $x \in f^{-1}(B)$. We would like to show that $x \in f^{-1}(A)$, i.e. that $f(x) \in A$. We would like to show that $x \in f^{-1}(B)$, i.e. that $f(x) \in B$. But this is clearly the case, so we are done.	This is the forward direction ($\Rightarrow$) of the proof for $f^{-1}(A \cap B) = f^{-1}(A) \cap f^{-1}(B)$. Similar to the above proof, the wording is a little awkward and could be a little clearer. Interestingly, in the beginning the proof didn't work for this statement because the program somehow failed the first step of assuming an arbitrary $x \in f^{-1}(A \cap B)$, unlike the rest of the other proofs where it has no such problem. After that statement is included in the given premises of the problem, `robotone` can complete the proof as it's supposed to.
(4)	Prove that $f^{-1}(A) \cap f^{-1}(B) \subseteq f^{-1}(A \cap B)$. **Proof.** Let x be an element of $f^{-1}(A) \cap f^{-1}(B)$. Then $x \in f^{-1}(A)$ and $x \in f^{-1}(B)$. Then $f(x) \in A$ and $f(x) \in B$. We would like to show that $x \in f^{-1}(A \cap B)$, i.e. that $f(x) \in A \cap B$. We would like to show that $f(x) \in A \cap B$, i.e. that $f(x) \in A$ and $f(x) \in B$. But this is clearly the case, so we are done.	This is the backward direction ($\Leftarrow$) of the proof for $f^{-1}(A \cap B) = f^{-1}(A) \cap f^{-1}(B)$. This proof is clear and straightforward.

Table 5.1 – continued from previous page

No.	Statements & Proofs	Analysis
(5)	Prove that $f^{-1}(A \cup B) \subseteq f^{-1}(A) \cup f^{-1}(B)$. **Proof.** Let x be an element of $f^{-1}(A \cup B)$. Then $f(x) \in A \cup B$. Then $f(x) \in A$ or $f(x) \in B$. We would like to show that $x \in f^{-1}(A) \cup f^{-1}(B)$, i.e. that $x \in f^{-1}(A)$ or $x \in f^{-1}(B)$. We would like to show that $x \in f^{-1}(A)$, i.e. that $f(x) \in A$. But this is clearly the case, so we are done. We would like to show that $x \in f^{-1}(A) \cup f^{-1}(B)$, i.e. that $x \in f^{-1}(A)$ or $x \in f^{-1}(B)$. We would like to show that $x \in f^{-1}(A)$, i.e. that $f(x) \in A$. We would like to show that $x \in f^{-1}(B)$, i.e. that $f(x) \in B$. But this is clearly the case, so we are done.	This is the forward direction ($\Rightarrow$) of the proof for $f^{-1}(A \cup B) = f^{-1}(A) \cup f^{-1}(B)$. The logic of this proof is acceptable but somewhat twisted, since there are some unnecessarily repeated sentences.

Table 5.1 – continued from previous page

No.	Statements & Proofs	Analysis
(6)	Prove that $f^{-1}(A) \cup f^{-1}(B) \subseteq f^{-1}(A \cup B)$. **Proof.** Let x be an element of $f^{-1}(A) \cup f^{-1}(B)$. Then $x \in f^{-1}(A)$ or $x \in f^{-1}(B)$. Since $x \in f^{-1}(A)$, we have that $f(x) \in A$. Since $x \in f^{-1}(B)$, we have that $f(x) \in B$. We would like to show that $x \in f^{-1}(A \cup B)$, i.e. that $f(x) \in A \cup B$. We would like to show that $f(x) \in A \cup B$, i.e. that $f(x) \in A$ or $f(x) \in B$. But this is clearly the case, so we are done. We would like to show that $x \in f^{-1}(A \cup B)$, i.e. that $f(x) \in A \cup B$. We would like to show that $f(x) \in A \cup B$, i.e. that $f(x) \in A$ or $f(x) \in B$. But this is clearly the case, so we are done.	This is the backward direction ($\Leftarrow$) of the proof for $f^{-1}(A \cup B) = f^{-1}(A) \cup f^{-1}(B)$. Similar to the above proof, this one also repeats some sentences unnecessarily, but the overall logic works.

Table 5.1 – continued from previous page

No.	Statements & Proofs	Analysis

(7)	Prove that $A^c \cap B^c \subseteq (A \cup B)^c$. **Proof.** Let x be an element of $(A)^c \cap (B)^c$. Then $x \in (A)^c$ and $x \in (B)^c$. Then $x \notin A \cup B$ and $x \notin A$. Then it is not that case that $x \in A$. Since $x \in (B)^c$, we have that $x \notin B$. Then it is not that case that $x \in B$. Since $x \notin A \cup B$, it is not that case that $x \in A \cup B$. We would like to show that $x \in (A \cup B)^c$, i.e. that $x \notin A \cup B$. We would like to show that $x \notin A \cup B$, i.e. that it is not that case that $x \in A \cup B$. But this is clearly the case, so we are done.	This is the backward direction ($\Leftarrow$) of the proof for $(A \cup B)^c = A^c \cap B^c$. Similar to the above proof, this one also repeats some sentences unnecessarily, but the overall logic works. Originally, the proof didn't work because `robotone` could not deduce that $x \notin A$ and $x \notin B$ implies $x \notin A \cup B$. Thus we tried adding into the library several different equivalent statements of the above fact, and in the end the following makes the proof work,
		`Result "" [`
		`      parse formula "in(x, complement(A))",`
		`      parse formula "in(x, complement(B))"]`
		`       (parse formula "notin(x, union(A, B))")`
		This does look like adding the conclusion of the proof to the library, however. A better solution would be implementing a more general use of De Morgan's laws.

5.3.2 Incomplete Proofs

In this section we present two examples of incomplete proofs written by `robotone`, and give some hypotheses on why they do not work. Both of them concern the newly added definition of surjection.

Table 5.2: Incomplete proofs.

No.	Statements & Partial Proofs	Analysis
(1)	If g,f are surjections then (g o f) is a surjection. **Partial Proof.** Let y be an element of C. Then, since g from B to C is a surjection, we have that $y \in g(B)$. That is, there exists $z \in B$ such that $g(z) = y$. Since f from A to B is a surjection and $z \in B$, we have that $z \in f(A)$. That is, there exists $u \in A$ such that $f(u) = z$. We would like to find $v \in A$ s.t. $g(f(v)) = y$.	We tried to add a new definition to the program's library of facts and theorems, in this case the definition of surjection, and this is the first proof to test the new definition out. At first, we define surjection in a simplified fashion as `("surjection(f)", "forall y.( exists x.(equals(applyfn(f,x) , y)))")` Given the definition above, the program simply makes no moves. Then we change the definition to `("surjection(f, A, B)", "forall y .(in(y, B) => (exists x.(in(x , A) & equals(applyfn(f,x),y) )))")` With the new definition, `robotone` manages to make much better progress toward the complete proof. The only missing step that this proof needs is, "Set $v = u$ and we are done". Our hypothesis is that the priority of tactics allows the program to clean up too early and delete some premises that are actually needed later in the proof.

Table 5.2 – continued from previous page

No.	Statements & Proofs	Analysis
(2)	If f is a surjection then $f(A)^c \subset f(A^c)$. **Partial Proof.** Let x be an element of $(f(A))^c$. Then $x \notin f(A)$. Then it is not that case that $x \in f(A)$. We would like to find $y \in (A)^c$ s.t. $f(y) = x$. But $y \in (A)^c$ if and only if $y \notin A$. But $y \notin A$ if and only if it is not that case that $y \in A$.	This proof is incomplete because `robotone` is not using the fact that f is a surjection. After deducing that $x \notin f(A)$, the rest of the proof should have been along the lines of, "Because f is surjection, there exists t such that $f(t) = x$ and $t \notin A$ since $x \notin f(A)$. So $t \in A^c$, therefore $x = f(t) \in f(A^c)$ and we are done."

CHAPTER *6*

Challenges and Future Work

In this chapter we discuss `robotone's` limitations and future work on the software. We encounter typical challenges when one attempts to develop a computer system that closely imitates the workings of the human brain, namely the human brain's ability to create new thoughts, and to retain a seemingly infinite number of little facts and experiences across many different problem domains, which very few systems can achieve, especially given the limited time frame and resources. There is certainly much work to be done in the future regarding applying appropriate machine learning techniques for more dynamic fact-accumulating mechanisms and problem solving strategies within the current logic framework, as well as adding more tactics to `robotone`'s arsenal of proof techniques.

6.1 Challenges

6.1.1 Robotone Cannot Apply Creativity in Mathematical Proofs

"Creativity" in this context means the construction of new variables, functions, and other identities that may be critical to the arguments in the proof. We find that `robotone` is incapable of creating anything new; it can only work exclusively with what is included in, or is obviously a direct result of, the premises of the given problem.

This came up quite early in this project as we considered certain proofs regarding sequences and series in the Real Analysis course, which often require algebraic manipulation of ϵ. For example, consider the proof for the Cauchy criterion, which states that every convergent sequence is Cauchy.

Theorem 6.1 (Cauchy criterion)**.**
Every convergent sequence is Cauchy.

Proof. Let $(x_n) \in \mathbb{R}$ be a sequence that converges to $L \in \mathbb{R}$. Let $\epsilon > 0$. Then $\frac{\epsilon}{2} > 0$. Because $(x_n) \to L$, there exists $N \in \mathbb{R}$ such that for all $k > N, |\, x_k - L \,| < \frac{\epsilon}{2}$. Then for any $m > N$ and $n > N$, we have

$$\begin{aligned} |\, x_m - x_n \,| &\leq |\, x_m - L \,| + |\, L - x_n \,| \text{ by the triangle inequality} \\ &= |\, x_m - L \,| + |\, x_n - L \,| \\ &< \frac{\epsilon}{2} + \frac{\epsilon}{2} \\ &= \epsilon. \end{aligned}$$

Thus (x_n) is Cauchy. □

In this example, the main trick is to come up with $\frac{\epsilon}{2}$ instead of trying to work directly with ϵ itself. As for why $\frac{\epsilon}{2}$ and not other expressions of ϵ, a human mathematician would observe that the definition of a Cauchy sequence mentions the distance between only two elements of the sequence beyond a certain point N, meaning the triangle inequality can be combined directly with the inequalities established by the definition of convergence. The logic framework that `robotone` follows does not yet have this capability of noticing such subtle patterns in order to form something new that can be useful in the proof. Another consequence is that `robotone` cannot disprove a false statement by presenting a counterexample.

A huge reason is that `robotone` does not have a symbolic algebra engine that can manipulate expressions into a certain form, which is often required by proofs. Existing platforms such as Maple and Mathematica have not achieved that purpose yet, despite teams of engineers and readily available resources.

6.1.2 ROBOTONE CANNOT APPLY UNDERLYING ASSUMPTIONS LIKE HUMANS

Mathematics is taught as early as primary school in most places. Inevitably, over the years of learning, human mathematicians have come to accept quite a number of mathematical facts as true, and use them in proofs with or without stating them explicitly. However, a program like `robotone` does not have those years of training and can only work with what is manually supplied in its library of facts by the authors and users of the program, who may or may not remember to include each and every single one of those facts. Consequently, `robotone` may fail to prove a statement successfully even though it looks obvious to a human mathematician.

6.2 FUTURE WORK

We outline two possible ways to extend the capabilities of `robotone` below. There are certainly other aspects of `robotone` to improve, which can be addressed in future research.

6.2.1 CREATING NEW TACTICS

The list of tactics implemented in `robotone` as mentioned in Chapter 4 is by no means an exhaustive list of strategies that human mathematicians use in proving or disapproving statements. In the future, if a symbolic algebra system is developed for `robotone`, that would call for many more new tactics regarding creating new mathematical objects to assist with the proofs. For example, an additional class of tactics that enables contradiction proofs and creation of counterexamples can greatly enhance `robotone`. In addition, a more seasoned and experienced mathematician with years into a mathematics career is more likely to be able to offer more intricate tactics to solve a problem compared to an undergraduate student, which can be a different extension of `robotone` for more advanced mathematicians.

6.2.2 Application of Machine Learning

A potential improvement that we notice is that the priority of tactics can be more dynamic. In their original paper, Ganesalingam and Gowers describe `robotone` as deterministic, meaning it produces the exact same proof for a given problem every time it runs, because the priority of tactics is hard-coded into the source code [7]. However, human mathematicians do not always operate in this way. A single mathematician certainly can devise multiple different ways to solve one problem. Additionally, they do not follow a built-in list of logical or mathematical "tactics", but choose an appropriate inference rule based on how the proof progresses so far.

It is worth noting that the method Ganesalingam and Gowers use to determine the priority of tactics is to "work through large numbers of problems [themselves]," take notes of what tactics seem suitable, apply them to more problems, and adjust the priority if it leads to incorrect choices, in order to eventually arrive at the list of tactics by priority implemented in the software [7]. This process is remarkably similar to what most machine learning algorithms do: learning certain features from a given set of training data, making adjustments on the way, and then testing on a set of test data to see whether further changes to the configurations of the model are necessary, before arriving at the most viable model. Therefore, incorporating machine learning algorithms into this central component of `robotone` may yield non-deterministic yet potentially better proofs.

Here, we illustrate a possible machine learning scenario for improving `robotone`'s tactic selection mechanism. A possible way to achieve this is to create a machine learning model that is trained by using the sequence of tactics used in each available proof by `robotone`, which ends in either "we're done" (successful) or "no moves possible" (unsuccessful). The model should move towards sequences that end up in a successful proof and away from those that end up in a unsuccessful one. The initial state of the model is one already implemented in `robotone` at the moment as shown by Listing 6.1 with 26 tactics in their currently fixed order. The trained model would ingest a sequence of tactics used so far in the current proof, and output a full list of tactics with new priority order for the next move in the proof.

```
deleteDone
deleteDoneDisjunct
deleteDangling
deleteUnmatchable
peelAndSplitUniversalConditionalTarget
splitDisjunctiveHypothesis
splitDisjunctiveTarget
peelBareUniversalTarget
removeTarget
collapseSubtableauTarget
forwardsReasoning
forwardsLibraryReasoning
expandPreExistentialHypothesis
elementaryExpansionOfHypothesis
backwardsReasoning
backwardsLibraryReasoning
elementaryExpansionOfTarget
expandPreUniversalTarget
solveBullets
automaticRewrite
unlockExistentialUniversalConditionalTarget
unlockExistentialTarget
expandPreExistentialTarget
convertDiamondToBullet
rewriteVariableVariableEquality
rewriteVariableTermEquality
```

Listing 6.1: Tactics listed by their descending order of priority

Since we are not sure exactly how to arrange tactics by their priority so that solving problems is most successful, and different priority orders may affect the outcome of a proof greatly, an unsupervised learning technique has more potential than a supervised one. Thus, we separate the set of proofs that `robotone` can generate into successful and unsuccessful ones. The successful proofs can be used to build a model that discovers characteristics of a priority order of tactics that results in successful proofs, which we want to achieve. Meanwhile, the unsuccessful proofs can be used in a secondary model that discovers the characteristics a priority order that results in the opposite, which we want to avoid. Since building these two models are analogous (the difference lies only in how we interpret the final output), we only describe how the model can be trained with successful proofs.

For each tactic T, we give all tactics ratings to denote how likely they would follow T, which can be saved as a tactic transition matrix illustrated in Table 6.1.

Table 6.1: The row of T_i contains all the ratings of all 26 tactics regarding how likely each of them is to follow T_i. So for example, after T_1, since T_{26} has a higher rating than T_2, T_{26} has been observed to follow T_1 more often in the data and thus more likely to follow T_1 to form a successful proof. One way to calculate this rating is by using the ratio between the number of transitions from T_i to T_j divided by the total number of tactic transitions in the whole proof (e.g. a 50-step proof should have 49 transitions). The numbers below are for illustration purposes only and not real values.

	T_1	T_2	...	T_{26}
T_1	0	0.11	...	0.36
T_2	0	0	...	0.18
...	...	...	...	...
T_{26}	0	0.07	...	0

Then we can think of a proof as a series of steps, $[S_1, S_2, \ldots, S_n]$ where each S_i is a tactic (as listed in Listing 6.1 and described in details in Section 4.1.1). The entry point is S_1. S_2 follows S_1, so in the developing model, we give S_2 a higher rating on S_1's row, and so forth. It may seem that after looking at all the training data, we should arrive at a model in the form of the above matrix with the perfect ratings for subsequent tactics after seeing the previous ones.

Not quite. There are two main problems with this simplified model. One, it actually modifies the probability of the next tactic based on only the last tactic used, not the entire sequence before that. Two, there is never enough data for us to be entirely sure that the ratings are correct. Thus, to account for these nuances, we use an unsupervised learning technique called Restricted Boltzmann Machine (RBM) and Deep Belief Network.

RBM is a kind of Boltzmann machines, which are networks of "symmetrically connected, neuron-like units that make stochastic decisions". RBM is often used as building blocks for Deep Belief Networks [11]. We can use this technique to modify our table above to account for the mentioned disadvantages, by having hidden layers to find certain patterns and features in the sequence of steps in a proof. Overall, this model would look like Figure 6.1.

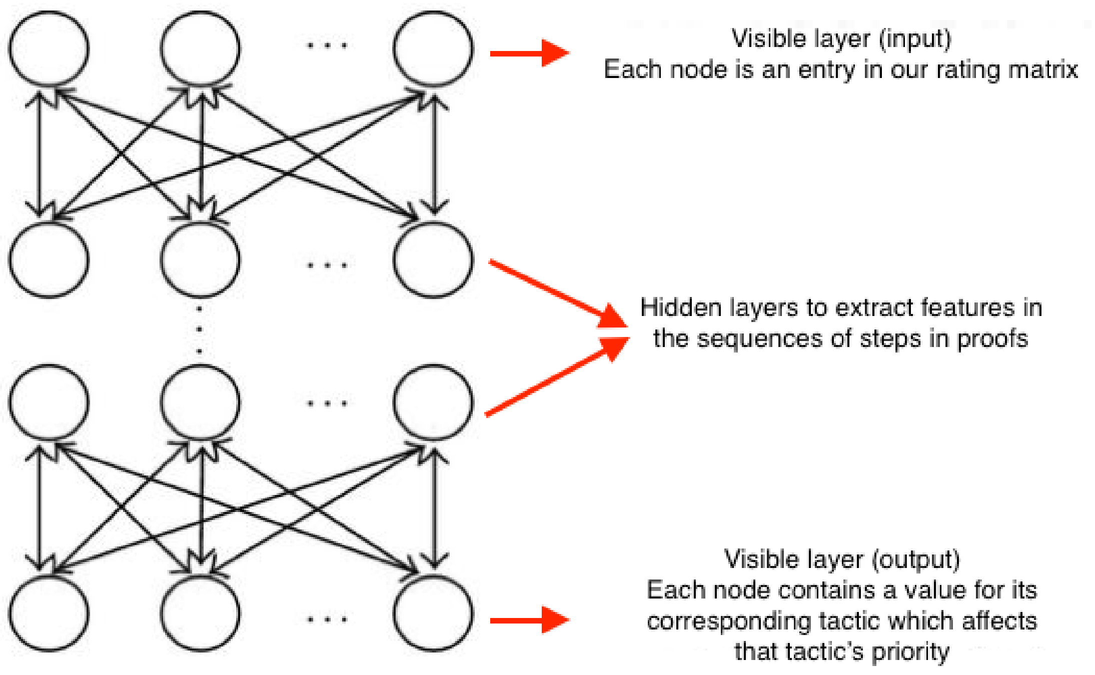

Visible layer (input)
Each node is an entry in our rating matrix
Hidden layers to extract features in the sequences of steps in proofs
Visible layer (output)
Each node contains a value for its corresponding tactic which affects that tactic's priority

Figure 6.1: Proof-of-concept deep belief network model for tactic prioritization. For a model built with successful proofs, we would sort the output nodes in descending order so that the tactics with the higher values are tried first, and vice versa for a model built with unsuccessful proofs.

The input to train the model is a tactic transition matrix which represents a single successful proof. The visible layer is our input where each node is an entry in our transition matrix, meaning there are $26^2 = 676$ nodes total. The hidden layers extract and encode different features from this matrix. Finally, the output layer consists of 26 nodes, one for each tactic, that contain the output "rating" for each tactic. We can then sort the tactics by these numbers to arrive at the new priority order for the tactics.

Consider an example use case where we are in the middle of a proof whose sequence of steps so far is $[T_7, T_3, T_{20}, T_8, T_7, T_3, T_9, T_7]$. This is encoded into a tactic transition matrix using the process detailed above. We want to use the trained model to get the list of 26 tactics in a new priority order such that it maximizes the chance of success for the proof. It is worth noting that we are not picking only one tactic for the next move, but instead getting the whole list in the best priority order. The reason is because the tactic with the highest rating might not be applicable for the proof at hand; there might be too many or too few variables for example. So suppose after sorting the output of the model, the new order of tactics is $[T_3, T_8, T_{20}, ..., T_1, T_{26}]$. `Robotone` tries the first two T_3 and T_8 but they are not applicable to the proof, so the next move in the proof is T_{20}. Then the sequence of steps in this proof is now $[T_7, T_3, T_{20}, T_8, T_7, T_3, T_9, T_7, T_{20}]$, and we continue to run the model, get all the tactics in a new priority order after each step from our machine learning model, and pick the next move as the first applicable one from that order, until the problem is solved or no move is possible (meaning we look for other factors in `robotone` to improve). This shows that the model can help `robotone` become more dynamic in its tactic selection mechanism to maximize the success of a proof.

CHAPTER 7

Conclusion

Automated theorem proving is a new and very promising area of research as a combination of pure mathematics and theoretical computer science. Lambda calculus and functional programming languages can offer a new way to implement automated theorem provers based on first-order logic in addition to the traditional logic programming paradigm. It is another direction that can have great potential by representing and manipulating math problems and proofs as λ-terms. However, one of the most prominent problems in the field of automated theorem proving is that most systems are not exactly user-friendly and often the learning curve is very steep. Thus, most human mathematicians and mathematics students in the undergraduate level are not familiar with them despite their benefits in helping with problem solving and solution verification.

This work investigates formal logic and lambda calculus theories, explores the use of an automated theorem prover software called `robotone` in the course MATH-332 Real Analysis I at The College of Wooster, and expands the software in the following ways:

- Added 16 problems in `Problems.hs`, seven of which result in successful proofs.
- Separated mathematical library into its own module `RealAnalysis.hs`, and enriched its Real Analysis contents.

- Rewrote `Main.hs` for better modularity and cleaned up unnecessary code for easier future maintenance and development.
- Created a Docker image that contains all of `robotone's` dependencies and compilation requirements, thus making the software's installation and usage much easier.
- Built a GUI on the MacOS platform for a user-friendly interaction with `robotone`.

Ganesalingam and Gowers definitely have made great progress with the first prototype of `robotone` in creating an automated theorem prover that produces proofs easy to understand by human mathematicians even if they do not know much about the program. However, the learning curve before being able to use or modify this program is still pretty steep because Haskell is a superb but unforgiving functional programming language, in which the problem statements and mathematics library are written. This is a universal problem for most automated theorem proving systems, which can be addressed in future research.

Furthermore, since `robotone` is mainly based on first-order logic and written in a functional programming language, its proof mechanism is rather limited and not robust enough to model all of the workings of a human mathematician's thinking. This project has tried to address some of those problems, but many challenges still lie ahead. The biggest one would be how to incorporate some creativity aspects into the program so that the program can come up with a new construction or example to complete a proof.

APPENDIX A

Newly Added Definitions and Theorems

Here we list the new definitions and theorems added to `RealAnalysis.hs` file. These additions customize robotone for delivering proofs for the course MATH-332 Real Analysis I.

A.0.1 New Definitions

1. Surjection: Let $f : A \to B$. f is a surjection if and only if for all $y \in B$, there exists some $x \in A$ such that $f(x) = y$.

```
    ("surjection(f, A, B)", "forall y.(in(y, B) => (exists x.(in(x,
    A) & equals(applyfn(f,x),y))))")
```

2. Limit point: x is a limit point of a set A if there exists a sequence a_n in A such that $a_n \to x$ and for all n, $x \neq a_n$.

```
        ("islimitpoint(x, A)", "exists an.(sequencein(an, A) &
   tendsto(an, x) & forall n.(~equals(x, kthterm(an, n))))")
```

3. Bounded: A set A is bounded if there exists $N > 0$ such that for all $a \in A$, $N \geq |a|$. It gets interesting here because as humans, we can instinctively consider sequences as sets and thus this definition of being bounded can apply. The program, however, needs 2 different definitions of bounded-ness, one for sequences:

```
("bounded(an)", "exists N.(forall n.(lessthan(d(0,kthterm(an,n)), N
   )))")
```

And one for sets:

```
          ("bounded(A)", "exists N.(forall a.(in(a,A) => lessthan(a
   , N)))")
```

4. Compact:

```
          ("compact(A)", "closed(A) & bounded(A)")
```

5. Increasing:

```
          ("increasing(an)", "forall m n.(lessthan(m,n) => lessthan
   (kthterm(an, m), kthterm(an, n)))")
```

6. Decreasing:

```
          ("decreasing(an)", "forall m n.(lessthan(m,n) => lessthan
   (kthterm(an, n), kthterm(an, m)))")
```

7. Monotone:

```
          ("monotone(an)", "increasing(an) | decreasing(an)")
```

8. Supremum:

```
          ("sup(s, A)", "forall a.(in(a, A) => lessthan(a, s)) &
   forall v.(lessthan(v, s) => exists b.(in(b, A) & lessthan(v, b))
   )")
```

9. Infimum:

```
          ("inf(s, A)", "forall a.(in(a, A) => lessthan(s, A)) &
   forall v.(lessthan(s, v) => exists b.(in(b, A) & lessthan(b, v))
   )")
```

A.0.2 New Theorems

1. Monotone Convergence Theorem:

```
      Result "Monotone Convergence Theorem" [
        parse formula "bounded(an)",
        parse formula "monotone(an)"]
         (parse formula "converges(an)")
```

2. Every convergent sequence is bounded:

```
    Result "every convergent sequence is bounded" [
        parse formula "converges(an)"]
         (parse formula "bounded(an)")
```

APPENDIX B

Examples of Proofs with Steps

This appendix demonstrates 2 proofs including detailed steps, one is successful and the other is not. The final proofs are given at the top below the problem statements. After that, on the left are the steps of how the program arrives at the final proof are listed out in boxes as they are represented in the logic. On the right, we can see what steps correspond to what sentences in the final proof. A summary of the two proofs is below:

If f is an injection then $f(A) \cap f(B) \subset f(A \cap B)$

Let x be an element of $f(A) \cap f(B)$. Then $x \in f(A)$ and $x \in f(B)$. That is, there exists $y \in A$ such that $f(y) = x$ and there exists $z \in B$ such that $f(z) = x$. Since f is an injection, $f(y) = x$ and $f(z) = x$, we have that $y = z$. We would like to find $u \in A \cap B$ s.t. $f(u) = x$. But $u \in A \cap B$ if and only if $u \in A$ and $u \in B$. Since $y = z$, we have that $y \in B$. Therefore, setting $u = y$, we are done.

If g,f are surjections then (g o f) is a surjection.

Let y be an element of C. Then, since g from B to C is a surjection, we have that $y \in g(B)$. That is, there exists $z \in B$ such that $g(z) = y$. Since f from A to B is a surjection and $z \in B$, we have that $z \in f(A)$. That is, there exists $u \in A$ such that $f(u) = z$. We would like to find $v \in A$ s.t. $g(f(v)) = y$.

If f is an injection then $f(A) \cap f(B) \subset f(A \cap B)$

Let x be an element of $f(A) \cap f(B)$. Then $x \in f(A)$ and $x \in f(B)$. That is, there exists $y \in A$ such that $f(y) = x$ and there exists $z \in B$ such that $f(z) = x$. Since f is an injection, $f(y) = x$ and $f(z) = x$, we have that $y = z$. We would like to find $u \in A \cap B$ s.t. $f(u) = x$. But $u \in A \cap B$ if and only if $u \in A$ and $u \in B$. Since $y = z$, we have that $y \in B$. Therefore, setting $u = y$, we are done.

L1

H1. f is an injection

T1. $f(A) \cap f(B) \subset f(A \cap B)$

1. Expand pre-universal target T1.

L1

H1. f is an injection

T2. $\forall x.(x \in f(A) \cap f(B) \Rightarrow x \in f(A \cap B))$

2. Apply 'let' trick and move premise of universal-conditional target T2 above the line.

Let x be an element of $f(A) \cap f(B)$.

L1

x

H1. f is an injection

H2. $x \in f(A) \cap f(B)$

T3. $x \in f(A \cap B)$

3. Quantifier-free expansion of hypothesis H2.

Since $x \in f(A) \cap f(B)$, $x \in f(A)$ and $x \in f(B)$.

L1

x

H1. f is an injection

H3. $x \in f(A)$

H4. $x \in f(B)$

T3. $x \in f(A \cap B)$

4. Expand pre-existential hypothesis H3.

By definition, since $x \in f(A)$, there exists $y \in A$ such that $f(y) = x$.

L1

$x \ y$

H1. f is an injection

H5. $y \in A$

H6. $f(y) = x$

H4. $x \in f(B)$

T3. $x \in f(A \cap B)$

5. Expand pre-existential hypothesis H4.

By definition, since $x \in f(B)$, there exists $z \in B$ such that $f(z) = x$.

L1

$x \ y \ z$

H1. f is an injection

H5. $y \in A$

H6. $f(y) = x$

H7. $z \in B$

H8. $f(z) = x$

T3. $x \in f(A \cap B)$

6. Forwards reasoning using H1 with (H6,H8).

Since f is an injection, $f(y) = x$ and $f(z) = x$, we have that $y = z$.

L1

$x\ y\ z$

H1. f is an injection [Vuln.; Used with (H6,H8).]
H5. $y \in A$
H6. $f(y) = x$ [Vuln.]
H7. $z \in B$
H8. $f(z) = x$ [Vuln.]
H9. $y = z$

T3. $\boldsymbol{x \in f(A \cap B)}$

7. Expand pre-existential target T3.

We would like to find $u \in A \cap B$ s.t. $f(u) = x$.

L1

$x\ y\ z$

H1. f is an injection [Vuln.; Used with (H6,H8).]
H5. $y \in A$
H6. $f(y) = x$ [Vuln.]
H7. $z \in B$
H8. $f(z) = x$ [Vuln.]
H9. $y = z$

T4. $\boldsymbol{\exists u.(u \in A \cap B \wedge f(u) = x)}$

8. Unlock existential target T4.

We would like to find $u \in A \cap B$ s.t. $f(u) = x$.

L1

$x\ y\ z$

H1. f is an injection [Vuln.; Used with (H6,H8).]
H5. $y \in A$
H6. $f(y) = x$ [Vuln.]
H7. $z \in B$
H8. $f(z) = x$ [Vuln.]
H9. $y = z$

L2$^\blacklozenge$

$u^\blacklozenge$

T5. $\boldsymbol{u^\blacklozenge \in A \cap B}$
T6. $f(u^\blacklozenge) = x$

9. Quantifier-free expansion of target T5.

But $u \in A \cap B$ if and only if $u \in A$ and $u \in B$.

L1

$x\ y\ z$

H1. f is an injection [Vuln.; Used with (H6,H8).]
H5. $y \in A$
H6. $f(y) = x$ [Vuln.]
H7. $z \in B$
H8. $f(z) = x$ [Vuln.]
H9. $\boldsymbol{y = z}$

L2$^\blacklozenge$

$u^\blacklozenge$

T7. $u^\blacklozenge \in A$
T8. $u^\blacklozenge \in B$
T6. $f(u^\blacklozenge) = x$

10. Rewrite z as y throughout the tableau using hypothesis H9.

Since $y = z$, we have that $y \in B$.

L1

$x\ y\ z$

H1. f is an injection [Vuln.; Used with (H6,H8).]
H5. $y \in A$
H6. $f(y) = x$ [Vuln.]
H10. $y \in B$

L2$^\blacklozenge$

$u^\blacklozenge$

T7. $u^\blacklozenge \in A$
T8. $u^\blacklozenge \in B$
T6. $f(u^\blacklozenge) = x$

11. Moved H6 down, as x can only be utilised by T6.

L1

$x\ y\ z$

H1. f is an injection [Vuln.; Used with (H6,H8).]
H5. $y \in A$
H10. $y \in B$

L2$^\blacklozenge$

$u^\blacklozenge$

T7. $u^\blacklozenge \in A$
T8. $u^\blacklozenge \in B$

L3

H6. $f(y) = x$ [Vuln.; From L1.]

T6. $f(u^\blacklozenge) = x$

12. Choosing $u^\blacklozenge = y$ matches all targets inside L2$^\blacklozenge$ against hypotheses, so L2$^\blacklozenge$ is done.

Therefore, setting $u = y$, we are done.

L1

$x\ y\ z$

H1. f is an injection [Vuln.; Used with (H6,H8).]
H5. $y \in A$
H10. $y \in B$

L2$^\blacklozenge$ Done

13. All targets of L1 are 'Done', so L1 is itself done.

L1 Done

Problem solved.

If g,f are surjections then (g o f) is a surjection.

Let y be an element of C. Then, since g from B to C is a surjection, there exists $u \in B$ such that $g(u) = y$. Since f from A to B is a surjection and $u \in B$, there exists $v \in A$ such that $f(v) = u$. We would like to find $x \in A$ s.t. $g(f(x)) = y$.

L1

H1. f from A to B is a surjection
H2. g from B to C is a surjection

T1. $g \circ f$ from A to C is a surjection

1. Expand pre-universal target T1.

L1

H1. f from A to B is a surjection
H2. g from B to C is a surjection

T2. $\forall y.(y \in C \Rightarrow \exists x.(x \in A \wedge g(f(x)) = y))$

2. Apply 'let' trick and move premise of universal-conditional target T2 above the line.

Let y be an element of C.

L1

y

H1. f from A to B is a surjection
H2. g from B to C is a surjection
H3. $y \in C$

T3. $\exists x.(x \in A \wedge g(f(x)) = y)$

3. Forwards reasoning using H2 with H3.

Since g from B to C is a surjection and $y \in C$, there exists $u \in B$ such that $g(u) = y$.

L1

$y \ u[y]$

H1. f from A to B is a surjection
H2. g from B to C is a surjection [Vuln.; Used with H3.]
H3. $y \in C$ [Vuln.]
H4. $u[y] \in B$
H5. $g(u[y]) = y$

T3. $\exists x.(x \in A \wedge g(f(x)) = y)$

4. Deleted H3, as this unexpandable atomic statement is unmatchable.

L1

$y \ u[y]$

H1. f from A to B is a surjection
H2. g from B to C is a surjection [Vuln.; Used with H3.]
H3. $y \in C$ [Vuln.]
H4. $u[y] \in B$
H5. $g(u[y]) = y$

T3. $\exists x.(x \in A \wedge g(f(x)) = y)$

5. Delete H2 as no other statement mentions C.

L1

$y \ u[y]$

H1. f from A to B is a surjection
H2. g from B to C is a surjection [Vuln.; Used with H3.]
H3. $y \in C$ [Vuln.]
H4. $u[y] \in B$
H5. $g(u[y]) = y$

T3. $\exists x.(x \in A \wedge g(f(x)) = y)$

6. Forwards reasoning using H1 with H4.

Since f from A to B is a surjection and $u \in B$, there exists $v \in A$ such that $f(v) = u$.

L1

$y \; u[y] \; v[u[y]]$

H1. f from A to B is a surjection [Vuln.; Used with H4.]
H2. g from B to C is a surjection [Vuln.; Used with H3.]
H3. $y \in C$ [Vuln.]
H4. $u[y] \in B$ [Vuln.]
H5. $g(u[y]) = y$
H6. $v[u[y]] \in A$
H7. $f(v[u[y]]) = u[y]$

T3. $\exists x.(x \in A \wedge g(f(x)) = y)$

7. Deleted H4, as this unexpandable atomic statement is unmatchable.

L1

$y \; u[y] \; v[u[y]]$

H1. f from A to B is a surjection [Vuln.; Used with H4.]
H2. g from B to C is a surjection [Vuln.; Used with H3.]
H3. $y \in C$ [Vuln.]
H4. $u[y] \in B$ [Vuln.]
H5. $g(u[y]) = y$
H6. $v[u[y]] \in A$
H7. $f(v[u[y]]) = u[y]$

T3. $\exists x.(x \in A \wedge g(f(x)) = y)$

8. Delete H1 as no other statement mentions B.

L1

$y \; u[y] \; v[u[y]]$

H1. f from A to B is a surjection [Vuln.; Used with H4.]
H2. g from B to C is a surjection [Vuln.; Used with H3.]
H3. $y \in C$ [Vuln.]
H4. $u[y] \in B$ [Vuln.]
H5. $g(u[y]) = y$
H6. $v[u[y]] \in A$
H7. $f(v[u[y]]) = u[y]$

T3. $\boldsymbol{\exists x.(x \in A \wedge g(f(x)) = y)}$

9. Unlock existential target T3.

We would like to find $x \in A$ s.t. $g(f(x)) = y$.

L1

$y \; u[y] \; v[u[y]]$

H1. f from A to B is a surjection [Vuln.; Used with H4.]
H2. g from B to C is a surjection [Vuln.; Used with H3.]
H3. $y \in C$ [Vuln.]
H4. $u[y] \in B$ [Vuln.]
H5. $g(u[y]) = y$
H6. $v[u[y]] \in A$
H7. $f(v[u[y]]) = u[y]$

L2♦

$x^\blacklozenge$

T4. $x^\blacklozenge \in A$
T5. $g(f(x^\blacklozenge)) = y$

No moves possible.

APPENDIX C

List of All Proofs

If f is an injection then $f(A) \cap f(B) \subset f(A \cap B)$

Let x be an element of $f(A) \cap f(B)$. Then $x \in f(A)$ and $x \in f(B)$. That is, there exists $y \in A$ such that $f(y) = x$ and there exists $z \in B$ such that $f(z) = x$. Since f is an injection, $f(y) = x$ and $f(z) = x$, we have that $y = z$. We would like to find $u \in A \cap B$ s.t. $f(u) = x$. But $u \in A \cap B$ if and only if $u \in A$ and $u \in B$. Since $y = z$, we have that $y \in B$. Therefore, setting $u = y$, we are done.

If g, f are injections then $(g \circ f)$ is an injection.

Let x, y and z be such that $g(f(x)) = z$ and $g(f(y)) = z$. Then, since g is an injection, we have that $f(x) = f(y)$. Therefore, since f is an injection, $x = y$ if $f(y) = f(y)$. Since g is an injection and $g(f(y)) = z$, $f(y) = f(y)$ if $g(f(y)) = z$. But this is clearly the case, so we are done.

$$A \subseteq f^{-1}(f(A))$$

Let x be an element of A. We would like to show that $x \in f^{-1}(f(A))$, i.e. that $f(x) \in f(A)$. But this is clearly the case, so we are done.

$$f(f^{-1}(A)) \subset A$$

Let x be an element of $f(f^{-1}(A))$. Then there exists $y \in f^{-1}(A)$ such that $f(y) = x$. Since $y \in f^{-1}(A)$, we have that $f(y) \in A$. Since $f(y) = x$, we have that $x \in A$. But this is clearly the case, so we are done.

$$f(A \cap B) \subseteq f(A) \cap f(B)$$

By definition, since $y \in f(A \cap B)$, there exists $z \in A \cap B$ such that $f(z) = y$. Since $z \in A \cap B$, $z \in A$ and $z \in B$. We would like to show that $y \in f(A) \cap f(B)$, i.e. that $y \in f(A)$ and $y \in f(B)$. We would like to show that $y \in f(A)$. But this is clearly the case, so we are done. Thus $y \in f(B)$ and we are done.

$$f^{-1}(A \cap B) \subseteq f^{-1}(A) \cap f^{-1}(B)$$

Since $x \in f^{-1}(A \cap B)$, we have that $f(x) \in A \cap B$. Then $f(x) \in A$ and $f(x) \in B$. We would like to show that $x \in f^{-1}(A) \cap f^{-1}(B)$, i.e. that $x \in f^{-1}(A)$ and $x \in f^{-1}(B)$. We would like to show that $x \in f^{-1}(A)$, i.e. that $f(x) \in A$. We would like to show that $x \in f^{-1}(B)$, i.e. that $f(x) \in B$. But this is clearly the case, so we are done.

$$f^{-1}(A) \cap f^{-1}(B) \subseteq f^{-1}(A \cap B)$$

Let x be an element of $f^{-1}(A) \cap f^{-1}(B)$. Then $x \in f^{-1}(A)$ and $x \in f^{-1}(B)$. Then $f(x) \in A$ and $f(x) \in B$. We would like to show that $x \in f^{-1}(A \cap B)$, i.e. that $f(x) \in A \cap B$. We would like to show that $f(x) \in A \cap B$, i.e. that $f(x) \in A$ and $f(x) \in B$. But this is clearly the case, so we are done.

$$f^{-1}(A \cup B) \subseteq f^{-1}(A) \cup f^{-1}(B)$$

Let x be an element of $f^{-1}(A \cup B)$. Then $f(x) \in A \cup B$. Then $f(x) \in A$ or $f(x) \in B$. We would like to show that $x \in f^{-1}(A) \cup f^{-1}(B)$, i.e. that $x \in f^{-1}(A)$ or $x \in f^{-1}(B)$. We would like to show that $x \in f^{-1}(A)$, i.e. that $f(x) \in A$. But this is clearly the case, so we are done. We would like to show that $x \in f^{-1}(A) \cup f^{-1}(B)$, i.e. that $x \in f^{-1}(A)$ or $x \in f^{-1}(B)$. We would like to show that $x \in f^{-1}(A)$, i.e. that $f(x) \in A$. We would like to show that $x \in f^{-1}(B)$, i.e. that $f(x) \in B$. But this is clearly the case, so we are done.

$$f^{-1}(A) \cup f^{-1}(B) \subseteq f^{-1}(A \cup B)$$

Let x be an element of $f^{-1}(A) \cup f^{-1}(B)$. Then $x \in f^{-1}(A)$ or $x \in f^{-1}(B)$. Since $x \in f^{-1}(A)$, we have that $f(x) \in A$. Since $x \in f^{-1}(B)$, we have that $f(x) \in B$. We would like to show that $x \in f^{-1}(A \cup B)$, i.e. that $f(x) \in A \cup B$. We would like to show that $f(x) \in A \cup B$, i.e. that $f(x) \in A$ or $f(x) \in B$. But this is clearly the case, so we are done. We would like to show that $x \in f^{-1}(A \cup B)$, i.e. that $f(x) \in A \cup B$. We would like to show that $f(x) \in A \cup B$, i.e. that $f(x) \in A$ or $f(x) \in B$. But this is clearly the case, so we are done.

If A and B are open sets, then $A \cup B$ is also open.

Let x be an element of $A \cup B$. Then $x \in A$ or $x \in B$. Since A is open and $x \in A$, there exists $\alpha > 0$ such that $w \in A$ whenever $d(x, w) < \alpha$. Since B is open and $x \in B$, there exists $\beta > 0$ such that $p \in B$ whenever $d(x, p) < \beta$. We would like to find $\eta > 0$ s.t. $z \in A \cup B$ whenever $d(x, z) < \eta$. But $z \in A \cup B$ if and only if $z \in A$ or $z \in B$. We know that $z \in A$ if $d(x, z) < \alpha$. Therefore, setting $\eta = \alpha$, we are done. We would like to find $\theta > 0$ s.t. $u \in A \cup B$ whenever $d(x, u) < \theta$. But $u \in A \cup B$ if and only if $u \in A$ or $u \in B$. We know that $u \in B$ if $d(x, u) < \beta$. Therefore, setting $\theta = \beta$, we are done.

If A, B,and C are open sets, then $A \cup (B \cup C)$ is also open.

Let x be an element of $A \cup B \cup C$. Then $x \in A$ or $x \in B \cup C$. Since A is open and $x \in A$, there exists $\alpha > 0$ such that $w \in A$ whenever $d(x, w) < \alpha$. Since $x \in B \cup C$, $x \in B$ or $x \in C$. Since B is open and $x \in B$, there exists $\delta' > 0$ such that $r \in B$ whenever $d(x, r) < \delta'$. Since C is open and $x \in C$, there exists $\delta'' > 0$ such that $s \in C$ whenever $d(x, s) < \delta''$. We would like to find $\eta > 0$ s.t. $z \in A \cup B \cup C$ whenever $d(x, z) < \eta$. But $z \in A \cup B \cup C$ if and only if $z \in A$ or $z \in B \cup C$. We know that $z \in A$ if $d(x, z) < \alpha$. Therefore, setting $\eta = \alpha$, we are done. We would like to find $\beta > 0$ s.t. $v \in A \cup B \cup C$ whenever $d(x, v) < \beta$. But $v \in A \cup B \cup C$ if and only if $v \in A$ or $v \in B \cup C$. We would like to show that $v \in B \cup C$, i.e. that $v \in B$ or $v \in C$. We know that $v \in B$ if $d(x, v) < \delta'$. Therefore, setting $\beta = \delta'$, we are done. We would like to find $\gamma > 0$ s.t. $p \in A \cup B \cup C$ whenever $d(x, p) < \gamma$. But $p \in A \cup B \cup C$ if and only if $p \in A$ or $p \in B \cup C$. We would like to show that $p \in B \cup C$, i.e. that $p \in B$ or $p \in C$. We know that $p \in C$ if $d(x, p) < \delta''$. Therefore, setting $\gamma = \delta''$, we are done.

If A and B are open sets, then $A \cap B$ is also open.

Let x be an element of $A \cap B$. Then $x \in A$ and $x \in B$. Therefore, since A is open, there exists $\eta > 0$ such that $u \in A$ whenever $d(x, u) < \eta$ and since B is open, there exists $\theta > 0$ such that $v \in B$ whenever $d(x, v) < \theta$. We would like to find $\delta > 0$ s.t. $y \in A \cap B$ whenever $d(x, y) < \delta$. But $y \in A \cap B$ if and only if $y \in A$ and $y \in B$. We know that $y \in A$ whenever $d(x, y) < \eta$ and that $y \in B$ whenever $d(x, y) < \theta$. Assume now that $d(x, y) < \delta$. Then $d(x, y) < \eta$ if $\delta \leqslant \eta$ and $d(x, y) < \theta$ if $\delta \leqslant \theta$. We may therefore take $\delta = \min(\eta, \theta)$ and we are done.

If A and B are closed sets, then $A \cap B$ is also closed.

Let (a_n) and a be such that (a_n) is a sequence in $A \cap B$ and $a_n \to a$. Then (a_n) is a sequence in A and (a_n) is a sequence in B. Therefore, since A is closed and $a_n \to a$, we have that $a \in A$ and since B is closed and $a_n \to a$, we have that $a \in B$. We would like to show that $a \in A \cap B$, i.e. that $a \in A$ and $a \in B$. But this is clearly the case, so we are done.

The pre-image of a closed set U under a continuous function f is closed.

Let (a_n) and a be such that (a_n) is a sequence in $f^{-1}(U)$ and $a_n \to a$. Then $f(a_n)$ is a sequence in U. We would like to show that $a \in f^{-1}(U)$, i.e. that $f(a) \in U$ and since U is closed, $f(a) \in U$ if $f(a_n) \to f(a)$. Let $\epsilon > 0$. We would like to find N s.t. $d(f(a), f(a_n)) < \epsilon$ whenever $n \geqslant N$. Since f is continuous, there exists $\delta > 0$ such that $d(f(a), f(a_n)) < \epsilon$ whenever $d(a, a_n) < \delta$. Since $a_n \to a$, there exists N' such that $d(a, a_n) < \delta$ whenever $n \geqslant N'$. Therefore, setting $N = N'$, we are done.

The pre-image of an open set U under a continuous function f is open.

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.

If f and g are continuous functions, then $g \circ f$ is continuous.

Take x and $\epsilon > 0$. We would like to find $\delta > 0$ s.t. $d(g(f(x)), g(f(y))) < \epsilon$ whenever $d(x, y) < \delta$. Since g is continuous, there exists $\eta > 0$ such that $d(g(f(x)), g(f(y))) < \epsilon$ 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.

If f is a continuous function and $(a_n) \to a$, then $(f(a_n)) \to f(a)$

Let $\epsilon > 0$. We would like to find N s.t. $d(f(a), f(a_n)) < \epsilon$ whenever $n \geqslant N$. Since f is continuous, there exists $\delta > 0$ such that $d(f(a), f(a_n)) < \epsilon$ whenever $d(a, a_n) < \delta$. Since $a_n \to a$, there exists N' such that $d(a, a_n) < \delta$ whenever $n \geqslant N'$. Therefore, setting $N = N'$, we are done.

$A^c \cap B^c \subseteq (A \cup B)^c$.

Let x be an element of $(A)^c \cap (B)^c$. Then $x \in (A)^c$ and $x \in (B)^c$. Then $x \notin A \cup B$ and $x \notin A$. Then it is not that case that $x \in A$. Since $x \in (B)^c$, we have that $x \notin B$. Then it is not that case that $x \in B$. Since $x \notin A \cup B$, it is not that case that $x \in A \cup B$. We would like to show that $x \in (A \cup B)^c$, i.e. that $x \notin A \cup B$. We would like to show that $x \notin A \cup B$, i.e. that it is not that case that $x \in A \cup B$. But this is clearly the case, so we are done.

If g, f are surjections then $(g \circ f)$ is a surjection.

Let y be an element of C. Then, since g from B to C is a surjection, there exists $u \in B$ such that $g(u) = y$ and $g(u) \in C$. Since f from A to B is a surjection and $u \in B$, there exists $v \in A$ such that $f(v) = u$ and $f(v) \in B$. We would like to find $x \in A$ s.t. $g(f(x)) = y$ and $g(f(x)) \in C$.

If f is a surjection then $f(A)^c \subset f(A^c)$

Let x be an element of $(f(A))^c$. Then $x \notin f(A)$. Then it is not that case that $x \in f(A)$. We would like to find $y \in (A)^c$ s.t. $f(y) = x$. But $y \in (A)^c$ if and only if $y \notin A$. But $y \notin A$ if and only if it is not that case that $y \in A$.

$$(A \cap B)^c \subset A^c \cup B^c$$

Let x be an element of $(A \cap B)^c$. Then $x \notin A \cap B$. Then it is not that case that $x \in A \cap B$. We would like to show that $x \in (A)^c \cup (B)^c$, i.e. that $x \in (A)^c$ or $x \in (B)^c$. We would like to show that $x \in (A)^c$, i.e. that $x \notin A$. We would like to show that $x \notin A$, i.e. that it is not that case that $x \in A$. We would like to show that $x \in (B)^c$, i.e. that $x \notin B$. We would like to show that $x \notin B$, i.e. that it is not that case that $x \in B$.

$$A^c \cup B^c \subset (A \cap B)^c$$

Let x be an element of $(A)^c \cup (B)^c$. Then $x \in (A)^c$ or $x \in (B)^c$. Since $x \in (A)^c$, we have that $x \notin A$. Then it is not that case that $x \in A$. Since $x \in (B)^c$, we have that $x \notin B$. Then it is not that case that $x \in B$. We would like to show that $x \in (A \cap B)^c$, i.e. that $x \notin A \cap B$. We would like to show that $x \notin A \cap B$, i.e. that it is not that case that $x \in A \cap B$. We would like to show that $x \in (A \cap B)^c$, i.e. that $x \notin A \cap B$. We would like to show that $x \notin A \cap B$, i.e. that it is not that case that $x \in A \cap B$.

$$(A \cup B)^c = A^c \cap B^c$$

We would like to show that $(A \cup B)^c \subset (A)^c \cap (B)^c$, i.e. that $(A \cup B)^c \subset (A)^c$ and $(A \cup B)^c \subset (B)^c$.

If A, B,and C are open sets, then $A \cap (B \cap C)$ is also open.

Let x be an element of $A \cap B \cap C$. Then $x \in A$ and $x \in B \cap C$. Therefore, since A is open, there exists $\eta > 0$ such that $u \in A$ whenever $d(x,u) < \eta$ and $x \in B$ and $x \in C$. Therefore, since B is open, there exists $\theta > 0$ such that $v \in B$ whenever $d(x,v) < \theta$ and since C is open, there exists $\alpha > 0$ such that $w \in C$ whenever $d(x,w) < \alpha$. We would like to find $\delta > 0$ s.t. $y \in A \cap B \cap C$ whenever $d(x,y) < \delta$. But $y \in A \cap B \cap C$ if and only if $y \in A$ and $y \in B \cap C$. We know that $y \in A$ whenever $d(x,y) < \eta$. But $y \in B \cap C$ if and only if $y \in B$ and $y \in C$. We know that $y \in B$ whenever $d(x,y) < \theta$ and that $y \in C$ whenever $d(x,y) < \alpha$. Assume now that $d(x,y) < \delta$. Then $d(x,y) < \eta$ if $\delta \leqslant \eta$, $d(x,y) < \theta$ if $\delta \leqslant \theta$ and $d(x,y) < \alpha$ if $\delta \leqslant \alpha$.

If A and B are closed sets, then $A \cup B$ is also closed.

Let (a_n) and a be such that (a_n) is a sequence in $A \cup B$ and $a_n \to a$. We would like to show that $a \in A \cup B$, i.e. that $a \in A$ or $a \in B$. Since A is closed and $a_n \to a$, $a \in A$ if (a_n) is a sequence in A. Since B is closed and $a_n \to a$, $a \in B$ if (a_n) is a sequence in B. Take n. Take n'.

References

1. Barendregt, H. P., Wil Dekkers, Richard Statman, and Fabio Alessi. *Lambda Calculus with Types*. New York: Cambridge University Press: Ohio Library and Information Network, and Association for Symbolic Logic, 2013.

2. Barendsen, Erik, and Henk Barendregt. *Introduction to Lambda Calculus*. Institute for Computing and Information Sciences, Radboud University, 2000. `ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf`.

3. Boyer, R. S., M. Kaufmann, and J. S. Moore. "The Boyer-Moore Theorem Prover and Its Interactive Enhancement." *Computers and Mathematics with Applications* 29, 2: (1995) 27–62. `http://rave.ohiolink.edu/ejournals/article/327666409`.

4. Coq Community. "The Coq Proof Assistant.", 2017. `https://coq.inria.fr/`. Accessed on 02/17/2017.

5. Docker Inc. "Docker.", 2017. `https://www.docker.com`. Accessed on 08/17/2017.

6. EQP Community. "EQP: Equational Prover.", 2017. `http://www.cs.unm.edu/~mccune/eqp/`. Accessed on 02/17/2017.

7. Ganesalingam, M., and W.T. Gowers. "A Fully Automatic Theorem Prover with Human-Style Output." *Journal of Automated Reasoning* 58: (2017) 253–291. `http://rave.ohiolink.edu/ejournals/article/345367909`.

8. Gransden, Thomas. "Boosting Automated Reasoning by Mining Existing Proofs.", 2013. `http://www.cs.le.ac.uk/people/tg75/arw13.pdf`. Accessed on 02/17/2017.

9. Heras, Jónathan, and Ekaterina Komendantskaya. "ML4PG in Computer Algebra Verification." *CoRR* abs/1302.6421. `http://arxiv.org/abs/1302.6421`.

10. Hindley, J. Roger, and J. P. Seldin. *Lambda-calculus and Combinators, an Introduction*. Cambridge, UK;New York: Cambridge University Press, 2008.

11. Hinton, G. E. "Boltzmann Machine." *Scholarpedia* 2, 5: (2007) 1668. Revision #91075.

12. Hudak, Paul. "Lecture notes in CPSC-201 Introduction to Computer Science.", 2008. `http://www.cs.yale.edu/homes/hudak/CS201S08/lambda.pdf`. Accessed on 09/17/2017.

13. Kragl, Bernhard. "Machine Learning for Automated Theorem Proving.", 2015. `http://pub.ist.ac.at/~chl/ML2015/kragl-sma2015.pdf`. Accessed on 02/17/2017.

14. Mcgregor, Ralph Eric. *Automated Theorem Proving Using Sat*. Ph.D. thesis, Clarkson University, Potsdam, NY, USA, 2011. AAI3471671.

15. Miller, Dale, and Gopalan Nadathur. *Programming with Higher-Order Logic*. New York, NY: Cambridge University Press, 2012.

16. Nipkow, Tobias, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283. New York, Berlin: Springer, 2002, 1st edition.

17. Plaisted, David A. "Automated Theorem Proving." *Wiley Interdisciplinary Reviews: Cognitive Science* 5: (2014) 115–128. `http://rave.ohiolink.edu/ejournals/article/321835984`.

18. Rojas, Raúl. "A Tutorial Introduction to the Lambda Calculus." *CoRR* abs/1503.09060. `http://arxiv.org/abs/1503.09060`.

19. Vampire Community. "The Vampire Theorem Prover.", 2017. `http://www.vprover.org`. Accessed on 02/17/2017.

20. Wos, Larry et al. *Automated Reasoning: Introduction and Applications*. Englewood Cliffs, N.J: Prentice-Hall, 1984.