Imperial College
London

BEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Emmy:
A Proof Assistant for Reasoning about Programs

Author:
Junfeng Xu

Supervisor:
Sophia Drossopoulou

June 17, 2019

Submitted in partial fulfillment of the requirements for the BEng Computing of Imperial College London

Abstract

We present EMMY, a proof assistant optimised for teaching and learning, that fills the gap between existing teaching tools and powerful practical theorem provers. EMMY supports a many-sorted first-order logic in which recursive functions, data structures, and arithmetic operations can be expressed. EMMY can express and prove the properties of many computer programs, in addition to theorems in propositional and first-order logic.

To make it convenient for students to start using EMMY, we also developed a web-based interface for EMMY, which has been proven in tests to be easy to learn. Alternatively, the users may also write proofs in an LISP-like DSL, and check their proofs by 'running' them using EMMY's interpreter.

Since we intend to use EMMY in the teaching of the reasoning about programs course, we demonstrated the proving power of EMMY with regard to the course materials. The results are satisfying: we believe that EMMY has enough proving power to be used in actual teaching.

Emmy Open Save HIDE ALL SHOW ALL CHECK

Declare HIDE X

```
x  Data List = nil | cons Int List
x  append: List List -> List
x  rev: List -> List
ADD
```

The integer type (Int) and atom type (Atom) are already defined for you.

Lemmas HIDE X

x LD $\forall$l1:List.$\forall$l2:List.rev (append l1 l2) = append (rev l2) (rev l1)

ADD

Define HIDE X

```
x  append-b   append nil l = l
x  append-i   append (cons i l1) l2 = cons i (append l1 l2)
x  rev-b      rev nil = nil
x  rev-i      rev (cons i l) = append (rev l) (cons i nil)
ADD
```

Proof HIDE X

Checked: Correct

Goal: $\forall$l:List.l = rev (rev l)

G Prove $\forall$l:List.l = rev (rev l) ->Function X

by induction on the definition of data structure List

Induction principle is correct

Base case

P+I P+IA

To show nil = rev (rev nil) X ->Subproof

By rev-b

Inductive Step

X Take l: List such that l = rev (rev l)

X Take x: Int

P+I P+IA

To show cons x l = rev (rev (cons x l)) X ->Subproof

By append-b append-i rev-b rev-i LD %Hypo

Add Case

+ A+ I+ IA+ Ind+ E+

Renumber

ADD COMMENT ADD PROOF ADD DECLARE ADD LEMMAS ADD DEFINE

Figure 1: A proof of theorem B.15 written using the web interface of Emmy

Acknowledgements

I would like to thank Professor Sophia Drossopoulou, my supervisor, for giving me the oppotunity to work on this project, and her kind guidance throughout the course of the project.

I would like to thank Theo Charalambous and Ben Pahnke for helping me test Emmy, and providing me with invaluable feedback.

I would also like to thank Fangyi Zhou for sharing with me a list of projects available for MEng students. Without their help, I would not have know about this project in the first place.

Contents

I Introduction 7

1 Introduction 8
- 1.1 Motivation . . . 8
- 1.2 Objectives . . . 8
- 1.3 Contribution . . . 8

2 Background 10
- 2.1 Logic and Proofs . . . 10
  - 2.1.1 Propositional and First-order Logic . . . 10
  - 2.1.2 Natural Deduction . . . 11
  - 2.1.3 Stylised Proof . . . 15
- 2.2 Reasoning about Programs . . . 16
  - 2.2.1 Induction . . . 17
  - 2.2.2 Hoare Logic . . . 19
- 2.3 Theorem Provers . . . 20
  - 2.3.1 Automatic Theorem Proving . . . 20
  - 2.3.2 Proof Assistants . . . 22
- 2.4 Logic Teaching Tools . . . 24

II Emmy 26

3 Overview of Emmy 27

4 Emmy's Logic 28
- 4.1 Terms, Functions, and Types . . . 28
  - 4.1.1 The `Atom` Type . . . 30
  - 4.1.2 Integer Terms and Integer Functions . . . 30
  - 4.1.3 Data Structures . . . 30
  - 4.1.4 Typing Rules . . . 31
- 4.2 Formulae . . . 32
  - 4.2.1 Equality . . . 34
  - 4.2.2 Integer Predicates . . . 34
  - 4.2.3 '`if`' Expression . . . 34
  - 4.2.4 Induction Marker . . . 35
- 4.3 Semantics of Emmy's Logic . . . 36
  - 4.3.1 Semantic of Data Structure Constructors . . . 36
- 4.4 Translation into SMT-LIB Logic . . . 38
  - 4.4.1 Translation of Data Structure Constructors . . . 38

5 Emmy Programs **39**
5.1 Declarations . . . 40
5.2 Proofs . . . 40
5.2.1 Proof Steps . . . 40
5.3 Lemmas . . . 46
5.4 Definitions . . . 46
5.4.1 Definition with Cases . . . 47
5.4.2 Induction over Function Definition . . . 48
5.5 Program . . . 52

6 Proof Checking **53**
6.1 Lemmas . . . 53
6.2 Function Definitions . . . 54
6.2.1 Exhaustion Check . . . 54
6.2.2 Translation of Function Definitions . . . 54
6.3 Checking Types . . . 56
6.4 Checking Steps . . . 57
6.4.1 Simple Steps . . . 57
6.4.2 Assumptions . . . 59
6.4.3 Introductions . . . 62
6.4.4 Induction Step . . . 63
6.4.5 Equalities Step . . . 66
6.5 Checking Program . . . 67
6.6 Checking Entailment . . . 68
6.6.1 Checking Entailment Syntactically . . . 68
6.6.2 Checking Entailment Using an SMT Solver . . . 68

7 Implementation **76**
7.1 Proof Checker . . . 77
7.1.1 Logic and Program Representation . . . 77
7.1.2 Proof Checking Workflow . . . 79
7.2 Web Interface . . . 79
7.2.1 Technical Details . . . 80
7.2.2 Screenshots of the Web Interface . . . 80
7.3 Server . . . 84
7.4 `#lang emmy` . . . 84

III Evaluation and Conclusion **85**

8 Soundness of Induction **86**
8.1 Induction Over Data Structures . . . 86
8.2 Induction Over Function Definitions . . . 87
8.2.1 Correctness of *FunctionInductionPrinciple* . . . 88
8.2.2 The Original Formula is True if Function Terminates . . . 90
8.3 Unfolding of Induction Markers . . . 90
8.4 Non-terminating Functions . . . 90

9 Evaluation **92**
9.1 Language Support . . . 92
9.2 Proving Power . . . 93
9.2.1 Logic . . . 93
9.2.2 Reasoning about Program . . . 94
9.3 Proving Powers of SMT Solvers . . . 97
9.3.1 Data Structure Properties . . . 97
9.3.2 Induction over Data Structure . . . 97
9.3.3 Entailment between Equivalent Formulae . . . 98
9.4 Non-terminating Functions . . . 99
9.5 User Feedback . . . 100
9.5.1 Procedure . . . 100
9.5.2 Results . . . 100
9.5.3 Improvements Made in Response to User Feedback . . . 101

10 Conclusion **102**
10.1 Future Works . . . 102

IV Bibliography and Appendices **104**

A Translation into SMT-LIB Script **110**

B Theorems **112**
B.1 Logic . . . 112
B.2 Reasoning about Programs . . . 112

C Example Programs for Section 9.3.3 **119**

D Program Language **128**

E Example Proofs **131**
E.1 Propositional Logic . . . 131
E.1.1 `example1.prf` . . . 131
E.2 First-order Logic . . . 133
E.2.1 Non-sorted First-order Logic . . . 133
E.2.2 Stylised Proof in Non-sorted First-order Logic . . . 135
E.2.3 Stylised Proof in Many-sorted First-order Logic . . . 137
E.3 Arithmetics . . . 138
E.4 Induction . . . 138
E.4.1 Over Recursive Data Structures . . . 138
E.4.2 Over Recursive Function Definitions . . . 143

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

Currently, there exists a huge gap between the educational theorem provers used in teaching, and the practical theorem provers used in research and the industry. The educational theorem provers, such as Pandora [15], and Panda [43], are often developed by educational institutions and used as teaching tools in logic courses. While being friendly to newcomers, they usually lack the expressiveness to prove more sophisticated theorems, such as properties of computer programs. Meanwhile, there exists many powerful practical proof tools, including the famous Coq proof assistant [19], and the Isabelle/HOL proof assistant [67]. While these tools are very expressive and are capable of formulating and proving more advanced theorems, they have high barrier of entries, and require considerable amounts of prerequisite knowledge in logic, making them inaccessible to students who have just started learning.

We argue that the lack of more expressive teaching tools hampers the teaching and learning of the reasoning about programs course, which is a mandatory first-year course at the Department of Computing, Imperial College London. Students taking this course have no choice but to write all proofs by hand, which is a tedious and error-prone process. The gap between teaching tools and practical tools also discourage students from learning about the arts of theorem proving and program verification, driving away future contributors to a domain with a lot of potential.

1.2 Objectives

The main objectives of this project is to build a proof tool that addresses the above concerns. Namely, the system should aim to assist the students taking the reasoning about programs course, who have already learnt how to write logic proofs in natural deduction, but do not have the time nor the prerequisite knowledge needed to learn to use a powerful, 'real-world' theorem prover or proof assistant.

This requires the tool to be able to express the proofs that the students might encounter when learning the course, while remaining approachable enough so that the students can become productive using this tool with minimal learning and practice.

1.3 Contribution

The main contributions of this project are:

- We devloped Emmy, a proof assistant that puts an emphasis on easy of learning and low barrier of entry, while having enough power to handle the vast majority of the proofs in the first half of the reasoning about program course.
- We built a web-bases graphical user interface for Emmy, making it easily accessible to users without needing to install anything on their own machines.
- We demonstrated how our proof assistant can be used to aid the learning and teaching of the reasoning about programs course, by showing how to solve questions in the course material using Emmy.

Figure 1.1: A proof oheorem written using the web interface of Emmy.

Chapter 2

Background

2.1 Logic and Proofs

In this section we look at how theorems can be formalised using symbolic logic, and the proof processes with which the theorems can be proven.

2.1.1 Propositional and First-order Logic

The syntax and semantics of propositional and first-order logic described here are based on the lecture notes from the C140 Logic course at Department of Computing, Imperial College [75], and the book *Logic in Computer Science: Modelling and Reasoning about Systems* [55].

Propositional Logic

In propositional logic, we are mainly concerned about the relation between *propositions*, which, as far as we are concerned, can be treated as statements that can be either true or false.

For example, the statements

- It is raining.
- It is sunny.

are all propositions. Whether these proposition are true depends on the interpretation of them. We can represent each of these propositions with a letter. For the rest of this section, we will name the two propositions above R, and S respectively.

From these atomic propositions, we may construct more complicated logic formulae which we may reason about. Each individual proposition is a formula. We may also construct formulae by 'connecting' existing formulae using logic connectives. We can, for example, have formulae like 'It is *not* raining', 'It is raining, *and* it is sunny', and '*If* it is raining, *then* it is *not* sunny'. We can use symbols to represent the 'connection' between the propositions, so that these three propostions can be written as '$\neg R$', '$R \wedge S$', and '$R \rightarrow \neg H$' respectively.

Again, whether these formulae are true depends on the interpretation. We say that a formula is 'satisfiable' if there exists an interpretation that makes the formula true, and we say it is 'valid' if it is always true.

First-order Logic

Propositional logic is useful when reasoning about the relation between propositions. However, it is often impossible to express more complicated statements using propositional logic. Consider this famous argument in the form of syllogism:

- All men are mortal.
- Socrates is a man.
- Therefore, Socrates is mortal.

In propositional logic, we can express neither 'a man', 'all men', 'Socrates', nor 'Socrates is mortal'. First-order logic, also known as predicate logic, solves this problem by introducing variables, constants, predicate symbols, and quantifiers, so that the three statements above can be expressed using the following logic formulae:

- $\forall x.\ [Man(x) \rightarrow Mortal(x)]$
- $Man(socrates)$
- $Mortal(socrates)$

In the above formulae, x is a variable, *socrates* is a constant, while *Man* and *Mortal* are predicate (or relation) symbols, where $Man(x)$ means that 'x is a *man*' and $Mortal(socrates)$ means that '*socrates* is *mortal*'. Predicate symbols take 'arguments' which must be variables or constants. A predicate that takes no arguments is the same as propositions in propositional logic. In the first formula, $\forall$ is the symbol for 'for all', thus $\forall x$ means that '*forall* x'. Therefore, the entire formula can be interpreted as '*for all* x, *if* x is a *man*, then x is *mortal*.'

Apart from '*for all*', we can also express '*there exists*' in first-order logic, using the '$\exists$' symbol. The formula $\exists x.\ Mortal(x)$, for example, can be interpreted as '*there exists* an x, such that x is *mortal*.'

Many-sorted First-order Logic

Many-sorted first-order logic differs from normal, unsorted first-order logic by associating each term with a 'sort'. The sort of a term indicates what kind of object the term is.

We may write, for example,

$$\forall x : \text{Man}.\ Mortal(x)$$

which means that '*for all* x of type Man, x is *Mortal*.'

Many-sorted first-order logic allows some property to be expressed using sorts, which, as can be seen in the above example, allows certain properties to be expressed more succintly. The sorts in many-sorted first-order logic can also be used to express 'types' in typed programming languages.

2.1.2 Natural Deduction

When working with logic, we often want to prove formally the validity of formulae. Proving methods include resolution, which is often employed by automatic tools (see 2.3.1), the use of truth tables, which is often not feasible for large proofs, and various forms of deduction.

Here we are particularly interested in natural deduction, a family of formal systems that allows proofs to be written in a way akin to 'intuitive, informal reasoning' [70]. Natural

deduction is widely taught in introductory logic courses, including the first-year logic course at Imperial College.

The name 'natural deduction' first appeared in Gerhard Gentzen's 1935 article [45] [44]. Gentzen stated that he wished to 'set up a formal system which comes as close as possible to actual reasoning' [44], as opposed to existing systems developed by Frege, Russell, and Hilbert, which he found 'far removed' from the methods used in practice. The result, in his own words, was 'ein Kalkül des natürlichen Schließens', 'a calculus of natural deduction'.

Gentzen presented 'calculus NJ', a natural deduction system for intuitional logic, and 'calculus NK', obtained by adding inference rules to NJ to make it compatible with classical logic. Under Gentzen's system, proofs are written as derivation trees of true formulae, which are derived from some assumptions, using a set of derivation rules.

Gentzen was, however, not the first person to come up with such a system. In 1934, Stanisław Jaśkowski published an article in which he demonstrated how to 'analyze a practical proof by making use of the method of suppositions' [56]. In Jaśkowski's system, proofs are written as sequences of proof steps, which is more akin to the system taught at Imperial College today.

Both Gentzen and Jaśkowski focused on the use of assumptions in their systems [69]. Such an emphasis might reflect the actual methods employed by mathematicians at that time, as Jaśkowski remarked, 'the chief means employed in their method is that of an arbitrary supposition.' [56]

Since mid-20th century, a number of syntactically different 'natural deduction' systems have emerged, many of them containing features that 'stem from' the works of Gentzen and Jaśkowski [69]. We will now discuss the difference between some systems of natural deduction, with regard to our project.

Styles of Natural Deduction

To demonstrate the difference between the various styles of natural deduction, we try to proof theorem 2.1 in the various styles we discuss.

Theorem 2.1. If $P \to Q$, then $\neg Q \to \neg P$.

Gentzen The 'initial formulae', which all sit on the top of the proof tree with no other formulae above, are all assumptions, named by numbers written above them. Some derivation rules use assumptions. For derivations using such rules, the assumptions used must be 'made explicit', which in our case via making the assumption number subscript of the rule name.

Notice that since Gentzen used the symbol '→' in his sequent calculus (see 2.1.2). We therefore use '⊃', the symbol used by Gentzen, instead of '→' to denote implication in this proof.

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overset{2}{\neg Q} \qquad \dfrac{\overset{3}{P} \qquad \overset{1}{P \supset Q}}{Q}\ \supset\text{-E}}{\bot}\ \bot\text{-E}}{\neg P}\ \bot\text{-I}_3}{\neg Q \supset \neg P}\ \supset\text{-I}_2}{(P \supset Q) \supset (\neg Q \supset \neg P)}\ \supset\text{-I}_1$$

□

Gentzen described the two ways in which his system differs from 'actual reasoning' [45]:

1. In actual reasoning, there is 'a linear sequence of utterances' [45], which cannot be represented in a tree-like derivation.
2. In actual reasoning, people are 'accustomed to applying repeatedly a result once it has been obtained' [45]. But in Gentzen's system, a derived formula can only be used once, which often leads to redundant 'branches' in the derivation tree.

Jaśkowski The proof here is written in the style employed in Jaśkowski's 1934 article [56]. An alternative style, using boxes to track the 'lifetime' of assumptions, was mentioned in the footnote of the said article. We do not discuss the alternative style here as it is too similar to our 'box style' natural deduction which we will discuss later.

Notice that Jaśkowski wrote formulae using Polish Notation, which is a way to write logic formulae without brackets [12], named after the nationality of logician Jan Łukasiewicz, of whom Jaśkowski was a student [77]. In Polish notation, the connective is put in front of its arguments. So, for example, '$p \to \neg q$' would be written as '$\to p \neg q$', or '$CpNq$' in the originial notation.

While Polish Notation is not commonly used in logic nowdays, we preserve Jaśkowski's original syntax for the sake of consistency with the original article, but replace the connectives with modern ones for better clarity.

Proof.

$$
\begin{aligned}
&1 \cdot\ S \to pq \\
&1 \cdot 2 \cdot\ S \neg q \\
&1 \cdot 2 \cdot 2 \cdot\ Sp \\
&1 \cdot 2 \cdot 2 \cdot\ q \\
&1 \cdot 2 \cdot\ \neg p \\
&1 \cdot\ \to \neg q \neg p \\
&\ \to \to pq \to \neg q \neg p
\end{aligned}
$$

□

This proof is, in fact, very similar to one of the examples provided in Jaśkowski's article. The letter 'S' in the proof means 'suppose', such that the formula 'Sq' would me 'suppose that q'. The numeric prefix before each step indicates the assumptions under which this step is written. For example, if a step is prefixed by '$1 \cdot 2$', then it means that this step is written under the assumptions '$S \to pq$' and '$S \neg q$'.

'Box' In the 'box' style, the scope of assumptions are tracked using boxes, hence the name. The 'box' style is similar to a system mentioned by Jaśkowski [56] [70]. It is currently the natural deduction method taught in the logic course at Department of Computing, Imperial College [75].

In our 'box' style, theorem 2.1 can be proven as:

1.	$P \to Q$	Assume
2.	$\neg Q$	Assume
3.	P	Assume
4.	Q	$\to$-Elim 1 3
5.	$\bot$	$\bot$-Intro 2 4
6.	$\neg P$	$\neg$-Intro 3 5
7.	$\neg Q \to \neg P$	$\to$-Intro 2 6
8.	$(P \to Q) \to (\neg Q \to \neg P)$	$\to$-Intro 1 7

If the formula we wish to prove is of the form '$P_1 \to \cdots \to P_n \to Q$', we can then formulate the proof as '*given* that $P_1 \cdots P_n$ are true, prove Q'. This can be helpful if there are many 'premises', which would otherwise result in many nested boxes, which we consider unnecessary.

After 'extracting' the outermost premise, we arrive at the following proof:

1.	$P \to Q$	Given
2.	$\neg Q$	Assume
3.	P	Assume
4.	Q	$\to$-Elim 1 3
5.	$\bot$	$\bot$-Intro 2 4
6.	$\neg P$	$\neg$-Intro 3 5
7.	$\neg Q \to \neg P$	$\to$-Intro 2 6

Each step, which is a proven formula, in our proof is indexed by a number. We write the natural deduction rule on the right side of each line, along with the number of the formula used in the inference. In fact, proofs in the 'box' style can be regarded as sequential representations of proofs written in Gentzen's tree style, where the numbers written on the right of each step indicates the subtrees of the derivation tree of the formula proven in that step. However, we do not need to duplicate a subproof if we wish to reuse some proven formulae.

The sequential structure of our 'box' style makes it easy for human to comprehend the proof, while the boxes makes it very easy to track the lifetime of assumptions.

Sequent Calculus

In addition to the two systems of natural deduction, NJ and NK, Gentzen also developed two more generalised deduction systems, LJ and LK, which became what we call 'sequent calculus' nowdays. In the sequent calculus, a 'sequent' is an expression which takes the following form [45]:

$$P_1, \cdots, P_n \to Q_1, \cdots, Q_m$$

The above expression has the same informal meaning as the following formula [45]:

$$(P_1 \wedge \cdots \wedge P_n) \supset (Q_1 \vee \cdots \vee Q_m)$$

Notice that here we use '⊃' to as the implication connective, as Gentzen used '→' in sequents.

Proofs in sequent calculus are, like Gentzen's natural deduction, structured as trees. The 'leaves' of the tree would be 'initial sequents', which are tautologies of the form '$A \rightarrow A$'. We then apply derivation rules to these sequents to obtain a sequent of the form '$\rightarrow B$', which means that B, the theorem we wish to prove, remains true under all situations.

As shown by Gentzen, his system NK, LK, and classical predicate logic ('LHK') are equivalent. He also presented a way to translate NJ into LJ.

2.1.3 Stylised Proof

The 'stylised proof' is a logic proof style taught in the reasoning about program course [29]. Like a proof in the 'box' style natural deduction, a stylised proof consists of a sequence of steps, where each step follows from some previous steps.

We write a stylised proof as:

Proof. **Given:**

(1)	Some statement
(2)	Another statement
(3)	Yet another statement

Show:

Some interesting statement

Proof:

(4)	Some intermediate result	from (1) (2)
(5)	Another intermediate result	from (4) (3)
(6)	Some interesting statement	from (5)

□

Unlike natural deduction, in which formulae can only be proven *syntatically*, using inference rules, in a stylised proof, if we say that 'step x follows from steps y and z', then we are making the following statement about the *semantics* of the steps: 'step x is true if steps y and z are true'. This gives us much more freedom in terms of the structure of the proof: we may omit trivial steps which are otherwise required.

For example, consider the following theorem:

Theorem 2.2. If $A \wedge C$ and $B \wedge D$, then $A \wedge B \wedge C \wedge D$.

A proof in natural deduction would take nine steps:

1.	$A \wedge C$	Given
2.	$B \wedge D$	Given
3.	A	$\wedge$-Elim 1
4.	B	$\wedge$-Elim 2
5.	C	$\wedge$-Elim 1
6.	D	$\wedge$-Elim 2
7.	$C \wedge D$	$\wedge$-Intro 5 6
8.	$B \wedge C \wedge D$	$\wedge$-Intro 4 7
9.	$A \wedge B \wedge C \wedge D$	$\wedge$-Intro 3 8

While in stylised proof it takes only one.

Proof. **Given:**

(1)	$A \wedge C$
(2)	$B \wedge D$

Show:

$A \wedge B \wedge C \wedge D$

Proof:

(3)	$A \wedge B \wedge C \wedge D$	from (1) (2)

□

The above proof is correct because we know $A \wedge C, B \wedge D \models A \wedge B \wedge C \wedge D$. That is, semantically, (3) follows from (1) and (2).

The expressiveness and brevity of stylised proof comes at the obvious price of rigourosity: it is much harder to detect an error in a stylised proof than in a natural deduction proof. The correctness of a proof in natural deduction can be easily checked by human, by checking the application of each derivation rule. However, it is much harder to check the correctness of a step in a stylised proof. It is easy to make mistakes when writing a stylised proof without realising.

2.2 Reasoning about Programs

Logic provides a language for modelling program behaviours and reasoning about them formally [55]. This allows us to prove the correctness of computer programs, using reasoning methods based on logic.

We will then discuss some methods of reasonong, which are taught in the first-year Reasoning about Program course at Imperial College.

2.2.1 Induction

Mathematical Induction

Mathematical induction is a technique widely used when proving properties of natural numbers. Paraphrasing Peano's ninth axiom [68], we can express principle of mathematical induction as the following formula in first-order logic:

$$P(0) \land \forall x \in \mathbb{N}.[P(x) \rightarrow P(x+1)] \rightarrow \forall x \in \mathbb{N}.P(x) \tag{2.1}$$

That is, to prove that property P holds for all natural numbers, we need to prove that 1) P holds for 0 [1], and that 2) for an arbitrary natural number x, if P holds for x, then P holds for $x+1$.

We usually call the first case the base case, the second case the inductive step, and the formula expressed in 2.1 the inductive principle.

It is straightforward to apply mathematical induction when reasoning about functions recursively defined over natural numbers, as the two are structurely similar. In eairly 1960s, John McCarthy presented a method called 'recursion induction', 'for making arguments that in the usual formulations are made by mathematical induction' [60]. He then showed how to use recursion induction to prove the equality of functions [61]. The reasoning course however uses a method more similar to mathematical induction, which can be argued to be more approachable to students who already have the knowledge of mathematical induction.

We can demonstrate mathematical induction using the factorial function, which was used as an example in McCarthy's 1962 paper [60]. Consider the following Haskell function which is supposed to calculate the factorial of a number, and the mathematical definition of the factorial function:

```
-- x >= 0
f :: Int -> Int
f x | x == 0  = 1
    | x > 0   = x * f (x - 1)
```

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ \prod_{i=1}^{x} i & \text{otherwise} \end{cases}$$

We wish to prove that the Haskell function `f`, is equivalent to the factorial function. First, we can define the property $P(x) \equiv$ `f x` $= x!$. Then, we need to prove that $P(x)$ holds for all $x \in \mathbb{N}$.

First, we prove that $P(0)$ holds:

$$\begin{aligned} \texttt{f 0} &= 1 && \text{(From the definition of } \texttt{f}\text{)} \\ &= 0! && \text{(From the definition of factorial)} \end{aligned}$$

Then, we prove that for arbitrary $x \in \mathbb{N}$, $P(x+1)$ holds given $P(x)$ holds.

$$\begin{aligned} \texttt{f (x + 1)} &= (x+1) \times \texttt{f x} && \text{(From the definition of } \texttt{f}\text{)} \\ &= (x+1) \times x! && \text{(From induction hypothesis)} \\ &= (x+1)! && \text{(From the definition of factorial)} \end{aligned}$$

Notice that the above is, in a sense, a stylised proof.

[1] It should be noticed that in Peano's original treatise the first natural number is 1. We are starting from 0 here to adhere to a modern definition of natural numbers.

Structural Induction

Apart from functions that deal with natural numbers, we can also prove properties of functions that operate on recursively defined data structures, using a method called structural induction.

The core idea of structural induction, as described by Burstall [16], is to first prove a property of the most elementary data, then prove the property for more complex data, given that the property is proven for 'all data of lesser complexity'. The structure of a proof is analogous to a proof using mathematical induction: first prove the base case(s), then prove the inductive step(s) using some assumptions.

Consider the following definition of a linked list data structure in Haskell:

```
data List a = Nil
            | Cons a (List a)
```

For the `List` data structure, we can derive the following induction principle:

$$P(\texttt{Nil}) \land \forall \texttt{t} \in \texttt{List a}.\, [P(\texttt{t}) \rightarrow \forall \texttt{v} \in \texttt{a}.\, P(\texttt{Cons v t})] \rightarrow \forall \texttt{l} \in \texttt{List a}.\, P(\texttt{l})$$

Similar to the induction principle for natural numbers, our induction principle for `List` is an implication whose premise is a conjuction of one base case and one inductive step. For more complicated data structures, there might be more cases that we need to prove. Consider a tree-like data structure:

```
data Tree a b = Leaf a
              | Flower b
              | Branch (Tree a b) (Tree a b)
              | Trunk a (Tree a b)
```

An induction principle for `Tree a`, with two base cases and two inductive steps, would be:

$$\begin{aligned}
&\forall \texttt{l} \in \texttt{a}.\, P(\texttt{Leaf l}) \\
&\land \forall \texttt{f} \in \texttt{b}.\, P(\texttt{Flower f}) \\
&\land \forall \texttt{t1}, \texttt{t2} \in \texttt{Tree a b}.\, [P(\texttt{t1}) \land P(\texttt{t2}) \rightarrow P(\texttt{Branch t1 t2})] \\
&\land \forall \texttt{t} \in \texttt{Tree a b}.\, [P(\texttt{t}) \rightarrow \forall \texttt{l} \in \texttt{a}.\, P(\texttt{Trunk l t})] \\
&\qquad \rightarrow \forall \texttt{t} \in \texttt{Tree a b}.\, P(\texttt{t})
\end{aligned}$$

While we mainly focus on the reasoning of functional programs using induction, similar strategies can also be applied to recursive functions and data structures in imperative languages.

Zeno

We often need to manually extract induction principles from the definitions of recursive data structures, either to use them in hand-written proofs, or to supply them to automatic theorem provers that does not support induction (see 2.3.1).

Zeno is a tool that automatically generates proofs of functions over recursively defined data structures [79]. Zeno can generate induction principles for definitions of recursive functions and recursively defined data structures. as well as auxilary lemmas if needed in a

proof. Compared to other comparable provers, Zeno has stronger proving power, and similar performance.

The internal language of Zeno is, a modified version of the internal language of the Glasgow Haskell Compiler (GHC), to represent recursive data structures and functions. For example, natural number and the 'less than' relation can be expressed in HC as following:

```
data  Nat = Zero | Succ  Nat

letrec  (<=) = \x -> \y -> case <lq1> x of
  { Zero -> True; Succ x' -> case <lq2> y of
    { Zero -> False; Succ y' -> x' <= y' } }
```

Zeno's representation of recursive functions, which allows easy extration of induction principles, is of interest to our project, as we plan to implement inductive reasoning in our proof system.

2.2.2 Hoare Logic

Programs written in imperative programming languages pose significant challenge to reasoning, as the outcome of each operation in such programs depends on the implicit state of the program at the moment the operation is carried out. The state of the program, which may, for example, describe the value of each variable at the moment, can also be manipulated as the program executes. We must make the state of the program explicit in order to be able to reason about imperative programs.

One way to formalise the state of the program, as introduced by Tony Hoare, under the influence of the work of Robert W. Floyd [81], is to 'state the connection precondition (P), a program (Q) and a description of the result of its execution (R)' [51], using the notation shown in 2.2, which is now recognised as a 'Hoare Triple':

$$P\ \{Q\}\ R \tag{2.2}$$

The above Hoare Triple states that 'if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion' [51]. In other words, R describes the state of the program after Q is executed when the state of the program is described by P.

Consider a imperative programming language with integer arithmetics [2] and variable assignment, where $x := \epsilon$ means assign the value of the expression ϵ to variable x, we can then construct the Hoare Triple 2.3, which states that 'if $x > 10$ and $y > 10$, then after executing $z := x + y$, $z > 20$.

$$x > 10 \wedge y > 10\ \{z := x + y\}\ z > 20 \tag{2.3}$$

If the precondition is irrelevant then we can also replace it with 'true'.

$$\text{true}\ \{x := 10\}\ x = 10 \tag{2.4}$$

Hoare also proposed rules for dealing with conditionals and loops, and later, rules regarding functions [18]. Together, these rules form the basis of the second half of the reasoning of programs course.

[2] It is certainly possible in real life that such a programming language may suffer from integer overflow when adding very large numbers, or, in the extreme case where the numbers are too large to fit in the computer's memory, crash even if the language supports arbitrarily large numbers. We might need to assume that we are considering the purely theoretical semanticas of the language, without bothering how a real-life implementation of the language may go wrong, for now.

Dafny

Dafny is an imperative programming language and a program verifier. [59], with support for recursive functions, classes, inductive data types [76]. In Dafny, code specification can be written within the program code, making it easy to clearly express properties of imperative programs.

Hoare logic can be easily expressed in Dafny. For example, the two Hoare triples shown above can be expressed in Dafny as:

```
assert x > 10 && y > 10;
var z: int := x + y;
assert z > 20;
```

```
assert true;
var x: int := 10;
assert x == 10;
```

Dafny also supports more advanced constructs in Hoare logic, such as pre- and post-conditions of functions, and loop variants and invariants. There is also support for inductive reasoning.

Dafny programs are verified automatically, by first translating the program into Boogie 2 [5] an intermediate representation used in verification, and then checked using an automatic prover (see 2.3.1). Programmers only needs to state the assertions, pre- and post-conditions, loop invariants and other properties of the code when writing the code. They do not need to perform the actual proving themselves.

While we do not implement imperative reasoning, the data structure axioms, which are used by Dafny, [59], are of interest to us.

2.3 Theorem Provers

There already exists many tools for constructing, searching, automating, and verifying proofs. As noted by the authors of Edinburgh LCF, a theorem prover developed in the 1970s, there exists two 'extreme sytles of doing proofs on a computer' [49]:

Automatic theorem proving The user only provides the goal and some axioms, and the system will search for a proof with little or no user input.

Proof checking The user provides a proof, the system merely checks the correctness of the supplied proof.

We will look at some concepts related to theorem proving, and discuss how they relate to our project.

2.3.1 Automatic Theorem Proving

Automatic theorem proving can be regarded as 'algorithmic methods for proving theorems' [42]. When using an automatic theorem prover, the users need to formulate the theorem they wish to prove in a format, such as conjunctive normal form, as stipulated by the prover. The prover will then find a proof for the theorem automatically.

A widely-used strategy of automatic theorem proving is SAT solving, which refers to determining the satisifiability of logic formulae. A notable SAT solving algorithm is the Davis-Putnam algorithm [23], which first applies Herbrand's theorem to 'reduce' first-order clauses to propositional ones [17], and then apply propositional resolution to determine the satisifiability. Davis-Putnam algorithm also has some modern derivatives, such as Chaff

[63]. There also exists alternative methods such as tableaux, which can work with a more diverse set of logic theorems.

While convenient for research and industrial purposes, the 'black-box' nature of automatic theorem provers makes them unsuitable for use in introductory courses, where the students are expected to understand the proving process. In addition, the algorithms used in automatic theorem proving are very different from the methods taught in logic and reasoning courses, making them unaccessible to students with in these courses.

However, it is possible to use automatic theorem proving techniques in proof verification tools to For example, to allow proofs to be checked automatically using automatic provers [2] [37].

SMT Solving

Satisfiability modulo theories (SMT) is about determining the satisfiability of first-order formulae with regard to a certain set of background theories [65]. The set of background theories might include arithmetics, arrays, lists, strings, and even recursive data structures [64], making it possible to efficiently express properties of programs, which is difficult in more traditional theorem provers based on SAT solving. Thus, SMT solving can be used for not only basic SAT solving, but also model checking, and program verification [24]. A noticable use of SMT in automatic reasoning is Boogie, an 'intermediate verification language' that is used in the translation of program written in a higher-level language, such as Dafny [59], into a form that can be checked using theorem provers, which, by default, is Z3, an SMT solver [1].

Many modern SMT solvers, including Z3 [65], CVC4 [10], and Yices [36], implement the SMT-LIB standard, which specifies the syntax and semantics of the SMT-LIB language and related logic theories[8], such as arithmetics, bit vectors, quantifiers, and, in more recent versions, data structures [6]. The SMT-LIB language provides a way to formulate theories, as well as to interact with the solvers. The language's syntax is based on LISP's S-expressions [62], making it very easy for other programs to construct formulae in the language, and to parse the output from a solver.

In our project, an SMT solver can be used to check the correctness of proof steps. For example, consider the following proof:

1.	$A \wedge B$	Given
2.	$C \wedge D$	Given
3.	$A \wedge D$	By 1 2

We wish to check the correctness of step 3, that is, to check whether or not the formulae obtained in steps 1 and 2 lead to the formula in step 3. We can formulate this question as checking the validity of the formula $(A \wedge B) \wedge (C \wedge D) \rightarrow A \wedge D$.

To prove that a propositional formula is valid using an SMT (or SAT) solver, we can instead 'refute' the negation of that formula [23], that is, to prove that the negation of that formula is not satiafiable. We now need to determine the satisfiability of the formula $\neg\big((A \wedge B) \wedge (C \wedge D) \rightarrow A \wedge D\big)$, which can be checked using the following SMT-LIB program:

```
;; Declare all propositions we are going to use
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
```

```
(declare-const D Bool)

;; The formula
(assert (not (to (and (and A B) (and C D)) (and A D))))

;; Tell the solver to check the satisfiability of the assertions
(check-sat)
```

After running the program, The SMT solver would give us the output 'unsat', telling us that $\neg\big((A \land B) \land (C \land D) \to A \land D\big)$ is not satisfiable, hence step 3 is correct.

If we use only natural deduction rules in this proof, then we would have two extra steps, which we consider unnecessary for an immediately obvious proof. The use of an SMT solver allows the omission of such proof steps, making proofs more concise, without sacrificing the correctness of proofs.

2.3.2 Proof Assistants

Proof assistants, which are computer tools that help their users to write proofs, sit somewhere in between the two extremes. They often require their users to write detailed proofs, but still offer some degree of automation to make the process easier.

Popular proof assistants include Coq [19], HOL/Isabelle [66], and Agda [14]. Proof assistants usually set their basis upon some foundamental logic theories. Coq is based on the calculus of inductive constructions [11] a dependent type system; Agda is also based on dependent types, and is in fact a programming language that can be executed; and Isabelle is based on Higher-order logic (HOL).

Due to the interactive nature of proof assistants, users often operate proof assistants either through an IDE, such as the Coq IDE.[19] and Isabelle/jEdit [82] or through a text editor enhanced with a plugin for the proof assistant, such as agda-mode, a plugin for Emacs [38]. There is also Proof General, which is a 'generic tool' that supports multiple theorem provers [3].

Tactics

Many proof assistants provide a set of 'tactics', which are instructions that tells the system how to construct a proof. For example, if we want to prove '$A \to B$' in natural deduction, our first reaction would be 'let us make an assumption!' We can then tell our proof assistant to 'make an assumption', without elaborating what assumptions it should make. Upon receiving such an instruction, the proof assistant will then look at '$A \to B$', the goal of our proof, and decide what assumption to make based on the form of our goal. In this case, the proof assistant will make the assumption A, and change the goal of the proof to B. That is, to prove $A \to B$, we must prove B under the assumption that A is true. Other than that, we may also tell the proof assistant to 'apply the $\land$-introduction rule', or even 'apply the induction principle of natural numbers'.

Tactic systems can be considered as a way to automate the proving process, because they allow the users of proof assistants to omit the details of each step when writing proofs, and focus on the 'strategy' [49] of finding a proof. Some proof assistants also offer tactics that constructs proofs fully automatically, such as `auto` tactic in Coq [19]. There have also been attempts to use external SAT and SMT solvers (see 2.3.1) to implement automation tactics in Coq [37] [2].

However, since tactics often involve the transformation of assumptions and goals, it is quite mentally burdensome to keep track of the state of proof. The finished proof is also somehow opaque, as the intermediate states are not obvious without the use of an IDE. Tactics systems also make proof assistants harder to learn, as the use must now learn the tactic language in addition to a language used for proofs, which may be too demanding [26]. The steep learning curve makes such proof assistants unsuitable for an introductory course.

Example: Prove Theorem 2.1 Using Coq

We take Coq as an example, as it is very popular, and has a mature tactic system [26]. When writing a proof, the Coq IDE will display a list of current subgoals to be proved, and a list of assumptions that have been made. The Coq IDE also allows users to 'step' through their proof, to see how the use of an individual command changes the assumptions and subgoals.

In Coq [19], theorem 2.1 can be expressed as the following Theorem:

```
Theorem transposition (p: Prop) (q: Prop): (p -> q) -> (~q -> ~p).
```

The Coq IDE will then say

```
1 subgoals
p : Prop
q : Prop
_______________________________________(1/1)
(p -> q) -> ~ p -> ~ q
```

which tells us that there is one 'subgoal' we need to prove, which is our theorem, written below the horizontal line.

To prove the theorem, we need to first write '`Proof.`', and write the body of our proof, using Coq tactics, below '`Proof.`'.

```
Proof.
  intros.
```

Here, we have written an incomplete proof using the `intros` tactic. `intros` introduces new assumptions automatically, In our case, the new assumptions introduced will be '`p -> q`' (which means '$p \to q$'), and '`~q`' (which means '$\neg q$').

Coq IDE will then give the following output:

```
1 subgoals
p : Prop
q : Prop
H : p -> q
H0 : ~ q
_______________________________________(1/1)
~ p
```

The newly introduced assumptions are placed above the horizontal line. The formula `~p` (which means '$\neg p$') at below the horizontal line is the current subgoal we need to prove.

The full proof would be:

```
Theorem transposition (p: Prop) (q: Prop): (p -> q) -> (~q -> ~p).
Proof.
  intros.
  contradict H0.
  apply H.
  assumption.
Qed.
```

Once Coq tells us that there are '`No more subgoals`', we know that we have proven everything we need to prove.

2.4 Logic Teaching Tools

A number of teaching tools have been developed in order to aid the teaching of logic such as Pandora [15], and Panda [43]. Many of them are developed by universities and used in their logic courses.

Such logic teaching tools often facilitate construction of proofs in a certain style, such as natural deduction [15] and can check or even grade proofs automatically [83]. Some tools also offer tutorials and guide to students. However, these tools often have limited proving power. They tend to implement only proving of propositional and first-order logic theories, which is only enough for introductory logic courses, but not sufficient for reasoning about programs.

Pandora

Proof Assistant for Natural Deduction using Organised Rectangular Areas (Pandora) is a tool for learning first-order natural deduction written in Java [15]. The tool is developed and used at Imperial College and expresses natural deduction in the sequential 'box' style (see 2.1.2). Pandora provides a graphical user interface for constructing proofs, a built-in tutor system, and the capability to work 'backwards' from the goal. It does not support reasoning about programs nor automatic proof checking, although both features were mentioned as possible extensions in the original paper.

As a Java GUI application, Pandora depends on the Java Runtime which impedes its usage on students' machines. The Java Applet version of Pandora, which could be accessed through a web browser, is also no longer easily accessible as most modern browsers have removed support for Java Applets [54], highlighting the need for educational tools to adapt to the changes in UI technology.

iProve

iProve is a web-based proof assistant designed for teaching logic [50]. It supports propositional and first-order logic, as well as the reasoning of basic arithmetics. iProve provides an online interface for building proofs using natural deduction using the 'box' style (see 2.1.2). Proofs in iProve are checked using Z3 (see 2.3.1), allowing intermediate steps to be omitted. There is currently no support for reasoning about programs or induction of any kind. It also supports usage of lemmas to streamline the proving process.

iProve was implemented as a modern web application, using the React JavaScript library [71]. It requires no local installation, but needs an internet connection to run. The users write proof steps on the web interface. The proof steps are then parsed and checked by the Z3 solver in the backend.

The Little Prover, *the Little Typer*, J-Bob, and Pie

The Little Prover is a book about proving properties of programs [41]. It is written by Daniel P. Friedman and Carl Eastlund. Friedmand has previously co-authored a series of introductory books on programming in LISP and Scheme, whose titles all begin with 'the Little'. Books in this series are structured as series of dialogues between two people, and introduces concepts as the dialogue progresses. The result is an engaging and humorous tone and a very gentle learning curve.

The Little Prover introduces J-Bob, a proof assistant written for this book [47], named after J Moore and Bob Boyer, two pioneers of theorem provers. J-Bob is implemented in Scheme and ACL2 [57]. Despite its relative simplicity, J-Bob is capable of proving all theorems presented in the book. The book also provides a brief overview of other proof assistants for interested readers to explore.

Also co-authered by Friedman, *the Little Typer* is an introductory book on dependent types [40]. Following the same format as Friedman's previous books, the book introduces concepts aboud dependent types and shows how dependent types can be used to ensure the correctness of programs.

Programs in *the Little Typer* are written in a Pie, a dependently typed teaching language designed specifically for this book[48]. Implemented as a `#lang` language in Racket [25], Pie can be installed using Racket's package manager, and requires Racket to run.

Part II

Emmy

Chapter 3

Overview of Emmy

Emmy is a proof assistant designed for teaching purposes. Users may express logic theorems in Emmy, write proofs of the theorems using Emmy's language, and ask Emmy to check the correctness of their proofs.

Users can access Emmy via its online interface, where they can write and check proofs, or use the command line application of Emmy.

The logic of Emmy is a variant of many-sorted first-order logic, capable of expressing many important programming language features, including recursive functions and data structures. This means that many common and useful properties about computer programs can be formulated and expressed using Emmy's logic. We develop the syntax of Emmy's logic in chapter 4.

Emmy supports a structured variant of stylised proofs. This makes proofs extremely flexible in Emmy: the user can produce a very detailed proof, writing down every step in their reasoning process, or they can write one single step to prove a complicated theorem, and let Emmy to ensure that the proof is indeed correct. We describe the syntax of proofs and supporting elements, such as function definitions and lemmas, in chapter 5.

To know whether or not a proof is correct, we developed an algorithmic definition of the correctness of proofs. The full description, along with a formal definition, of the proof checking algorithm is available in chapter 9.3.

Proof checking in Emmy relies heavily on external SMT solvers, which are tools that can check the satisfiability of logic formulae automatically. This is why it is possible to write correct proofs in Emmy without writing down every single step in the proof. The SMT solvers we use implement the SMT-LIB Standard [8], which specifies the syntax and semantics of the logics the SMT solvers support. By default, Emmy uses the Z3 theorem prover [65], but it also supports the CVC4 theorem prover [10]. We describe how we use SMT solvers in section 6.6.2, and the power and limitations of SMT solvers in section 9.3.

We implemented Emmy as a computer program, using the Racket language, and developed a web interface for accessing Emmy, written in JavaScript. We describe Emmy's implementation in chapter 7.

Chapter 4

Emmy's Logic

Emmy uses a variant of many-sorted first-order logic, with quantifiers, uninterpreted functions, and data structures, to express properties of programs. A large range of properties regarding simple functional programs such as:

Theorem 4.1. *reverse* is its own inverse function

$$\forall l : \texttt{List}.\ reverse(reverse(l)) = l$$

can be expressed easily and clearly in Emmy's logic.

In this chapter, we give a detailed, formal description of the abstract syntax of our logic. We then give a brief description of the semantics of our logic, and a detailed description of our treatment of inductive data structures.

4.1 Terms, Functions, and Types

Informally, a *term* denotes an object, which could be a person, a tree, or an integer. A *function* is a mapping from one or more terms to a term. For example, we may use the function symbol *age* to represent a mapping from a person to that person's age, or use + to map two integers to their sum.

Each term has an associated *type* [1], which denotes what kind of object it is. For example, if we use the term x to denote an integer in a mathematical expression, then x has type `Int`. Functions also have types, which determines what should the types of their arguments be, and what is the type that is being mapped to by the function. The *age* function, for example, takes a term of type `Person`, and gives a term of type `Int`.

Formally, we adapt the definition of terms taught in the logics course [75], and define terms as:

Definition 4.1. Term formation

Terms are formed according to the following rules:

- A *constant* with a type is a term. We consider integers to be constants, so integers are terms as well, with `Int` as their types.

[1] 'Type' here is the same as 'sort' in many-sorted logic. We chose the name 'type', instead of using the established name 'sort', to reduce friction when expressing expressions in typed function programming in our logic.

- A *variable* with a type is a term.
- If f is a function that accepts terms of type $\tau_1, \cdots, \tau_n$ as its arguments, and $t_1, \cdots, t_n$ are terms with types $\tau_1, \cdots, \tau_n$, then $f(t_1, \cdots, t_n)$ is a term.

Terms formed according to the above rules are called *well-formed terms*.

Constants and variables are called *atomic terms* as they do not contain any subterms.

Notice that constants and variables must have types, otherwise they are not terms. A function must also have a type. A function must be applied to the correct number of arguments with correct types, in order to form a function application term.

In our logic, functions are *not* terms, which might be quite confusing for users who are used to functional programming, where functions can be treated just like any other term. This is due to the limitations in the underlying logic system, namely, the variant of many-sorted first-order logic specified by the SMT-LIB Standard. Many other features that exists in functional programming, including currying (partial application), and higher-order functions, are also absent from our system.

In the future, we may add support for higher-order functions by defunctionalisation, which we describe in section 10.1.

The meaning of each term and function is dependent on the *structure* of the program, which is an interpretation of each term as an object in the underlying *universe*. In Emmy, we do not bother much about the meaning of terms. We only allow users to *declare* terms, but not to *define* them. Hence all terms, except integers, whose treatment are described below, are uninterpreted.

Typing Statements and Contexts

We use the notation '$t : \tau$' to denote the statement 'term t has type τ', '$f : \tau_1 \cdots \tau_n \to \sigma$' to denote 'function f takes arguments of types $\tau_1 \cdots \tau_n$ and gives a term of type σ', and '$\tau :: \texttt{Type}$' to denote 'τ is a type.

Statements like the ones described above are called *typing statements*. The *context* is a set of typing statements. When working out the type of a term, we consult the context to find statements that could be used to determine the type of that term. The content of the context might change as variables may be introduced by quantifiers. We refer to the context under which the current operation is performed as the *current context*.

We denote a context by Γ.

Term Substitution

Term substitution means replacing one term with another. It can be defined formally as:

Definition 4.2. Term substitution

We denote 'substitude x by y in term t' by $[y/x]t$, where x, y can be any term.

$$\begin{aligned} [y/x]t &\triangleq y && \text{When } t = x \\ [y/x]t &\triangleq t && \text{When } t \neq x \\ [y/x]f(t_1, \cdots, t_n) &= f([y/x]t_1, \cdots, [y/x]t_n) \end{aligned}$$

We use term substitution when we check formula equivalence, which is in turn performed when checking steps, and generating induction principles.

We denote a single substitution using the letter s.

Sometimes, we have a sequence S of substitutions $(s_1, \cdots, s_n)$. The semantics of applying a sequence ot substitutions to a term is as below:

Definition 4.3. Semantics of sequence of substitutions

$$S\ t \triangleq s_1\left(s_2\left(\cdots(s_n\ t)\right)\right)$$

where $S = (s_1, s_2, \cdots, s_n)$.

4.1.1 The `Atom` Type

In order to encode unsorted first-order logic into our system, we provide a type called '`Atom`'. Every constant and variable in unsorted first-order logic will be given the type `Atom` when translated into our logic.

`Atom` is declared implicitly by the EMMY. Users do not need to declare `Atom` by themselves.

4.1.2 Integer Terms and Integer Functions

As stated before, all integers are also terms, whose types are `Int`. We also provide addition, subtraction, and multiplication as functions, which all have the type `Int Int` → `Int`. The `Int` type, as well as the operations, are all defined implicitly, so the users do not need to define them.

We do not provide division because integer division is not total: $\frac{3}{4}$ is not an integer, and $\frac{x}{0}$ is undefined.

4.1.3 Data Structures

A *data structure* is a type whose members can be 'constructed' using a set of *constructors*. A constructor of a data structure δ can be either

An atomic constructor which is a constant of type δ.

A constructor function which takes some arguments and returns a term of type δ.

If a constructor function takes terms of type δ as arguments, then it is also an *inductive constructor*. Atomic constructors, and constructor functions that are not inductive, are called *base constructors*.

We denote atomic constructors using a, and other constructors using c.

We require that a data structure must have at least one base constructor. Otherwise, the data structure is not well-founded, and we may not reason about it inductively. We will discuss this requirement in detail in section 4.3.1.

We also require names of all constructors to be unique.

We use the typing statement

$$\delta :: \{a_1, \cdots, a_n, c_1 : \tau_{1_1} \cdots \tau_{1_{n_1}}, \cdots, c_m : \tau_{m_1} \cdots \tau_{m_{n_m}}\}$$

to mean that

- δ is a data structure.
- $a_1, \cdots, a_n$ are atomic constructors of δ.
- $c_1, \cdots, c_m$ are constructors functions of δ, where c_i takes n_i arguments of types $\tau_{i_1}, \cdots, \tau_{i_{n_i}}$.

We often denote the set of constructors by C, and write $\delta :: C$ to mean that 'δ has constructors C'.

Constructors are the same as other constants and functions for term formation purposes. However, there are additional rules that restricts the ways in which constructors are interpreted, which we describe in detail in 4.3.1.

4.1.4 Typing Rules

We developed a system of typing ruels for determining the type of a specific term, as defined below:

Definition 4.4. Term typing

There exists the following typing rules:

$$\frac{\tau :: \texttt{Type} \in \Gamma}{\Gamma \vdash \tau :: \texttt{Type}}\ \textsc{K-Axiom} \qquad \frac{\delta :: \{\cdots\} \in \Gamma}{\Gamma \vdash \delta :: \texttt{Type}}\ \textsc{K-Data-Axiom}$$

$$\frac{\Gamma \vdash \tau :: \texttt{Type} \qquad t : \tau \in \Gamma}{\Gamma \vdash t : \tau}\ \textsc{T-Axiom} \qquad \frac{f : \tau_1 \cdots \tau_n \to \sigma \in \Gamma}{\Gamma \vdash f : \tau_1 \cdots \tau_n \to \sigma}\ \textsc{T-Fun}$$

$$\frac{\Gamma \vdash \delta :: \texttt{Type} \qquad \delta :: C \in \Gamma \qquad a \in C}{\Gamma \vdash a : \delta}\ \textsc{T-Cons-Const}$$

$$\frac{\Gamma \vdash \delta :: \texttt{Type} \qquad \delta :: C \in \Gamma \qquad c : \tau_1 \cdots \tau_n \in C}{\Gamma \vdash c : \tau_1 \cdots \tau_n \to \delta}\ \textsc{T-Cons-Fun}$$

$$\frac{\Gamma \vdash f : \tau_1 \cdots \tau_n \to \sigma \qquad \Gamma \vdash_{i \in [1,n]} t_i : \tau_i}{\Gamma \vdash f(t_1, \cdots, t_n) : \sigma}\ \textsc{T-App}$$

$$\frac{x \in \mathbb{Z}}{\Gamma \vdash x : \texttt{Int}}\ \text{T-Int}$$

Where $\Gamma \vdash t : \tau$ means that from the context Γ we can derive the conclusion that term t has type τ.

Term typing rules are also term formation rules. A well-formed term is a term that can be typed, under some context, and *vice versa*.

4.2 Formulae

A *formula* is a statement that is either true or false. It could be a *proposition*, such as 'I am happy'. It can also be the application of a *predicate* that describes the relation between some objects. For example, we can say 'x is greater than 10', where 'greater than' is the predicate that describes the relation between the two terms x and 10. Like a function, a predicates has a type, that determines what are the types of the terms whose relations that the predicate is describing. The simple statements 'true' and 'false' are also formulae, where the first one is always true, and the second one is always false.

From existing formulae, we can construct new formulae using *connectives*, by saying 'I am happy, *and* x is greater than 10', where 'and' is a connective that connects the two formula we have alreaidy seen. *Equality* is also one possible relation between two terms, so 'x is *equal to* 10' is also a formula. We can also *quantify* over variables. For example, we may say 'there exist an integer x, such that x is greater than 10', where x is being quantified. The resultant statements constructes using connectives and quantifiers are also formulae.

Like terms, our definition of formulae is largely identical to the definition of many-sorted first-order logic in the logics course [75]. We define formulae recursively as below:

Definition 4.5. Formula formation
Formulae are formed according to the following rules:

- A proposition P is a formula.
- $\top$ and $\bot$ are formulae.
- If ϕ is a formula, then its negation $\neg\phi$ is a formula.
- If ϕ_1 and ϕ_2 are both formulae, then $\phi_1 \; C \; \phi_2$ is a formula, where C is one of the binary connectives: $\wedge \quad \vee \quad \rightarrow \quad \leftrightarrow \quad \neg$
- If ϕ is a formula, t is a term name, and τ is a type, then $Q \; t : \tau.\phi$ is a formula, where Q is one of the quantifiers: $\forall \quad \exists$
- If P is a predicate that accepts arguments of types $\tau_1, \cdots, \tau_n$, and $t_1, \cdots, t_n$ are terms with types $\tau_1, \cdots, \tau_n$, then $P(t_1, \cdots, t_n)$ is a formula.
- If t_1 and t_2 are both terms of the same type, then $t_1 = t_2$ is a formula. We also consider a formula in this form to be the application of predicate = to arguments t_1, t_2.

Formulae formed according to the above rules are called *well-formed formulae*.

As in many-sorted logic, we require all quantifications to specify the type (sort) of the term quantified. Quantified formulae can be translated from unsorted first-order logic by using `Atom` as the type of all quantified terms, such that

$$\forall x.\phi$$

becomes

$$\forall x : \texttt{Atom}.\phi$$

Notice that the above definition of formula formation mentions term typing statements like 'τ is a type'. The meaning of such statements are defined in the previous section. When

checking whether or not a formula is well-formed, we must also check whether or not the terms it mentions are well formed and have the correct types, and whether or not the types it mentions are types as well.

Term Substitution (in Formulae)

Similar to term substitution for terms, we define term substitution for formulae as below:

Definition 4.6. Term substitution
We denote 'substitute x by y in term t' by $[y/x]t$, where x, y can be any term.

$$
\begin{aligned}
[y/x]\top &\triangleq \top \\
[y/x]\bot &\triangleq \bot \\
[y/x]P &\triangleq P \\
[y/x]P(t_1, \cdots, t_n) &\triangleq P([y/x]t_1, \cdots, [y/x]t_n) \\
[y/x]\neg\phi &\triangleq \neg[y/x]\phi \\
[y/x](\phi_1 \; C \; \phi_2) &\triangleq [y/x]\phi_1 \; C \; [y/x]\phi_2 \\
[y/x]Q\; t : \tau.\phi &\triangleq Q\; t : \tau.\phi && \text{When } x = t \\
[y/x]Q\; t : \tau.\phi &\triangleq Q\; t' : \tau.[y/x][t'/t]\phi && \text{When } y = t, x \neq t \\
[y/x]Q\; t : \tau.\phi &\triangleq Q\; t : \tau.[y/x]\phi && \text{When } y \neq t, x \neq t
\end{aligned}
$$

where C is one of the binary logic connectives, Q is either $\forall$ or $\exists$, and t' is fresh.

Term substituion in formulae is used when checking the equivalence between formulae, and when generating induction principles.

We define application of a sequence of substitutions to a formula in the same manner as definition 4.3.

Equivalence

Informally, two formula are equivalent if we can rename bound variables in one formula so it would become the same as the other. We define formula equivalence formally as:

Definition 4.7. Formula equivalence
The relation $=_\mathrm{F}$, 'formula equivalence', between two formulae, is defined as:

$$
\begin{gathered}
\top =_\mathrm{F} \top \\
\bot =_\mathrm{F} \bot \\
P =_\mathrm{F} P \\
P(t_1, \cdots, t_n) =_\mathrm{F} P(t'_1, \cdots, t'_n) \leftrightarrow \forall i \in [1, n]. t_i = t'_i \\
\neg\phi =_\mathrm{F} \neg\phi' \leftrightarrow \phi =_\mathrm{F} \phi' \\
\phi_1 \; C \; \phi_2 =_\mathrm{F} \phi'_1 \; C \; \phi'_2 \leftrightarrow \phi_1 =_\mathrm{F} \phi'_1 \wedge \phi_2 =_\mathrm{F} \phi'_2 \\
Q\; t : \tau.\phi =_\mathrm{F} Q\; t' : \tau.\phi' \leftrightarrow [t'/t]\phi =_\mathrm{F} \phi'
\end{gathered}
$$

where C is one of the binary logic connectives, and Q is either $\forall$ or $\exists$.

From the definition, we obtain the following theorem:

Theorem 4.2. Formula equivalence is weaker than logic equivalence
If $\phi_1 =_F \phi_2$ is true, then $\phi_1 \leftrightarrow \phi_2$ is true.
However, even if $\phi_3 \leftrightarrow \phi_4$, $\phi_3 =_F \phi_4$ is not necessaily true.

The first property can be proven by induction on the definition of $=_F$. The second can be proven by finding an counter-example: $A \land B \leftrightarrow B \land A$, but $A \land B \neq_F B \land A$.

We use $=_F$ when we want to quickly check whether or not two formulae are equal except for the naming of bound variables. We will describe and justify the usage of $=_F$ in detail in section 6.6.1.

4.2.1 Equality

We have a predicate $=$ that stands for the equality of two terms. However, unlike other predicates, which can only accept arguments of certain types, as they are defined, $=$ can accept any pair of terms with the same type.

4.2.2 Integer Predicates

We provide four integer predicates: $<$ 'less than', $>$ 'greater than', $\leq$ 'less than or equal to', and $\geq$ 'greater than or equal to'. They all accept two integers as arguments, and are interpreted in the obvious way.

4.2.3 '`if`' Expression

We define '`if`, a shorthand notation for terms whose values depends on some conditions, as below:

Definition 4.8. Term formation [Extends definition 4.1]
If ϕ_C is a formula, and t_t, t_f are terms of type τ, then $\texttt{if}(\phi_C, t_t, t_f)$ is a term of type τ.
A term of this form is called an *`if` expression*.

Which gives the following typing rule:

Definition 4.9. Term typing [Extends definition 4.4]

$$\frac{\tau :: \text{Type} \in \Gamma \qquad \phi_C \text{ is a formula} \qquad \Gamma \vdash t_t : \tau \qquad \Gamma \vdash t_f : \tau}{\Gamma \vdash \texttt{if}(\phi_C, t_t, t_f) : \tau} \ \textsc{If-Expr}$$

Intuitively, the value of $\texttt{if}(\phi_C, t_t, t_f)$ is t_t if ϕ_C is true, and t_f if ϕ_C is false. We do not define the semantics for `if` expressions like this. Instead, we define the following syntactic equivalence:

Definition 4.10. `if` expression
If formula ϕ contains t as a subterm, and

$$t = \texttt{if}(\phi_C, t_t, t_f)$$

for some formula ϕ_C, and terms t_t, t_f of the same type, then ϕ is equivalent to

$$\phi_C \to [t_t/t]\phi \;\wedge\; \neg\phi_C \to [t_f/t]\phi$$

Which leads to the following theorem:

Theorem 4.3. Semantics of `if` expressions

The `if` expression $\texttt{if}(\phi_C, t_t, t_f)$ is equivalent to t_t if ϕ_C is true, otherwise, it is equivalent to t_f.

This is useful in the definition of functions. For example, we can define *abs*, which return the absolute value, by the following formula:

$$\forall n : \texttt{Int}.abs(n) = \texttt{if}(n < 0, -n, n)$$

which is equivalent to

$$\forall n : \texttt{Int}.\Big[\big(n < 0 \to abs(n) = -n\big) \;\wedge\; \big(\neg n < 0 \to n\big)\Big]$$

Notice that by definition 4.10, nested 'if' expressions are also valid, and can be used to define functions with more than two cases, such as fib, which returns the xth Fibonacci number:

$$\begin{aligned}\forall x : \texttt{Int}.\big[&\neg x < 0 \to fib(x) = \\ &\texttt{if}(x = 0, 0, \texttt{if}(x = 1, 1, fib(x-1) + fib(x-2))\big]\end{aligned}$$

which is equivalent to

$$\begin{aligned}\forall x : \texttt{Int}.\Big[&\neg x < 0 \to \\ &\big(x = 0 \to fib(x) = 0\big) \;\wedge \\ &\big(\neg x = 0 \to fib(x) = \texttt{if}(x = 1, 1, fib(x-1) + fib(x-2))\big)\Big]\end{aligned}$$

and

$$\begin{aligned}\forall x : \texttt{Int}.\Big[&\neg x < 0 \to \\ &\big(x = 0 \to fib(x) = 0\big) \;\wedge \\ &\Big(\neg x = 0 \to \\ &\quad \big(x = 1 \to fib(x) = 1\big) \;\wedge \\ &\quad \big(\neg x = 1 \to fib(x) = fib(x-1) + fib(x-2)\big)\Big)\Big]\end{aligned}$$

4.2.4 Induction Marker

Sometimes, some formula ϕ cannot be proven directly by the underlying SMT solver, but can be proven if we perform induction on ϕ, and break it down into an *inductive principle* ϕ', which entails ϕ.

We need a way to mark these formulae, so that the tool knows where induction should be performed. We thus define *induction markers* as below:

Definition 4.11. Formula formation [Extending definition 4.5]
If ϕ is a formula, then $\mathfrak{I}(\phi)$ is a formula.
We call such formulae *induction markers*.

If we have some formula ϕ, and we want to prove it by induction, then we may write $\mathfrak{I}(\phi)$ instead of ϕ, and the induction will be performed when $\mathfrak{I}(\phi)$ is checked.

We will define induction in definition 4.14. We will also define an $Unfold$ function, that transforms formulae in induction markers to corresponding induction principles, in definition 6.31.

We treat $\mathfrak{I}(\phi)$ and ϕ as the same in definitions. That is, if we have defined something for ϕ, then the same definition also applies to $\mathfrak{I}(\phi)$. The only exception is the $Unfold$ function, which explicitly deals with induction markers.

We also extend the definition of $=_{\mathrm{F}}$ by:

Definition 4.12. Formula equivalence [Extends definition 4.7]

$$\mathfrak{I}(\phi) =_{\mathrm{F}} \phi$$

4.3 Semantics of EMMY's Logic

Like normal many-sorted first-order logic, the semantics of our logic depends on an underlying *universe*, which contains numerous *objects*. The meaning of a statement (formula) in our logic depends on how it is *interpreted* in the underlying universe. Generally, under a certain interpretation, each term in our logic maps to one of the objects in the universe, each function, applied to correct arguments, returns one of these objects as well, and each predicate is maps to a relation between objects of corresponding sorts in the universe.

We do not define the universe and the interpretation by ourselve. In EMMYEMMYfunctions [2], and predicates can only be declared, not defined, thus their interpretation are not specified in our logic. Rather, we depend on the SMT solver underlying our system for constructing the universe and interpreting terms and formulae.

4.3.1 Semantic of Data Structure Constructors

While we treat constructors as functions and terms for the purpose of term formation, we must impose some restrictions on the ways in which they are interpreted. Consider the declarations

$$\begin{aligned} \texttt{Nat} &:: \texttt{Type} \\ z &: \texttt{Nat} \\ s &: \texttt{Nat} \to \texttt{Nat} \end{aligned}$$

whice declares the type of natural numbers, the natural number zero, and the successor function, that is supposed to map each natural number to the 'next' one.

[2] such that $\phi = P(t_1, \cdots, t_n)$, and We do have function definition, but it is only a syntactic sugar that gets translated into a logic assertion.

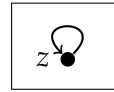
The problem with the above declarations is that they say nothing about the interpretation of term z and function s. We could intreprete them in, for example, the universe depicted by figure 4.1 a, where z is the only natural number, and the 'successor' function merely maps it back to itself. Similarly, if we have define a tree data structure as above, we may have 'branch' constructors that maps subtrees back to themselves, *et cetra*.

A set of axioms is thus needed to ensure that a certain data structure is interpreted as we expected. If we add Peano's sixth and seventh axioms [68]:

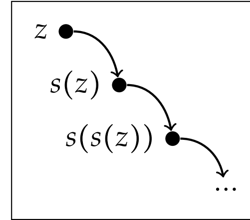
$$\forall x : \texttt{Nat}.\forall y : \texttt{Nat}.[x = y \leftrightarrow s(x) = s(y)] \quad (6)$$

$$\forall x : \texttt{Nat}.[s(x) \neq z] \quad (7)$$

to the above definitions, then we would obtain a universe like the one in figure 4.1 b, which is correct.



(a) A lonely universe.



(b) A universe with infinitely many natural numbers.

Figure 4.1: Two universes

Axioms like these are needed for all data structure constructors. We require that constructors abide the five properties proposed by the authors of Dafny [59], namely,

1. Constructors are injective.
2. Different constructors produces different values.
3. All data structure values are constructed using constructors.
4. Data structure values are partially ordered.
5. The order is well-founded.

The first three properties ensures that constructors in our logic work in the same way constructors in functional programming languages work. The last two are required for us to perform structural induction on them.

We define the order as below:

Definition 4.13. Data structure order
If the constructor c of data structure δ takes arguments of type $\tau_1, \cdots, \tau_n$, and $t_1, \cdots, t_n$ are terms of corresponding types, then for all $i \in [1, n]$, if $\tau_i = \delta$, then $t_i < c(t_1, \cdots, t_n)$.

It is obvious that the base constructors construct the minimal elements in the order, since they take no arguments of the type of the data structure, they cannot construct a term that is greater than any other term. Since we require all data structures to have at least one base constructors, the order is indeed well-founded for all data structures.

The well-founded ordering of a certain data structure is closely related to well-founded structural induction over that data structure: the minimal elements in the ordering, which are the non-inductive constructors, form the base cases in the induction, while inductive constructors form the inductive steps. The well-foundedness of the ordering is crucial: we rely on this property in our proof of the correctness of induction, in section 8.1.

Since we have defined the well-founded order for all data structures, we can perform well-founded induction over them. The *induction principle* for an induction over a certain quantification over a data structure can thus be defined as:

Definition 4.14. Data structure induction principle

For a context Γ such that $\Gamma \vdash \delta :: C$, that is, the data structure δ has C as its set of constructors, and formula ϕ, we define $InductionPrinciple$, the function that maps a quantification over a data structure to its induction principle, as:

$$InductionPrinciple(\Gamma, \forall t : \delta.\phi) \triangleq$$
$$\bigwedge_{a \in C} [a/t]\phi \wedge \bigwedge_{c:\tau_1 \cdots \tau_n \in C} \Big[\forall t_1 : \tau_1. \cdots \forall t_n : \tau_n. \big[\wedge_{i \in [1,n]} (\tau_1 = \delta \rightarrow [t_i/t]\phi) \rightarrow [c(t_1, \cdots, t_n)/t]\phi\big]\Big]$$

We will see how induction principles are used in section 6.4.4 ,and 6.6.2.

4.4 Translation into SMT-LIB Logic

Since we use SMT-LIB compliant solvers to check the proofs, we need to develop a way to translate terms and formulae in out logic into corresponding terms in SMT-LIB logic. Translation of types, functions, and constants are mostly straightforward: types are added to the signature as sorts, functions and their type are translated into *ranked function symbols*, which are function symbols associated with sequences of sort symbols [8, p. 46], and added to the signature, and constants are added to the signature as *ranked variables*, which are variable symbols associated with their sorts [8, p. 46].

In SMT-LIB logic, formulae are terms of **Bool** sort. Our propositions are thus translated to constants of **Bool** sort, and predicates becomes functions from the sorts of their arguments to **Bool**. We do not need to translate connectives as they are included in SMT-LIB theories we are using.

We define the translation formally in definitions 6.32, 6.33, and 6.34.

4.4.1 Translation of Data Structure Constructors

One way to translate data structure constructors is to translate the constructors to normal terms and functions, and generate appropriate axioms for them. This approach is employed by Dafny: its authors proposed five data structure properties, which we described in section 4.3.1, and Dafny axiomatises the first, second, and fourth properties, while achieving the fifth (well-foundedness of odering) by 'enforcing' the 'stratification' of data structures.

However, we choose to delegate this task to the underlying SMT solver, by directly declaring the data structures when consulting the SMT solvers, so that the solver can enforce these properties for us. We do it this way for the following reasons:

- The SMT solvers already have ways of ensuring the correct construction of recursive data structures. Doing it by ourself by extracting all axioms add unnecessary complexity to our proof-checking procedure.
- The SMT solvers underlying our system do not seem to handle instantiation of uninterpreted sorts (types) well, causing them to fail when working with quantified recursive data structures defined using terms and functions, even when axioms are supplied.

Chapter 5

Emmy Programs

Proofs in Emmy are written in a variant of the 'stylised' proof format, discussed in section 2.1.3: there is a sequence of steps, where each step follows from some previous steps or facts. The semantics of our proofs are similar to stylised proofs. However, our proofs are more structured. The lifetimes of assumptions and introduced variables are syntactically indicated, which is more similar to 'box' style proofs.

For example, the following proof of theorem 4.1

Proof. **Show:**

$\forall l : \texttt{List}.\ reverse(reverse(l)) = l$

Proof:

By induction.

Base Case Show $reverse(reverse(nil)) = nil$

Proof: By definitions of *reverse* and *append*.

Inductive Step Take some $l : \texttt{List}$, such that $reverse(reverse(l)) = l$. Take arbitrary $x : \texttt{Integer}$. Show $reverse(reverse(cons(x, l))) = cons(x, l)$.

Proof: By definitions of *reverse* and *append*.

□

can be written in the abstract syntax of Emmy as

$$
\begin{array}{l}
\langle \forall l : \texttt{List}.\ reverse(reverse(l)) = l, \\
\quad (\mathfrak{I}\langle \texttt{L}, \forall l : \texttt{List}.\ reverse(reverse(l)) = l, \\
\qquad (\langle (), \\
\qquad\quad reverse(reverse(nil)) = nil, \\
\qquad\quad (\$\langle 1, reverse(reverse(nil)) = nil, \{\texttt{r-n}, \texttt{r-c}, \texttt{app-n}, \texttt{app-c}\}\rangle)\rangle, \\
\qquad\ \langle (\langle l : \texttt{List}, reverse(reverse(l)) = l\rangle, \langle x : \texttt{Int}, \top\rangle), \\
\qquad\quad reverse(reverse(cons(x, l))) = cons(x, l), \\
\qquad\quad (\$\langle 2, reverse(reverse(cons(x, l))) = cons(x, l), \{\texttt{r-n}, \texttt{r-c}, \texttt{app-n}, \texttt{app-c}\}\rangle)\rangle)\rangle)\rangle
\end{array}
$$

whose structure corresponds to the structure of the stylised proof we have written above.

Apart from proofs, we also need ways to define functions like *reverse*, declare data structures like `List`, and provide useful lemmas when needed. The combination of all the proofs, declarations, definitions, and lemmas, is called a *program*.

In this chapter, we describe how programs are represented in EMMY. We specify the abstract syntax of EMMY programs, and describe the relation of elements in our programs to other proof systems and programming languages.

We will give a full description of the semantics of the programs, that is, how is the correctness of proofs determined, in the next chapter. We have also included the concrete syntax of EMMY in appendix D.

5.1 Declarations

Types, functions, predicates, and constants must be *declared* before they can be used in proofs. The declarations of types, functions, predicates, and constants are typing statements, denoted using notations introduced in the previous chapter.

Declarations are written in *declaration sections*, so they cannot be interleaved with proof steps. Variables, which are introduced by quantifiers, are 'declared' by the respective quantifications.

Formally, we define declaration section as:

Definition 5.1. Declaration section
A declaration section is a sequence of typing statements.

Declarations made in declaration sections are added to the context when checking proof steps.

5.2 Proofs

Intuitively, a *proof* is a sequence of *steps* that leads to a result, which, ideally, is our goal. It represents the process through which a goal an be obtained, where each step represents one 'milestone' in the proving process.

Formally, we define a proof as:

Definition 5.2. Proof
A *proof* is a pair

$$\langle \phi_G, \Pi \rangle$$

where

- ϕ_G is the *goal* of the proof, which is a formula whose validity we are trying to prove.
- Π is a sequence of proof steps. The formula in the last step should be the formula of the goal.

5.2.1 Proof Steps

Proof steps are 'tagged' products that mark 'steps' in the proving procedure. We label the type of each step by putting $\mathbb{BLACKBOARD}$ $\mathbb{BOLD}$ letters in front of the step.

Simple Step

A simple step is a step where an intermediate formula is proven to follow from some previous steps. For example, if we have

1.	$A \wedge B$	Given
2.	A	By 1
3.	$A \vee C$	By 2

then 'A By 1' would be a simple step in which we prove that the intermediate result A follows from step 1.

We give a number to each step, so that a step may be referenced by other steps in the same proof as the other steps' justification.

Formally, we define a simple step as:

Definition 5.3. Simple step

A simple proof step is a tagged triple

$$\mathbb{S}\langle i, \phi, J\rangle$$

where

- i is the *number* of the step.
- ϕ is the *formula* that is proven true in this step.
- J is a *justification* of this step. It is either
  - `Given`
  - `Tentative`
  - A set of step numbers.

The step number i of a step is not necessarily the same as the index of the step in the sequence containing it. We require that the step number i is unique within each *proof*.

We use ϕ_i to denote the formula in step i, and J_i to denote justification for step i. The step with number i can be similarly denoted as σ_i.

Assumption

We often make assumptions when proving. In the following proof, we assume $A \wedge B$, and prove that A follows from $A \wedge B$, hence we obtain $A \wedge B \rightarrow B$ at last:

1.	$A \wedge B$	Assumed
2.	A	By 1
3.	$A \wedge B \rightarrow A$	By 2

We put everything in the box in the above proof into an *assumption* step, including the assumption we made, and the *subproof* that is proved under that assumption. We define such a step as below:

Definition 5.4. Assumption Step

An assumption step is a tagged triple

$$\mathbb{A}\langle i, \phi, \Pi \rangle$$

where

- i is the *number* of the step.
- ϕ is the *formula* that is being assumed.
- Π is the *subproof* made under the assumption that ϕ is true. It is a sequence of steps which can use ϕ as a justification.

Similarly, we can denote the subproof in step i as Π_i.

Variable Introduction

When working with first-order proofs, we often introduce *variables* to be used in subsequent steps. We introduce the variable by giving it a name and state its type. We also give it a scope, or lifetime, in which it is declared, which is represented by a box in 'box' style natural deduction.

A variable introduction and the subproof that involves the introduced variable is defined as below:

Definition 5.5. Introduction step

An introduction step is a tagged triple

$$\mathbb{I}\langle i, t : \tau, \Pi \rangle$$

where

- i is the *number* of the step.
- $t : \tau$ is the *declaration* of the introduced variable, where t is the name of the variable, and τ is its type.
- Π is the *subproof* within which the variable is available.

Sometimes, we also make an assumption about the variable introduced, by saying 'take an integer x, such that $x > 10$'. Such step is defined as:

Definition 5.6. Introduction with assumption step

An introduction step with assumption is a quadruple

$$\mathbb{IA}\langle i, t : \tau, \phi, \Pi \rangle$$

where

- i is the *number* of the step.

- $t : \tau$ is the *declaration* of the introduced variable, where t is the name of the variable, and τ is its type.
- ϕ is the *formula* that is being assumed, it can contain free t.
- Π is the *subproof* within which the variable is available.

It can contain t.

Induction Steps

When we perform induction over either a recursively declared data structure, or a recursive function definition, we wish to structure our proof in a way that resembles the induction principle. We want to state our goal, list all base cases and their proofs, as well as all inductive steps, their induction hypotheses, and their proofs.

We define *induction step* to reflect the structure of induction:

Definition 5.7. Induction Step and induction cases

An induction step is either a tagged triple

$$\Im\langle i, \phi, C\rangle$$

that performs induction over a data structure, or a tagged quadruple

$$\Im\langle i, i_d, \phi, C\rangle$$

that performs induction over a function definition, where

- i is the *number* of the step.
- i_d, if exists, is the *name of the function definition* we want to perform induction on.
- ϕ is the *formula* that is being proven by induction.
- C is a sequence of *induction cases*.

where an *induction case* is a triple

$$\langle H, \phi_G, \Pi\rangle$$

where

- H is a sequence of *induction hypotheses* made for this case.
- ϕ_G is the *goal* proven in this case.
- Π is the *subproof* under the assumptions.

where an *induction hypothesis* is a pair

$$\langle t : \tau, \phi_H \rangle$$

where

- $t : \tau$ is the *declaration* of a variable introduced for this induction case.
- ϕ_H is an *assumption* we make about the introduced variable.

If we do not make any assumptions about a variable introduced in a case, we just trivially assume $\top$.

Each induction step is one single unit that proves an inductive property. Each case should only prove one base case or inductive step in the induction principle.

Notice that in order for the proof checking algorithm defined in section 6.4.4 to be correct, the goal must satisfy certain properties:

- If the step performs induction over a data structure, a valid goal must be a universal quantification over a data structure. That is, the goal must have the form $\forall t : \delta.\phi$, where δ is a data structure.
- If the step performs induction over a function definition, a valid goal must be a sequence of universal quantifications and implications with an induction marker inside. That is, a valid goal must be either
  - $\mathfrak{I}(\phi)$, an induction marker, or
  - $\forall t : \tau.\phi_G$, where ϕ_G is also a valid goal, or
  - $\phi \rightarrow \phi_G$, where ϕ_G is also a valid goal.

Equalities Steps

Very often, we write stylised proofs with 'chained' equalites that looks like:

Proof.

(1)	$t_0 = t_1$	from R_1
(2)	$= t_2$	from R_2
(3)	$= t_3$	from R_3
	$\vdots$	
(n)	$= t_n$	from R_n

□

in which we prove that $t = t_1$, and that $t_1 = t_2$, and so on, and eventually we prove $t_{n-1} = t_n$, which, by transitivity, proves that $t = t_n$. It is equivalent to the following stylised proof:

Proof.

(1)	$t_0 = t_1$	from R_1

(2)	$t_0 = t_2$	from R_2 (1)
(3)	$t_0 = t_3$	from R_3 (2)
	$\vdots$	
(n)	$t_0 = t_n$	from R_n $(n-1)$

□

To formulate it in our syntax, it would become something like:

$$\begin{aligned}(&\mathbb{S}\langle 1, t_0 = t_1, \{R_1\}\rangle,\\ &\mathbb{S}\langle 2, t_0 = t_2, \{R_2\ 1\}\rangle,\\ &\mathbb{S}\langle 3, t_0 = t_3, \{R_3\ 2\}\rangle,\\ &\vdots\\ &\mathbb{S}\langle n, t_0 = t_n, \{R_n\ (n-1)\}\rangle)\end{aligned}$$

which is very cumbersome, especially when written in the concrete syntax.

Therefore, we define a shorthand notation that proves a chain of equalities in one single step, as:

Definition 5.8. Equalities step

An equalities step is a tagged triple

$$\mathbb{E}\langle i, t_0, E\rangle$$

where

- i is the *number* of the step.
- t_0 is the *initial term*. We will show that the initial term is equal to some other term in this step.
- E is a sequence of *equalities*.

Where an *equality* is a pair

$$\langle t_i, J_i\rangle$$

where

- t_i is the *term* in the ith equality.
- J_i is the *justification* for the ith equality.

Using an equalities step, we may write the above proof as one step

$$(\mathbb{E}\langle 1, t_0, (\langle t_1, \{R_1\}\rangle, \langle t_2, \{R_2\}\rangle, \langle t_3, \{R_3\}\rangle, \cdots, \langle t_n, \{R_n\}\rangle)\rangle)$$

in which we prove that $t_0 = t_n$.

5.3 Lemmas

A *lemma* is a theorem that is useful when proving other theorems. Proving the lemmas each time we need them is tiresome, so we allow users to 'declare' a set of lemmas, which can be used in proofs without having to be proved. Thus, a lemma is similar to a simple step, except that a lemma does not need any justification: it is assumed to be true.

A single lemma is defined as:

Definition 5.9. Lemma
A *lemma* is a pair

$$\langle i, \phi \rangle$$

where

- i is the *number* of the lemma.
- ϕ is the formula of the lemma.

We require that the number of lemmas to be unique across the entire program, and that no step has the same number as a lemma.

And a lemma section is defined as:

Definition 5.10. Lemma section
A *lemma section* is a sequence of lemmas.

We can treat a lemma section as a syntatic sugar: declaring some lemmas is the same as putting all lemmas at the beginning of each proof, as steps that are '`Given`' to be true.

5.4 Definitions

In declarations, functions are declared, not defined, meaning that we do not impose any restriction on the interpretation of functions when we declare them. We now define *function definition*, a way to give the functions meanings, as:

Definition 5.11. Function definition
A *function definition* is a triple

$$\langle i, t_l, r \rangle$$

where

- i is the *number* of this definition.
- t_l is the *left-hand-side* (LHS) of the definition.
- The term r is the *right-hand-side* (RHS) of the definition.

We require all variables that appear free in r to also appear free in t_l. We also require all variables in t_l to be distinct from all other variables anywhere in the program.

We also require all terms and their subterms on the LHS to be either variables, constants, or function applications, and that all but the outermost function application on the LHS to be constructor functions applied to arguments.

The requirement that variable names are distinct is needed to prevent shadowing of variable names. In the implementation of the proof checker, we replace all variables on the LHSs by fresh variables before using the function definitions in checking proofs.

The reason for our second requirement is to make function definitions in out language similar to function definitions in Haskell.

For example, the following definitions of function s:

```
s x = x + 1
```

$$s(x) = x + 1$$

can be written as:

$$\langle \texttt{f-def}, \texttt{s-def}, s(x), x + 1 \rangle$$

We then define the definition section:

Definition 5.12. Definition section
A *definition section* is a sequence of definitions.

Definitions are semantically similar to lemmas: they are theorems whose truth we take for granted.

5.4.1 Definition with Cases

In a programming language, we may define a function as below:

```
f x :: Int -> Int
f x | x == -1    = 0
    | x == 0     = 1
    | otherwise  = 2
```

The definition of function definitions cannot define anything equivalent to the above definition of `f`. We must provide a way to express different cases:

Definition 5.13. Function definition [Amends definition 5.11]
r, the *right-hand-side* of the definition, should be a either a term, or a non-empty sequence of pairs $\langle \phi_C, t_b \rangle$, where each pair is called a *case*. The formula ϕ_C in each case is the *condition* of that case, and the term t_b is the *body* of that case.
We require that all variables that appears free in r to appear free in t_l.
We require that the cases are *exhaustive*, which means that the disjunction of all cases' conditions is valid.

The requirement that cases are exhaustive means that for all argument values, there is at least one case whose condition should be true. We develop a formal definition for case exhaustion in section 6.2.1, along with a procedure to check the exhaustion of cases.

Using cases, the function `f` can be defined as:

$$\langle \texttt{f-1}, f(x), (\langle x = -1, 0 \rangle$$

$$\langle x = 0, 1\rangle$$
$$\langle \top, 2\rangle)\rangle$$

which means that when $x = -1$, $f(x) = 0$, if $x = 0$, $f(x) = 1$, and if none of the above conditions are met, then $f(x) = 2$. The above definition is complete because $\top$, the condition of the last case, will always be true, therefore there will always be a case whose condition is true.

In the concrete syntax, we allow '`otherwise`' as an alias for '$\top$' as a case's condition, because in Haskell, `otherwise` is defined to be the same as `True` [21].

Notice that having exhaustive cases does not mean that the function returns a result for all possible arguments. For example, the following Haskell function

```
endless x :: Int -> Int
endless x | x < 0       = 0
          | otherwise   = 1 + endless (x + 1)
```

has exhaustive cases by our definition, but does not terminate when applied to a non-negative integer. We discuss the ramifications of non-terminating functions in section 8.2 and 9.4.

If $f(x)$ does not terminate, we say that $f(x)$ is *undefined*. Any formula that contains $f(x)$ would be *meaningless*.

5.4.2 Induction over Function Definition

When reasoning about the property of recursive functions, we can perform induction over the definition of such functions [27]: a recursive function should have a case without any recursive calls, which serves as our *base case* in the induction, and then the cases with recursive calls become *inductive steps* in the induction. We first prove that the propertu for the base cases, then, in a inductive step, we assume that the property holds for all values returned from recursive calls, and prove the property for that case.

An Example

For example, say we have the following definition of function f:

$$f(x,y) = \begin{cases} 1 & \text{if } x - y > 100 \\ (x+y) * f(x+10, y-10) & \text{otherwise} \end{cases}$$

And we want to prove the property:

$$\forall x : \texttt{Int}.\forall y : \texttt{Int}.P(f(x,y)) \tag{5.1}$$

Induction on integers does not help much. What we can do instead is to prove the following induction principle:

$$\forall x : \texttt{Int}.\forall y : \texttt{Int}.\Big[(x - y > 100 \rightarrow P(1)) \tag{5.2}$$
$$\wedge$$
$$\forall r : \texttt{Int}.\big(x - y \leq 100 \rightarrow r = f(x+10, y-10) \wedge P(x) \rightarrow P((x+y) * r)\big)\Big]$$

which implies

$$\forall x : \texttt{Int}.\forall y : \texttt{Int}.\forall z : \texttt{Int}.[f(x,y) = z \rightarrow P(z)] \tag{5.3}$$

From the definition of f, we know that the value of $x - y$ increases in each recursive call, therefore, f always terminates, and we have

$$\forall x : \texttt{Int}.\forall y : \texttt{Int}.\exists z : \texttt{Int}.f(x,y) = z \tag{5.4}$$

From 5.3 and 5.4, we could prove 5.1, the original property. Notice that if f does not terminate, then 5.1 cannot be proven.

For non-terminating functions, the situation is more tricky. Consider function g:

$$g(x) = \begin{cases} 0 & \text{if } x < 0 \\ g(x+1) & \text{otherwise} \end{cases}$$

And property:

$$\forall x : \texttt{Int}.Q(g(x)) \tag{5.5}$$

Then, the induction principle would be:

$$\forall x : \texttt{Int}.\big[(x < 0 \rightarrow Q(0)) \wedge \forall r : \texttt{Int}.[x \geq 0 \quad \wedge \quad r = g(x+1) \rightarrow Q(r)]\big] \tag{5.6}$$

which implies

$$\forall x : \texttt{Int}.\forall r : \texttt{Int}.[r = g(x) \rightarrow Q(r)] \tag{5.7}$$

However, property 5.7 does not imply 5.5, because for some values of x, $g(x)$ is undefined, and for such values, $Q(x)$ is meaningless.

In general, induction over function definitions proves some formula that looks like

$$\forall t_1 : \tau_1.\cdots\forall t_n : \tau_n.\forall r : \sigma.\big[f(t_1,\cdots,t_n) = r \rightarrow R(t_1,\cdots,t_n,r)\big] \tag{5.8}$$

instead of

$$\forall t_1 : \tau_1.\cdots\forall t_n : \tau_n.R(t_1,\cdots,t_n,f(t_1,\cdots,t_n)) \tag{5.9}$$

which is stronger than 5.8 [27]. Notice that Formula 5.9 follows from 5.8 if f terminates.

Generation of Induction Principle

To begin, we define $Calls_T$, which extracts all calls to a certain function in terms, and $Calls_F$, which extracts calls from formulae, as:

Definition 5.14. Extraction of calls from terms and formulae

$$\begin{aligned}
Calls_T(f,c) &\triangleq \varnothing && \text{if } c \text{ is a constant}\\
Calls_T(f,c) &\triangleq \varnothing && \text{if } v \text{ is a variable}\\
Calls_T(f,f'(t_1,\cdots,t_n)) &\triangleq \{f'(t_1,\cdots,t_n)\} && \text{if } f = f'\\
Calls_T(f,f'(t_1,\cdots,t_n)) &\triangleq \bigcup_{i\in[1,n]} Calls_T(f,t_i) && \text{if } f \neq f'
\end{aligned}$$

$$\begin{aligned}
Calls_F(f,P) &\triangleq \varnothing && P \text{ is a proposition}\\
Calls_F(f,\neg\phi) &\triangleq Calls_F(f,\phi) && \\
Calls_F(f,\phi_1 \; C \; \phi_2) &\triangleq Calls_F(f,\phi_1) \cup Calls_F(f,\phi_2) && C \text{ is a binary connective}\\
Calls_F(f,Q\; t : \tau\; \phi) &\triangleq Calls_F(f,\phi) && Q \text{ is either } \forall \text{ or } \exists
\end{aligned}$$

$$Calls_F(f, P(t_1, \cdots, t_n)) \triangleq \bigcup_{i \in [1,n]} Calls_T(f, t_i)$$
$$Calls_F(f, t_1 = t_2) \triangleq Calls_T(f, t_1) \cup Calls_T(f, t_2)$$

$Calls_T(f, t)$ returns the set of all unique calls of f in term t, and $Calls_F(f, \phi)$ returns all unique calls of f in formula ϕ. For example, we have:

$$Calls_F(f, f(x) + g(x) < f(x) + f(10 + x)) = \{f(x), f(10 + x)\}$$

Notice that if the same function call occurs twict in a formula, $Call_F$ will only return one, as there is only one unique call.

We use $Call_F(f, \phi)$ to find the arguments supplied to calls of f in ϕ. We substitute the arguments when generating the induction hypotheses.

We then define the function *FunctionInductionPrinciple*, which generates the induction principle for a formula from the definition of a function, as:

Definition 5.15. Function Induction Principle

$$FunctionInductionPrinciple(\Gamma, \langle i, f(t_1, \cdots, t_n), C\rangle, \phi) \triangleq \bigwedge_{i \in [1, |C|]} CaseInductionPrinciple(\Gamma, C_i)$$

where C, the cases in the function definition, is a sequence of at least two function cases, $|C|$ is its length, and C_i is the ith case.

We define *CaseInductionPrinciple* as:

$$CaseInductionPrinciple(\Gamma, \langle \phi_i, b_i \rangle) \triangleq$$
$$S_\phi\ \phi_i \land \bigwedge_{j \in [1, i-1]} (\neg S_\phi\ \phi_j) \ \rightarrow\ [S_\phi\ b_i / t_\phi]\phi \qquad \text{If } |Calls_T(f, b_i)| = 0$$
$$CaseInductionPrinciple(\Gamma, \langle \phi_i, b_i \rangle) \triangleq$$
$$\forall r : \sigma. \big[S_\phi\ \phi_i \land \bigwedge_{j \in [1, i-1]} (\neg S_\phi\ \phi_j) \qquad \text{If } |Calls_T(f, b_i)| = 1$$
$$\rightarrow\ (S_b\ [r/t_\phi]\ \phi \land r = S_b\ t_\phi \rightarrow [S_\phi\ [r/t_r]\ b_i / t_\phi]\ \phi)\big]$$

where

- $\Gamma \vdash f : \tau_1 \cdots \tau_n \rightarrow \sigma$.
- r is fresh.
- $\{t_\phi\} = \{f(t'_1, \cdots, t'_n)\} = Calls_F(f, \phi)$.
- $\{t_r\} = \{f(t''_1, \cdots, t''_n)\} = Calls_T(f, b_i)$.

and S_ϕ and S_b are sequences of substitutions such that

$$S_\phi = ([t'_1/t_1], \cdots, [t'_n/t_n])$$
$$S_b = ([S_\phi\ t''_1 / S_\phi\ t_1], \cdots, [S_\phi\ t''_n / S_\phi\ t_n])$$

We require that $|Calls_F(f, \phi)| = 1$.

We only define $FunctionInductionPrinciple$ for function definitions in which no more than one unique recursive call occurs. In addition, we require that exactly one call to the function occurs in the formula, that is, $|Calls_F(f,\phi)| = 1$. This is because when there are multiple calls, the generation of induction principle becomes very complicated, and that if there is no function call in the formula, then there is no point performing induction. By 'unique recursive call', we mean the application of the function over whose definition we perform induction to the same set of arguments. For example, in $f(x) = 1 + f(x-2) + f(x-2)$, there is only one unique recursive call, while in $fib(x) = fib(x-1) + fib(x-2)$, there are two.

Also, we only define $FunctionInductionPrinciple$ for function definitions with cases, as if there are no different cases, then there is either no need for induction because there is no recursive call, or the induction will not be well-founded as there is no base case.

We will prove the soundness of induction over function definitions in section 8.2.

Function Induction Markers

To perform function induction, we can use an induction step. We then need a way to denote the places where we wish to apply the induction principle, and the name of the function definition with regard to which we generate the inductive principle. We thus extend our existing definition of induction markers (definition 4.11) as below:

Definition 5.16. Formula formation [Extending definitions 4.5 and 4.11]
If ϕ is a formula, and i is the number of a function definition, then $\mathfrak{I}^i(\phi)$ is an *induction marker*.

Like the induction markers we defined in definition 4.11, we treat $\mathfrak{I}^i(\phi)$ and ϕ as the same in definitions, excepct in the $Unfold$ function.

The formula ϕ inside the induction marker $\mathfrak{I}^i(\phi)$ is what we apply the induction principle to.

We will discuss when the induction is performed in section 6.6.2.

Example of Function Induction Principle Generation

As an example, consider the following function definition, adapted from proof of theorem B.21:

$$g'(i,j,cnt,acc) = \begin{cases} acc & \text{if } cnt \geq i \\ g'(i,j,1+cnt,j+acc) & \text{otherwise} \end{cases}$$

which can be expressed in our language as:

$$d = \langle \texttt{gp-d}, g'(i,j,cnt,acc), (\langle cnt \geq i, acc\rangle \\ \langle \top, g'(i,j,1+cnt,j+acc)\rangle)\rangle$$

Now, if we want to prove the property

$$\forall x:\texttt{Int}.\forall y:\texttt{Int}.\forall cnt:\texttt{Int}.\forall acc:\texttt{Int}.[\\ P(x,y,cnt) \to g'(x,y,cnt,acc) = (x-cnt)*y+acc] \tag{5.10}$$

under some context Γ that would type g' correctly, we must perform induction on the definition of g': simple induction on integers does not work here.

First, we use an induction marker to mark the part of the formula that we want to prove inductively. In general, we put the induction marker inside all quantifications:

$$\forall x : \texttt{Int}.\forall y : \texttt{Int}.\forall cnt : \texttt{Int}.\forall acc : \texttt{Int}.\Big[\mathfrak{I}^{\texttt{gp-d}}\big(P(x,y,cnt) \to g'(x,y,cnt,acc) = (x - cnt) * y + acc\big)\Big] \tag{5.11}$$

We then generate the induction principle for the formula inside the induction marker.

$$\begin{aligned}
&FunctionInductionPrinciple(\Gamma, d, \phi) = \\
&\quad \big[P(x,y,cnt) \;\to\; (x \geq cnt \;\to\; acc = (x - cnt) * y + acc)\big] \\
&\quad \wedge \\
&\quad \forall r : \texttt{Int}.\Big[\top \wedge \neg cnt \geq i \\
&\qquad \to \\
&\qquad \big(P(x,y,cnt) \to r = (x - (1 + cnt)) * y + (j + acc)\big) \;\wedge\; r = g'(i, j, 1 + cnt, j + acc) \\
&\qquad \to \\
&\qquad \big(P(x,y,cnt) \;\to\; r = (x - cnt) * y + acc\big)\Big]
\end{aligned}$$

What we need to prove now is the two induction cases, whose conjunction implies ϕ.

5.5 Program

We have already defined ‘declare’, ‘proof’, ‘lemma’, and ‘definition’ sections. Now, we may finnally define formally *programs*: they are sequences of sections.

Definition 5.17. Program
A *program* is a sequence of sections, which can be either declare, proof, lemma, or definition sections.

Chapter 6

Proof Checking

In the previous chapter we have described the syntax of EMMY programs. EMMY takes such programs as inputs, and check the correctness of them, or, more specifically, the correctness of the proofs in these programs.

In this chapter, we describe how the correctness of proofs is defined, define the algorithm with which the correctness is checked, and explain how our system utilise external SMT solvers when checking proofs. We also present some possible alternations to our semantics and evaluate their values.

Entailment, Informally

In this chapter, we very often use the notation

$$\Gamma, \Delta, \phi_1, \cdots, \phi_n \vdash \phi_c$$

which is called an *entailment*. Informally, the meaning of the above entailment is

> Under the context Γ, and the function definitions Δ, the formulae $\phi_1, \cdots, \phi_n$ *entails* ϕ_c. That is, if $\phi_1, \cdots, \phi_n$ are all true, then ϕ_c is true.

The formulae $\phi_1, \cdots, \phi_n$, which are to the left of the turnstile ($\vdash$) are the *premises* of the entailment, and ϕ_c, the formula to the right, is the *consequence* of the entailment.

We develop a formal definition of entailment in section 6.6.

We mark everything that is carried out using the SMT solver in blue. If an entailment is marked in blue, then it means that it is being checked using the SMT solver.

6.1 Lemmas

Since lemmas and function definitions are syntatic sugar over normal steps, we translate them into normal steps before checking a proof:

Definition 6.1. Translation of Lemmas

$$Translate(\langle i, \phi \rangle) = \mathbb{S}\langle i, \phi, \texttt{Given} \rangle$$

No checking is performed, as we assume all lemmas to hold.

6.2 Function Definitions

6.2.1 Exhaustion Check

In definition 5.13, we require that cases in a definition to be exhaustive: they must cover all possible cases. If the cases are not exhaustive, then under certain circumstances, conditions of all cases are false, and the value of the function would be undefined.

We define the exhaustion check as below:

Definition 6.2. Exhaustion check

$$Exhaustive(\Gamma, (\langle \phi_1, t_1 \rangle, \cdots, \langle \phi_n, t_n \rangle)) \triangleq \Gamma, \{\} \models \phi_1 \vee \cdots \vee \phi_n$$

where Γ is the current context.

$t_1, \cdots, t_n$, the case bodies, are not relevent here.

That is, if the disjunction of all cases' conditions is valid, then the cases are exhaustive, as it would be impossible for all conditions to be false at the same time. The validity is checked using the entailment checking procedure, which may call the SMT solver. If the cases are not exhaustive, the program should stop and report an error.

Notice that we are checking the entailment with an empty set of definitions in the above definition.

In practice, we can first check if any of the conditions are $\top$, which corresponds to an `otherwise` term in the 'guards' of Haskell's function definitions. The $\top$ case would always be true, so it would 'capture' all situations that are not covered by previous cases, making all situations handled.

6.2.2 Translation of Function Definitions

To begin, we translate the RHS, which may contain multiple cases, to a single term:

Definition 6.3. Translation of RHS

$$TranslateRhs(C) \triangleq TranslateRhsCases(C) \quad \text{When } C \text{ is a sequence of cases}$$
$$TranslateRhs(t) \triangleq t \quad \text{When } t \text{ is a term}$$

where

$$TranslateRhsCases((\langle \phi_C, t_b \rangle, c \cdots)) \triangleq \mathtt{if}(\phi_C, t_b, TranslateRhsCases(c \cdots))$$
$$TranslateRhsCases((\langle \top, t_b \rangle, c \cdots)) \triangleq t_b$$
$$TranslateRhsCases((\langle \phi_C, t_b \rangle)) \triangleq t_b$$

Notice that as soon as we encounter a case with $\top$ as its condition, we throw away all remaining cases. Because if we translated it to

$$\mathtt{if}(\top, t_b, TranslateRhsCases(c \cdots))$$

then by theorem 4.3, it will always be equal to t_b. The rest of the cases are irrelevant.

Also notice that if there is only one case left, no matter what the condition is, we translate the case to its body. We could do this because we know that the cases are exhaustive.

Therefore, either one of the previous cases' condition is met, so the last case is irrelevant, or, if all previous cases' conditions are not met, then the condition of the last case must hold. Otherwise, the cases cannot be exhaustive.

The function *TranslateRhs* yields a term which will sit on the right hand side of the resultant formula. However, we cannot just assert that LHS and RHS are equal, because they both contain free variables whose types are unknown. We need to quantify over these variables at last, but before that, we need to infer the types of these free variables, which is not difficult, as the types of all functions and constructors have already been declared.

The type inference algorithm can be defined as:

Definition 6.4. Type inference

$$
\begin{aligned}
Infer(\Gamma,\tau,v) &\triangleq \{v:\tau\} && \text{when } \neg\exists\sigma.\Gamma \vdash v:\sigma \\
Infer(\Gamma,\tau,c) &\triangleq \varnothing && \text{when } \Gamma \vdash c:\sigma \\
Infer(\Gamma,\tau,f(t_1,\cdots,t_n)) &\triangleq \bigcup_{i\in[1,n]} Infer(\Gamma,\tau_i,t_i) && \text{when } \Gamma \vdash f:\tau_1\cdots\tau_n \to \tau \\
Infer(\Gamma,\tau,\mathtt{if}(\phi_C,t_t,t_f)) &\triangleq Infer(\Gamma,\tau,t_t) \cup Infer(\Gamma,\tau,t_f)
\end{aligned}
$$

The last case (for `if` expressions) is actually not needed, as we prohibit `if` expressions on LHS.

The function $Infer$ takes three arguments: the first is the current context, from which we may get the type of constants, functions, and constructors, the second is the 'expected' type of a term, and the third is the term whose, and whose subterms', types we wish to infer. It returns a set of typing statements, that should tell us about the types of all free variables. $Infer$ is undefined for all other cases, which should be caught as errors in the implementation.

Notice that it is possible for $Infer$ to return conflicting types for the same term:

$$
\begin{aligned}
&Infer(\{f:\tau\ \sigma->\tau\},\tau,f(x,x)) \\
&\quad = Infer(\{f:\tau\ \sigma->\tau\},\tau,x) \cup Infer(\{f:\tau\ \sigma->\tau\},\sigma,x) \\
&\quad = \{x:\tau\} \cup \{x:\sigma\} \\
&\quad = \{x:\tau, x:\sigma\}
\end{aligned}
$$

We therefore define the predicate *ContextCorrect*, to check the context returned by $Infer$:

Definition 6.5. Is a context correct?

$$ContextCorrect(\Gamma) \triangleq x:\tau \in \Gamma \wedge x:\sigma \in \Gamma \to \tau = \sigma$$

That is, a context is correct if each term and function appear in only one typing statement in the context. When inferring types of variables, if the same variable is given more than one type, than the implementation should report an error.

Using the above definitions, we arrive at the procedure for translating function definitions:

Definition 6.6. Translation of function definitions

$$\begin{aligned}&\mathit{Translate}(\langle i, f(t_1,\cdots,t_n), r\rangle) \triangleq \\ &\quad \mathbb{S}\langle i, \forall v_1 : \sigma_1.\cdots.\forall v_m : \sigma_m.f(t_1,\cdots,t_m) = \mathit{TranslateRhs}(r), \texttt{Given}\rangle\end{aligned}$$

where

Γ is the current context.

- $\Gamma \vdash f : \tau_1 \cdots \tau_n \to \tau$
- $\{v_1 : \sigma_1, \cdots, v_m : \sigma_m\} = \mathit{Infer}(\Gamma, \tau, f(t_1, \cdots, t_n)$
- $\mathit{ContextCorrect}(\mathit{Infer}(\Gamma, \tau, f(t_1, \cdots, t_n))$.

Function Call Lemma

From the above definitions and the fact that function definitions must be exhaustive, we may obtain the following conclusion, by induction over function *Translate*:

Lemma 6.1. Function call must fall into one case

If

- $\Gamma \vdash f : \tau_1 \cdots \tau_n \to \sigma$
- $\Gamma \vdash t_i : \tau_i$ for $i \in [1, n]$
- $d = \langle i, f(t_{a1}, \cdots, t_{an})(\langle \phi_1, b_1\rangle, \cdots, \langle \phi_m, b_m\rangle)\rangle$

then there exists some $i \in [1, m]$ such that

$$\begin{aligned}&\mathit{Translate}(d) \to \\ &\quad S\ \phi_i \wedge \bigwedge_{j \in [1, i-1]} (\neg S\ \phi_j) \ \wedge\ f(t_1, \cdots, t_n) = S\ b_i\end{aligned}$$

where $S = ([t_1/t_{a1}], \cdots, [t_n/t_{an}])$.

That is, all possible inputs to a function must satisfy the condition of one case, and must not satisfy the conditions of all previous cases, and the value of the function call would be equal to the body of that case, with all variables substituted.

We will use this lemma when proving the soundness of function induction principles.

6.3 Checking Types

Before a proof is checked, we need to ensure that all formulae in the proof is well-formed. That is, no function is applied to wrong arguments, and no predicate is applied to wrong variables. Our type system consists of rules that determines whether or not something is well-typed, and therefore well-formed (see section 4.4). We perform a type check before checking the proof, and proceeds checking only if no type errors are found.

We also include type declarations into the generated SMT-LIB script. However, we do not use the SMT solver for any type checking purposes, as our own type checking procedure can give much more detailed and specific error messages than what SMT-LIB specified. We expect any program that is approved by our type checker to run without type errors in the SMT solver.

6.4 Checking Steps

We can now develop an algorithm for checking the correctness of steps, that is, to check whether or not the conclusion we draw in a certain step follows from the justifications of this step.

6.4.1 Simple Steps

Intuitively, a step σ is correct under a context Γ if its formula is true given that all formulae in steps referenced in the justification of step σ are true under Γ. That is, the step $\sigma_i = \$\langle i, \phi_i, J_i \rangle$ is correct if $\Gamma, \Delta, \phi_{j_1}, \cdots, \phi_{j_n} \vdash \phi_i$, where $J_i \equiv \{j_1, \cdots, j_n\}$.

However, this simple definition does not take into consideration circular reference between steps. Consider the following steps, which are all nonsensical yet deemed correct under the above definition and an empty context:

$$\sigma_1 = \$\langle 1, A \wedge B, \{1\} \rangle$$
$$\sigma_2 = \$\langle 2, P \to Q, \{3\} \rangle$$
$$\sigma_3 = \$\langle 3, \neg Q \to \neg P, \{2\} \rangle$$

Obviously, there is little point allowing a step to say 'I am true because I am true'. We must therefore have some means of ruling out the use of circular references. A more aggressive approach would be to construct a 'dependency graph' for the step we are checking, where step σ_i 'depends on' σ_j if $j \in J_i$. If we encounter the step we are checking when constructing the graph, then we can declare that step invalid. However, this is computationaly heavy, and would result in many duplicated computations, especially for long, complicated proofs.

The approach used in our implementation is more passive: we maintain a mapping Φ_C from the numbers of yje steps we have already checked to those steps' formulae. We treat Φ_C as a binary relation between the set of possible step numbers and the set of possible formulae, such that for any step number i and formula ϕ, $\langle i, \phi \rangle \in \Phi_C$ if and only if $\Phi_C(i) = \phi$. For example, if we have checked the step $\$\langle 10, A \wedge B, \{9\} \rangle$, then $\langle 10, A \wedge B \rangle \in \Phi_C$, and $\Phi_C(10) = A \wedge B$.

It is obvious that if the step we are currently checking, whose number is by definition not in the domain of Φ_C, only uses steps in the domain of Φ_C as its justifications, then it cannot reference itself. We then add the current step number and formula to Φ_C, and check the next step. This approach is more efficient, but does not accept proofs where a step references a step after it even if no circular references actually exists.

To make lemmas and function definitions accessible to proof steps, we also add all (translated) lemmas and function definitions to Φ_C, before a proof is checked.

Combining handling of circular reference with the definition we purposed in the beginning, we arrive at the following definition of the correctness of a single step:

Definition 6.7. Correctness of a step

$$Correct(\Gamma, \Delta, \Phi_C, \$\langle i, \phi_i, J_i \rangle) =$$
$$\forall j \in J_i.\ j \in dom(\Phi_C) \quad \wedge \quad \Gamma, \Delta, \Phi_C(j_1), \cdots, \Phi_C(j_n) \vdash \phi_i$$

where $J_i \equiv \{j_1, \cdots, j_n\}$, and Γ is the context.

The entailment, marked in blue, is checked using the external SMT solver. We describe in detail how it can be checked in section 6.6.

We then need a way to construct the mapping Φ_C. The obvious way would be to start with an empty set, and add checked steps to the set as we move along the sequence of steps we are checking. The algorithm for checking an entire sequence of steps could thus be defined recursively as below:

Definition 6.8. Correctness of a sequence of steps

$$\begin{aligned} &StepsCorrect(\Gamma, \Delta, \Phi_C, ()) = \top \\ &StepsCorrect(\Gamma, \Delta, \Phi_C, (\mathbb{S}\langle i, \phi_i, J_i\rangle, \sigma' \cdots)) = \\ &\quad Correct(\Gamma, \Delta, \Phi_C, \mathbb{S}\langle i, \phi_i, J_i\rangle) \ \wedge \ StepsCorrect(\Gamma, \Delta, \Phi_C \cup \{\langle i, \phi_i\rangle\}, (\sigma' \cdots)) \end{aligned}$$

Notice that since we require all step numbers to be unique within a proof, no two mapping pairs with the same step number will be added to Φ_C. It is not possible to have $\langle i, \phi_i\rangle, \langle j, \phi_j\rangle \in \Phi_C$ such that $i = j$ and $\phi_i \neq \phi_j$.

Since we do not introduce (declare) any new term, the context is not changed.

This algorithm with return true if all steps are correct, otherwise it returns false.

Formulae `Given` to Be True

In the 'box' style natural deduction, we sometimes mark some steps as '`Given`'. That is, the result obatined in this step is 'given to be true', or 'taken for granted'. This can reduce the levels of assumptions that exist in a proof.

In Emmy, we allow '`Given`' to be the justification of a step. Such steps are deemed correct right away:

Definition 6.9. Correctness of a step [extends definition 6.7]

$$Correct(\Gamma, \Delta, \Phi_C, \mathbb{S}\langle i, \phi_i, \texttt{Given}\rangle) = \top$$

`Tentative` Steps

Sometimes, we know that we need to prove something, but do not know exactly how it can be proven. In this case, we can make this step '`Tentative`'. We have not proven it, but we wish it to be there in the proof, either to give us a hint, or to serve as a justification for following steps. We allow users to write such steps in order to lessen the restriction on how proofs can be written, which is one of the objectives of this project. Like '`Given`' steps, '`Tentative`' steps are always true. We can thus extend the definition 6.7 by:

Definition 6.10. Correctness of a step [extends definition 6.7]

$$Correct(\Gamma, \Delta, \Phi_C, \mathbb{S}\langle i, \phi_i, \texttt{Tentative}\rangle) = \top$$

Extraneous Justifications and Valid Formulae

Notice that since the logic of Emmy is monotonic: having extraneous justifications does not affect the outcome of the step checking algorithm. The sequence of steps

$$\begin{aligned}(&\mathbb{S}\langle 1, A \land B, \texttt{Given}\rangle,\\ &\mathbb{S}\langle 2, C \land D, \texttt{Given}\rangle,\\ &\mathbb{S}\langle 3, A, \{1,2\}\rangle)\end{aligned}$$

is correct, although step 3 does not need step 2 as a justification.

Our algorithm will also happily accept any step whose formula is valid, regardless of the justification of that step. Consider

$$(\mathbb{S}\langle 1, (A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A), \{\}\rangle)$$

which will be deemed correct even though no justification is given.

There does not seem to a simple way to deal with these issues, as invalidating the above proofs would require a change to our entailment-based proof checking algorithm, possibly necessitating the use of inference rules. We therefore do not handle such situations in our proof checker, by leaving these steps as correct steps. Proof checking front-ends may implement 'simplification' features that minimises the number of justifications used, or notify the user that a formula is valid.

6.4.2 Assumptions

A single assumption step consists of an assumption, and a subproof, which is a sequence of steps. We can define the correctness of an assumption step as the correctness of its subproof, under the assumption that the assumed formula is deemed to be true. Thus, the algorithm for checking a single assumption step is simple: we add the assumption to Φ_C, and then check the subproof with regard to the updated Φ_C.

To put this formally, we could extend definition 6.7 by adding a case for assumptions:

Definition 6.11. Correctness of a step [extends definition 6.7]

$$\mathit{Correct}(\Gamma, \Delta, \Phi_C, \mathbb{A}\langle i, \phi_i, \Pi_i\rangle) = \mathit{StepsCorrect}(\Gamma, \Delta, \Phi_C \cup \{\langle i, \phi_i\rangle\}, \Pi_i)$$

Now we need to find a way to 'use' the results obtained in the subproof, under the assumption, in later steps. Recall that in 'box' style natural deduction, we can have the following:

1.	$A \rightarrow B$	Given
2.	$B \rightarrow C$	Given
3.	A	Assume
4.	B	→-Elim 1 3
5.	C	→-Elim 2 4
6.	$A \rightarrow C$	→-Intro 3 5

Step 6 lists step 5, which is in the subproof under the assumption made in step 3, as its justification. Our algorithm for checking a sequence of steps, as in definition 6.8, cannot

handle this situation, because it does not add steps in the subproof of an assumption to Φ_C, rendering those steps inaccesible for later steps. However, we also must not simply throw all steps in a subproof into Φ_C directly. Consider the following sequence of steps:

$$
\begin{aligned}
&(\mathbb{A}\langle 1, A, (\\
&\quad \mathbb{S}\langle 2, A, \{1\}\rangle \\
&)\rangle, \\
&\mathbb{S}\langle 3, A, \{2\}\rangle)
\end{aligned}
$$

If we simply add step 2, which is in the subproof of step 1, into Φ_C after we have checked step 1, then step 3 would become correct, which is not what we want.

To derive a correct way to carry what we have proven in a subproof forward, we may look at how assumptions are handled in natural deduction, from which we derived the style of our proofs. In 'box' style natural deduction, an application of the arrow-introduction rule (→-Intro) looks like:

1.	P	Assume
2.	Q	Obtained
3.	$P \to Q$	→-Intro 1 2

If Q is proven true under the assumption that P is true, then using the arrow-introduction rule, we could derive the implication $P \to Q$. This gives us the first hint on developing our algorithm: when a proven formula is taken out of an subproof, it becomes the consequent in an implication, where the assumption becomes the antecedent.

Also notice that if we have something like:

1.	P	Assume
2.	$\vdots$	Hard work...
3.	Q	Obtained
4.	R	Also obtained
5.	$P \to R$	→-Intro 1 4

then we can also have:

1.	P	Assume
2.	$\vdots$	Hard work...
3.	Q	Obtained
4.	R	Also obtained
5.	$P \to R$	→-Intro 1 4
6.	P	Assume
7.	$\vdots$	The same hard work as above
8.	Q	Obtained, but we don't go another step and obtain R
9.	$P \to Q$	→-Intro 6 8

That is, if we can derive 'assumption implies last formula', then we can also derive 'assumption implies the formula before the last'. By induction, for all formula P' proven in the subproof, we may derive $P \to P'$. This gives us the second hint: any formula proven in a subproof can be taken out. It does not have to be the last one.

Our treatment of assumption steps is thus the following: after an assumption step $\mathbb{A}\langle i, \phi_i, \Pi_i\rangle$ is being checked, for each step $\mathbb{S}\langle j, \phi_j, J_j\rangle$ in the subproof Π_i, we add $\langle j, \phi_i \rightarrow \phi_j\rangle$ to Φ_C, so that the conclusion 'it is proven in step j that ϕ_j is true if ϕ_i is true' becomes accessible to later steps.

The algorithm described above can be formalised as the below addition to definition 6.8:

Definition 6.12. Correctness of a sequence of steps [extends definition 6.8]

$$\begin{aligned}&StepsCorrect(\Gamma, \Delta, \Phi_C, (\mathbb{A}\langle i, \phi_i, \Pi_i\rangle, \sigma' \cdots)) = \\ &\quad Correct(\Gamma, \Delta, \Phi_C, \mathbb{A}\langle i, \phi_i, \Pi_i\rangle) \\ &\quad \land StepsCorrect(\Gamma, \Delta, \Phi_C \cup Substeps(\mathbb{A}\langle i, \phi_i, \Pi_i\rangle), (\sigma' \cdots))\end{aligned}$$

Where the *Substeps* function is defined as:

Definition 6.13. The *Substep* function

$$Substeps(\mathbb{A}\langle i, \phi_i, \Pi_i\rangle) = \{\langle j, \phi_i \rightarrow \phi_j\rangle \mid \mathbb{S}\langle j, \phi_j, J_j\rangle \in \Pi_i\}$$

Assumption within Assumptions

Notice that we only 'extract' conclusions from simple steps in the subproof. If there is another assumption within a subproof, anything proven within that assumption's subproof will not be added to Φ_C.

There is no technical reason for imposing such an limit, as it is easy to recursively add all results from assumptions within assumptions to Φ_C. Rather, the choise is made for usability reasons: first, we do not wish to deviate too much from the syntax of 'box' style natural deduction, which, like all other forms of natural deduction, does not have any rule that eliminates two assumptions at the same time. Second, we are afraid that if we allow more than one level of 'arrow-introduction', then users may try to take things out of very deeply nested assumptions, resulting in long implications, which makes the proof harder to understand.

An alternative algorithm for extracting results from a subproof looks like:

Definition 6.14. The *Substep* function [alternative to 6.13]

$$\begin{aligned}&Substeps(\mathbb{A}\langle i, \phi_i, \Pi_i\rangle) = \\ &\quad \{\langle j, \phi_i \rightarrow \phi_j\rangle \mid \mathbb{S}\langle j, \phi_j, J_j\rangle \in \Pi_i\} \\ &\quad \cup \bigcup_{\mathbb{A}\langle j, \phi_j, \Pi_j\rangle \in \Pi_i} \{\langle k, \phi_i \rightarrow \phi_k\rangle \mid \langle k, \phi_k\rangle \in Substeps(\mathbb{A}\langle j, \phi_j, \Pi_j\rangle)\}\end{aligned}$$

We included this definition in the implementation of Emmy, which can be switched on using the '`--deep`' command line argument.

Reference to Assumption in Justification

Notice that in natural deduction, when performing arrow-introduction, we need to apply the rule to not only the step in which the consequent is proven, but also the step in which the assumption is made, as in the following examples (use of assumption step marked in red):

$$\dfrac{\dfrac{\overset{1}{P}}{Q}\text{ Hard work}}{P \to Q}\ \to\text{-Intro}_{(1)}$$

1.	P	Assume
2.	$\vdots$	Hard work
3.	Q	Obtained
4.	$P \to Q$	→-Intro 1 3

In Gentzen's original style, we need to give each assumption a number and refer to that number when introducing an implication, to indicate that the 'scope' of the assumption is over, or that the assumption is no longer in place after this step. This is necessary as in Gentzen's original style there is no other way to indicate the 'scope' of assumptions. The 'box' style made the 'scope' of assumptions clear, however, the inference rule remains the same, probably to mark the use of assumption explicitly, and to keep consistency with tree-style natural deduction systems.

Since our system does not use any inference rules, we do not require the assumption to be included in the justification, for the sake of simplicity. If we require the users to refer to the assumption whenever they use a result obtained under an assumption, we can replace the definition 6.13 with the following:

Definition 6.15. The *Substeps* function [alternative to 6.13]

$$Substeps(\mathbb{A}\langle i, \phi_i, \Pi_i\rangle) = \{\langle\langle i,j\rangle, \phi_i \to \phi_j\rangle \mid \mathbb{S}\langle j, \phi_j, J_j\rangle \in \Pi_i\}$$

Then, when we need to use, for example, the result obtained in step 10, which is proven under assumption 5, in our justification, we need to include $\langle 5, 10\rangle$ in our justification.

6.4.3 Introductions

Introductions are similar to assumptions: intuitively, the newly introduced term can be seen as the 'assumption' in an introduction. We define the correctness of a single introduction step as the correctness of its subproof. However, before checking its subproof, we do not change Φ_C. Instead, we add the declaration of the newly introduced term into the context.

We extend definition 6.7 by adding a new case:

Definition 6.16. Correctness of a step [extends definition 6.7]

$$Correct(\Gamma, \Delta, \Phi_C, \mathbb{I}\langle i, t : \tau, \Pi_i\rangle) = StepsCorrect(\Gamma, \Delta \cup \{t : \tau\}, \Phi_C, \Pi_i)$$

Now we need to add the steps in the subproof to Φ_C. In the case of assumptions, we say that the formulae proven in steps in the subproof are proven under an assumption. Here, instead of an assumption, we can say that they are proven under the context, which contains an arbitrary term of a certain type.

It now becomes obvious that the formulae proven in the subproof need to be 'wrapped' in 'for all' quantifiers when they are taken out of the subproof. Just like for assumptions, we put implication arrows in front of them. We thus add a case to the definition of *Substeps*:

Definition 6.17. The *Substep* function [extends definition 6.13]

$$Substeps(\mathbb{I}\langle i, t : \tau, \Pi_i \rangle) = \{\langle j, \forall t : \tau.\ \phi_j \rangle \mid \mathbb{S}\langle j, \phi_j, J_j \rangle \in \Pi_i\}$$

We also need to add a new case to *StepsCorrect* accordingly:

Definition 6.18. Correctness of a sequence of steps [extends definition 6.8]

$$\begin{aligned}&StepsCorrect(\Gamma, \Delta, \Phi_C, (\mathbb{I}\langle i, t : \tau, \Pi_i \rangle, \sigma' \cdots)) = \\ &\quad Correct(\Gamma, \Delta, \Phi_C, \mathbb{I}\langle i, t : \tau, \Pi_i \rangle) \\ &\quad \wedge StepsCorrect(\Gamma, \Delta, \Phi_C \cup Substeps(\mathbb{I}\langle i, t : \tau, \Pi_i \rangle), (\sigma' \cdots))\end{aligned}$$

By combining the definitions for assumptions and introductions, we obtain the following definitions for introductions with assumptions:

Definition 6.19. Correctness of a step [extends definition 6.7]

$$\begin{aligned}&Correct(\Gamma, \Delta, \Phi_C, \mathbb{IA}\langle i, t : \tau, \phi_i, \Pi_i \rangle) = \\ &\quad StepsCorrect(\Gamma, \Delta \cup \{t : \tau\}, \Phi_C \cup \{\langle i, \phi_i \}, \Pi_i)\end{aligned}$$

Definition 6.20. The *Substep* function [extends definition 6.13]

$$Substeps(\mathbb{IA}\langle i, t : \tau, \phi_i, \Pi_i \rangle) = \{\langle j, \forall t : \tau.\ [\phi_i \to \phi_j] \rangle \mid \mathbb{S}\langle j, \phi_j, J_j \rangle \in \Pi_i\}$$

Definition 6.21. Correctness of a sequence of steps [extends definition 6.8]

$$\begin{aligned}&StepsCorrect(\Gamma, \Delta, \Phi_C, (\mathbb{IA}\langle i, t : \tau, \phi_i, \Pi_i \rangle, \sigma' \cdots)) = \\ &\quad Correct(\Gamma, \Delta, \Phi_C, \mathbb{IA}\langle i, t : \tau, \phi_i, \Pi_i \rangle) \\ &\quad \wedge StepsCorrect(\Gamma, \Delta, \Phi_C \cup Substeps(\mathbb{IA}\langle i, t : \tau, \phi_i, \Pi_i \rangle), (\sigma' \cdots))\end{aligned}$$

Like assumptions, we can 'extract' formulae proven in nested introduction steps, or to require that the introduction with assumption step to be refered when a step from its subproof is used as a justification by later steps. Such changes would require modifications to *Substeps*, which can be carried out in a manner similar to definition 6.14 and 6.15.

6.4.4 Induction Step

An induction step can be used if we want to prove some inductive property ϕ, by either structural induction over a universal quantification over a data structure (see definition 4.14), or induction over a function definition (see section 5.4.2). In an induction step, we generate ϕ', the *induction principle* of ϕ, which is a conjunction of *induction cases* $\phi_1 \wedge \cdots \wedge \phi_n$. We will then prove ϕ' by proving $\phi_1, \cdots, \phi_n$ individually, and then, as proven in chapter 8, ϕ would follow from ϕ'.

To begin, we generate the induction principle. For structural inductions, we can simply apply $InductionPrinciple$ to the formula in the step. However, for induction over functon definitions, the property we wish to apply $FunctionInductionPrinciple$ to can be buried deep inside some quantifications and implications. Therefore, we need to define a separate function $CompleteFunctionInductionPrinciple$ to generate the full induction principle:

Definition 6.22. Extraction of induction principle over function definition

$$CompleteFunctionInductionPrinciple(\Gamma, d, \phi) \triangleq \bigwedge_{x \in [1,n]} Pre(\phi_x)$$

where
$\langle Pre, \phi_{\mathfrak{I}} \rangle = Extract(\phi)$

- $\bigwedge_{x\in[1,n]} = FunctionInductionPrinciple(\Gamma, d, \phi_{\mathfrak{I}})$
- d is a function definition.

and

$$Extract(\forall t : \tau.\phi) \triangleq \langle \lambda\phi_a.(\forall t : \tau.(p\ \phi_a)), \phi_{\mathfrak{I}} \rangle \quad \text{where } \langle p, \phi_{\mathfrak{I}} \rangle = Extract(\phi)$$
$$Extract(\phi \to \phi') \triangleq \langle \lambda\phi_a.(\phi \to (p\ \phi_a)), \phi_{\mathfrak{I}} \rangle \quad \text{where } \langle p, \phi_{\mathfrak{I}} \rangle = Extract(\phi')$$
$$Extract(\mathfrak{I}(\phi)) \triangleq \langle \lambda\phi_a.\phi_a, \phi \rangle$$

We use the λ notation to denote an anoumynous function, where if the function f equals $\lambda x.y$, then $f(z) = [z/x]y$.

The $Extract$ function finds the induction marker in a sequence of universal quantifications and implications. It returns two values:

- A function Pre, that, when applied to a formula, puts the formula inside the sequence of quantifications and implications 'outside' the induction marker.
- The formula inside the induction marker.

For example, if $\langle Pre, \phi_{\mathfrak{I}} \rangle = Extract(\forall t_1 : \tau_1.[\phi_1 \to \forall t_2 : \tau_2.\mathfrak{I}(\phi_2)]$, then $\phi_{\mathfrak{I}} = \phi_2$, and $Pre(\phi')$ would be equal to $\forall t_1 : \tau_1.[\phi_1 \to \forall t_2 : \tau_2.\phi']$ for any ϕ'.

$Extract$ is only defined if ϕ_i, the goal, is of a certain form (see the remarks on definition 5.7).

Using $CompleteFunctionInductionPrinciple$, we may find the property we wish to perform induction on, and generate the property quantified induction principle. For example, say we have a function f defined as

```
f :: Int -> Int
f x | x < 2  =  0
    | x >= 2 =  1 + f (x - 2)
```

$$d = \langle \texttt{f-def}, f(x), (\langle x < 2, 0 \rangle, \langle x \geq 2, 1 + f(x-2) \rangle) \rangle$$

and some property

$$\phi = \forall x : \texttt{Int}.[x \geq 0 \to \mathfrak{I}(P(f(x)))]$$

Then we would have

$$CompleteFunctionInductionPrinciple(\Gamma, d, \phi) =$$

$$\forall x : \texttt{Int}.[x \geq 0 \to x > 2 \to P(0)]$$
$$\wedge$$
$$\forall x : \texttt{Int}.\big[x \geq 0 \to \forall r : \texttt{Int}.[\neg x < 2 \wedge x \geq 2 \;\to\; P(r) \wedge r = f(x-2) \;\to\; P(1+r)]\big]$$

We then need a way to check whether or not the cases we have written in an induction step actually 'cover' the cases in the generated induction principle. We define *Covered* as:

Definition 6.23. Coverage of induction cases

$$Covered(\Gamma, \phi_P, (\langle H_1, \phi'_1, \Pi_1\rangle, \cdots, \langle H_n, \phi'_n, \Pi_n\rangle)) \triangleq$$
$$\forall x \in [1,n].\exists j \in [1,n].\big[\Gamma, \Delta, ApplyHypothesis(H_j, \phi'_j) \vdash \phi''_x\big]$$

where $\bigwedge_{x\in[1,n]} \phi''_x = \phi_P$, and

$$ApplyHypothesis((), \phi) \triangleq \phi$$
$$ApplyHypothesis((\langle t : \tau, \phi_H\rangle, h\cdots), \phi) \triangleq \forall t : \tau.[\phi_H \to ApplyHypothesis(h\cdots, \phi)]$$

By definition 6.23, the induction principle is covered by the induction cases if for each case in ϕ_P, the generated induction principle, there is an induction case whose actual goal entails the case in the induction principle. By 'actual goal', we mean the theorem that is actually proven in the case, when the induction hypotheses are taken into consideration. For example, if have a case like below:

$$\langle(\langle x : \texttt{Int}, \top\rangle, \langle y : \texttt{Int}, x + y > z\rangle), P(x,y), \Pi_i\rangle$$

which means

> In this case, take an integer x, and another integer y, such that $x + y > z$, then prove $P(x,y)$ by steps Π_i.

in this case, what we have actually proven would be

$$\forall x : \texttt{Int}.\forall y : \texttt{Int}.[x + y > z \to P(x,y)]$$

for some z.

Once we know that the inductive principle is correctly reflected by the cases in the step, we then proceed to check the correctness of individual cases: first, we check whether or not the subproofs in the cases are correct, under the induction hypothesis, second, we check whether or not the end result obtained in the subproofs entails the goals of the cases.

We define the correctness of an induction case as:

Definition 6.24. Correctness of an induction case

$$CaseCorrect(\Gamma, \Delta, \langle P, \phi, \Pi\rangle) \triangleq$$
$$StepsCorrect(\Gamma \cup Introductions(P), \Phi_C \cup \{\langle \mathcal{H}, Hypothesis(P)\rangle\}, \Pi)$$
$$\wedge\, \Gamma \cup Introductions(P), \Delta, Last_\Phi(\Pi) \vdash \phi$$

where $\mathcal{H}$ is a special step number that is distinct from all other step numbers, and

$$Hypothesis((\langle t_1 : \tau_1, \phi_1\rangle, \cdots, \langle t_n : \tau_n, \phi_n\rangle)) \triangleq \bigwedge_{i\in[1,n]} \phi_i$$

$$Introductions((\langle t_1 : \tau_1, \phi_1\rangle, \cdots, \langle t_n : \tau_n, \phi_n\rangle)) \triangleq \bigcup_{i \in [1,n]} t_i : \tau_i$$

We use a special step number '$\mathcal{H}$' to serve as the 'number' of the induction hypothesis. Steps within the subproof of induction cases may refer to '$\mathcal{H}$' in their justifications.

Combining the above two definitions, we arrive at the definition for the correctness of induction steps:

Definition 6.25. Correctness of a step [extends definition 6.7]

$$\begin{aligned}
&Correct(\Gamma, \Delta, \Phi_C, \Im\langle i, \phi_i, C\rangle) \triangleq \\
&\quad Covered(\Gamma, InductionPrinciple(\Gamma, \phi_i), C) \;\wedge\; \forall j \in [1, |C|].CaseCorrect(\Gamma, \Delta, C_j) \\
&Correct(\Gamma, \Delta, \Phi_C, \Im\langle i, i_d, \phi_i, C\rangle) \triangleq \\
&\quad Covered(\Gamma, CompleteFunctionInductionPrinciple(\Gamma, d, \phi_i), C) \\
&\quad\; \wedge\; \forall j \in [1, |C|].CaseCorrect(\Gamma, \Delta, C_j)
\end{aligned}$$

where $d = \langle i_d, C_d\rangle \in \Delta$

After that, only the final result proven in an induction step is added to Φ_C and passed to following steps:

Definition 6.26. Correctness of a sequence of steps [extends definition 6.8]

$$\begin{aligned}
&StepsCorrect(\Gamma, \Delta, \Phi_C, (\Im\langle i, \phi_i, C\rangle, \sigma' \cdots)) = \\
&\quad Correct(\Gamma, \Delta, \Phi_C, \Im\langle i, \phi_i, C\rangle) \;\wedge\; StepsCorrect(\Gamma, \Delta, \Phi_C \cup \{\langle i, \phi_i\rangle\}, (\sigma' \cdots))
\end{aligned}$$

6.4.5 Equalities Step

In an equalities step, we prove the equality $t_0 = t_n$ by proving a 'chain' of equalities $t_0 = t_1, t_1 = t_2, \cdots, t_{n-1} = t_n$. For each equality in the 'chain', there is a justification for it. Thus, an equalities step would be correct, if all equalities in the chain are entailed by their respective justifications.

We define the correctness as:

Definition 6.27. Correctness of a step [extends definition 6.7]

$$\begin{aligned}
&Correct(\Gamma, \Delta, \Phi_C, \mathbb{E}\langle i, t_0, (\langle t_1, J_1\rangle, \cdots, \langle t_n, J_n\rangle)\rangle) \triangleq \\
&\quad \bigwedge_{i \in [1,n]} \Big[J_i = \texttt{Given} \;\vee\; J_i = \texttt{Tentative} \\
&\qquad \vee\; \big(\forall j \in J_i.\, j \in dom(\Phi_C) \;\wedge\; \Gamma, \Delta, \Phi_C(j_1), \cdots, \Phi_C(j_n) \vdash t_{i-1} = t_i\big) \Big]
\end{aligned}$$

We allow '`Given`' and '`Tentative`' as justifications for each term in the equality chain, and have included them in the above definition

Since our intention is to prove that the initial term, t_0, is equal to the last term in the chain, t_n, we add '$t_0 = t_n$' as the result obtained in this step to Φ_C.

Definition 6.28. Correctness of a sequence of steps [extends definition 6.8]

$$\begin{aligned} &\mathit{StepsCorrect}(\Gamma, \Delta, \Phi_C, (\mathbb{E}\langle i, t_0, (\langle t_1, J_1\rangle, \cdots, \langle t_n, J_n\rangle)\rangle, \sigma'\cdots)) = \\ &\quad \mathit{Correct}(\Gamma, \Delta, \Phi_C, \mathbb{E}\langle i, t_0, (\langle t_1, J_1\rangle, \cdots, \langle t_n, J_n\rangle)\rangle) \\ &\quad \wedge \mathit{StepsCorrect}(\Gamma, \Delta, \Phi_C \cup \{\langle i, t_0 = t_n\rangle\}, (\sigma'\cdots)) \end{aligned}$$

6.5 Checking Program

A program is a sequence of sections, which could contain declarations, lemmas, function definitions, or more importantly, proofs. When checking an entire program, we maintain a set of declarations already made, a set of lemmas already declared, and a set of function definitions stated. We add new declarations, lemmas, and function definitions to them as we move through the program. Once we encounter a proof section, we check the proof using everything we have already seen.

To put this formally, we define the procedure *Check* as below:

Definition 6.29. Program checking

$$\begin{aligned} \mathit{Check}(\Gamma, \Lambda, \Delta, ()) &\triangleq () \\ \mathit{Check}(\Gamma, \Lambda, \Delta, (\Gamma', s\cdots)) &\triangleq (\top, \cdots\mathit{Check}(\Gamma \cup \Gamma', \Lambda, \Delta, s\cdots)) \\ \mathit{Check}(\Gamma, \Lambda, \Delta, (\Lambda', s\cdots)) &\triangleq (\top, \cdots\mathit{Check}(\Gamma, \Lambda \cup \Lambda', \Delta, s\cdots)) \\ \mathit{Check}(\Gamma, \Lambda, \Delta, (\Delta', s\cdots)) &\triangleq (\top, \cdots\mathit{Check}(\Gamma, \Lambda, \Delta \cup \Delta', s\cdots)) \\ \mathit{Check}(\Gamma, \Lambda, \Delta, (\langle \phi_G, \Pi\rangle, s\cdots) &\triangleq (\mathit{StepsCorrect}(\Gamma, \Delta, \mathit{Initialise}\Phi_C(\Lambda, \Delta), \Pi) \\ &\qquad \wedge \phi_G =_{\mathrm{F}} \mathit{Last}\phi(\Pi), \\ &\qquad \cdots\mathit{Check}(\Gamma, \Lambda, \Delta, s\cdots)) \end{aligned}$$

where

Γ, Λ, Δ are the sets of the declarations, lemmas, and function definitions we have already seen so far.

- Γ', Λ', Δ' are sequences of declarations, lemmas, and function definitions repectively.
- $(x, \cdots S)$ is shorthand for $(x, S_1, \cdots, S_n)$, where S_1 to S_n are elements in sequence S.
- The function $\mathit{Last}\phi(\Pi)$ returns the formula in the last step in the step sequence Π.

and the $\mathit{Initialise}\Phi_C$ function is defined as

$$\mathit{Initialise}\Phi_C(\Lambda, \Delta) \triangleq \{\langle i, \phi\rangle \mid \langle i, \phi, J\rangle \in \mathit{Translate}(\Lambda \cup \Delta)\}$$

for any set of lemmas Λ and any set of function definitions Δ.

The *Check* procedure takes sets of existing declarations, lemmas, and function definitions, and return a sequence of either $\top$ of $\bot$. For each proof, we return $\top$ if the proof is correct, that is, the proof steps are correct, and that the goal is equal to the last proof step, and $\bot$ otherwise. For all other sections, we simply return $\top$.

If any section is not well-formed, for example, because it uses an undeclared type, then the implementation should not proceed any further, and report an error.

As discussed in section 4.1.2, and 4.2.2, we provide arithmetic functions ($+$, $-$, and $*$), as well as the integer predicates ($<$, $>$, $\leq$, and $\geq$). They do not need to be declared by the users, but we still need to put them in the context somewhere when checking proofs. The solution is to provide a *prelude*, denoted by Γ_p, that contains the declarations of the atom type, the integer type, arithmetic operations, and arithmetic comparators.

At last, we may now define how programs are checked:

Definition 6.30. Checking program

To check a program P, we call function *Check* as following:

$$Check(\Gamma_p, \varnothing, \varnothing, P)$$

where Γ_p is the *prelude*.

6.6 Checking Entailment

At the heart of our step checking algorithm lies the question of whether an entailment of the form

$$\Gamma, \Delta, \phi_{j_1}, \cdots, \phi_{j_n} \vdash \phi_i \tag{6.1}$$

holds or not, where Γ is the context, ϕ_i is the formula proven in step σ_i, and $\phi_{j_1}, \cdots, \phi_{j_n}$ are formulae proven in the steps referenced in step σ_i's justification.

6.6.1 Checking Entailment Syntactically

First, we may observe that if there is only one premise, and that premise is equivalent to the consequence, then the entailment holds trivially, as it is in essence a tautology.

Theorem 6.1. Tautological entailment

If both ϕ and ϕ' are well-formed under context Γ, then $\phi =_F \phi' \rightarrow (\Gamma, \Delta, \phi \vdash \phi')$

This is a 'shortcut' that allows us to quickly check the entailment in trivial cases, without envoking an external SMT solver. We use it for a number of reasons:

- Checking equivalence is faster than calling an SMT solver, as the later would require starting a subprocess which has high overhead.
- In induction steps, we check whether or not the goals of induction cases written by the users entails the generated case goals. If they are correct, the user-written case goals are likely to be equivalent to the generated ones, but they might use different names for introduced and quantified variables.

6.6.2 Checking Entailment Using an SMT Solver

When we consult the external SMT solver, we need to encode this entailment into a form that can be understood by the SMT solver. The SMT solvers we use implement the SMT-LIB standard, however, they still differ in terms of features and proving powers.

accept SMT-LIB scripts, which is a language specified by the SMT-LIB standard [8], as their input format. Here, we describe how entailments can be translated into SMT-LIB terms and signatures written using abstract SMT-LIB syntax.

We define the translation into concrete SMT-LIB scripts, which are inputs to SMT-LIB compliant SMT solvers, in chapter A.

Choice of SMT Solver

Emmy supports two SMT solvers: Z3 [65] and CVC4 [10]. They both implement SMT-LIB standard version 2.0 [8], but also support many features from later versions of the SMT-LIB standard, such as data structures [6].

Other SMT-LIB compliant solvers may be also usable, but we have not tested them.

In the beginning of the project, we chose Z3 because it is well-known, and has a detailed user guide [46]. We later added support for CVC4, because CVC4 supports for induction natively, which is of interest to us, while Z3 does not.

Notice that CVC4 and Z3 are two top-performing solvers in SMT-COMP, a competetion of SMT solvers [7] [9] [78].

We prefer to use Z3 as the underlying solver, because it has better support for data structure properties, as discussed in section 9.3.1. While CVC4 supports induction, we feel that this feature is not very important for us, because Emmy can generate induction principles.

SMT-LIB Logic

Before we perform the check, we must specify the SMT-LIB *logic* under which the check is performed, as the logic we use determines the set of *theories*, which, in SMT-LIB, is the class of universes and their interpretations with the same signature [8], with regard to which we are checking our formulae.

In general, we use the `AUFNIA` logic specified by the SMT-LIB standard. The `AUFNIA` logic uses:

- The Core theory, which specifies basic propositional algebra, logic connectives, and the equality relation between terms.
- Arrays (`A`), which are not a part of our logic, but required to use data structures.
- Free sort and function symbols (`UF`).
- Quantifiers (Not quantifier-free [`QF`]).
- Non-linear Integer Algebra (`NIA`), which also interprets the integer arithmetics we wish to perform.

For CVC4, we need to use the `AUFDTNIA` logic, which includes the theory of data structures (`DT`), as CVC4 requires that we state explicitly that we want data structures in our logic, while for Z3 we do not have to.

CVC4 Options

If we use CVC4 as the underlying solver, we need to ‘switch on’ induction with some command line arguments when calling CVC4. The exact arguments, used by the authors of the paper “Induction for SMT Solvers” [72], are [73]:

```
--quant-ind --quant-cf --conjecture-gen
--conjecture-gen-per-round=3 --full-saturate-quant
```

Unfolding of Induction

In definitions 4.11 and 5.16, we have defined induction markers, which marks the formula upon which induction should be performed. We now define the $Unfold$ function, which applies induction principles to the formulae in induction markers, 'unfolding' these formulae to their induction principles:

Definition 6.31. Unfolding induction principle

$Unfold(\Gamma,\Delta,\mathfrak{I}(\forall t:\delta.\phi)) \triangleq InductionPrinciple(\Gamma,\forall t:\delta.\phi)$	
$Unfold(\Gamma,\Delta,\mathfrak{I}^{i}(\phi)) \triangleq FunctionInductionPrinciple(\Gamma,d_i,\phi)$	Where $d_i = \langle i, C\rangle \in \Delta$
$Unfold(\Gamma,\Delta,P) \triangleq P$	P is a proposition
$Unfold(\Gamma,\Delta,\neg\phi) \triangleq \neg Unfold(\Gamma,\Delta,f,\phi)$	
$Unfold(\Gamma,\Delta,\phi_1\ C\ \phi_2) \triangleq Unfold(\Gamma,\Delta,\phi_1)\ C\ Unfold(\Gamma,\Delta,\phi_2)$	C is a binary connective
$Unfold(\Gamma,\Delta,Q\ t:\tau\ \phi) \triangleq Q\ t:\tau\ Unfold(\Gamma,\Delta,\phi)$	Q is either $\forall$ or $\exists$
$Unfold(\Gamma,\Delta,P(t_1,\cdots,t_n)) \triangleq P(t_1,\cdots,t_n)$	
$Unfold(\Gamma,\Delta,t_1=t_2) \triangleq t_1=t_2$	

We prove that $Unfold$ is sound, that is, if $Unfold(\Gamma,\Delta,\phi)$ is defined, then $Unfold(\Gamma,\Delta,\phi) \to \phi$, in section 8.3.

Because we only use induction to *prove* something, only ϕ_c, the consequence of entailment 6.1, which we wish to prove, needs to be unfolded. Entailment 6.1 now becomes:

$$\Gamma,\Delta,\phi_1,\cdots,\phi_n \vdash Unfold(\Gamma,\Delta,\phi_c) \tag{6.2}$$

For brevity, we use ϕ'_c to denote $Unfold(\Gamma,\Delta,\phi_c)$, the 'unfolded' consequence, from now on.

Entailment to Horn Clause

We then transform entailment 6.2 into

$$\Gamma,\Delta,\phi_1,\cdots,\phi_n,\neg\phi'_c \vdash \bot \tag{6.3}$$

that is, under the context Γ, it is impossible for ϕ'_c to be false when $\phi_1,\cdots,\phi_n$ are all true, or, the *Horn clause* [53] $\phi_1,\cdots,\phi_n,\neg\phi'_c$ is not satisfiable.

Since SMT solvers check satisfiability, we need to formulate our entailment as a Horn clause, and try to refute this clause.

Context to SMT-LIB Signature

Next, we need to translate the current context, Γ, which is a set of typing statements. In SMT-LIB, the corresponding properties are specified in *signatures*, denoted by Σ, which, informally, contains definitions of sorts (which corresponds to our types), function and predicate symbols, constructors, and variables.

We define $ContextInSignature$, a function that takes a context and returns the assertion that the context has been fully encoded in a certain signature Σ, as:

Definition 6.32. Translation of context into SMT-LIB signature

$$ContextInSignature(\Gamma) \triangleq \forall s \in \Gamma.[\![s]\!]_S$$

where Γ is a context, and

$$\begin{aligned}
[\![\tau :: \texttt{Type}]\!]_S &\triangleq \tau \in \Sigma^S \wedge ar(\tau) = 0 \\
[\![t : \tau]\!]_S &\triangleq t \in \Sigma^F \wedge \langle t, (\tau)\rangle \in R \\
[\![f : \tau_1 \cdots \tau_n \to \sigma]\!]_S &\triangleq f \in \Sigma^F \wedge \langle f, (\tau_1 \cdots \tau_n\ \sigma)\rangle \in R \\
[\![P : \tau_1 \cdots \tau_n \to \texttt{Prop}]\!]_S &\triangleq P \in \Sigma^F \wedge \langle P, (\tau_1 \cdots \tau_n\ \mathbf{Bool})\rangle \in R \\
[\![\delta :: C]\!]_S &\triangleq \delta \in \Sigma^S \wedge ar(\delta) = 0 \\
&\quad \wedge \forall a \in C.\big[a \in \Sigma^C \wedge a \in \mathrm{con}_\Sigma(\delta) \wedge \langle a, (\delta)\rangle \in R\big] \\
&\quad \wedge \forall c : \tau_1 \cdots \tau_n \in C.\big[c \in \Sigma^C \wedge c \in \mathrm{con}_\Sigma(\delta) \\
&\qquad\qquad \wedge \langle c, (\tau_1 \cdots \tau_n\ \delta)\rangle \in R\big] \\
&\quad \wedge \forall a \in C.\big[\mathcal{T}^a \in \Sigma^T \wedge \mathrm{tes}_\Sigma(a) = \mathcal{T}^a && (\sim) \\
&\qquad\qquad \wedge \langle \mathcal{T}^a, (\delta\ \mathbf{Bool})\rangle \in R\big] \\
&\quad \wedge \forall c : \tau_1 \cdots \tau_n \in C.\big[\mathcal{T}^c \in \Sigma^T \wedge \mathrm{tes}_\Sigma(c) = \mathcal{T}^c && (\sim) \\
&\qquad\qquad \wedge \langle \mathcal{T}^c, (\delta\ \mathbf{Bool})\rangle \in R\big] \\
&\quad \wedge \forall c : \tau_1 \cdots \tau_n \in C.\Big[\forall i \in [1, n].\big[\mathcal{S}_i^c \in \Sigma^G && (\sim) \\
&\qquad\qquad\qquad \wedge \mathrm{sel}_\Sigma(c)_i = \mathcal{S}_i^c \\
&\qquad\qquad\qquad \wedge \langle \mathcal{S}_i^c, (\delta\ \tau_i)\rangle \in R\big]\Big]
\end{aligned}$$

where

For any constructor c of data structure δ, $\mathcal{T}^c$ maps c to a fresh function symbol that serves as its *tester*.

- For any constructor function $c : \tau_1 \cdots \tau_n$ of data structure δ, $\mathcal{S}_i^c$ maps c to a fresh function symbol that serves as its ith *selector*, where $1 \le i \le n$.

To distinguish between SMT-LIB sorts and types in Emmy, we use the `monospace` letters to write types in Emmy, and **bold** letters to write SMT-LIB sorts.

$[\![\cdot]\!]_S$, the interpretation function for typing statements, takes a typing statement, and returns a property that the signature Σ must satisfy.

Informally, what we need to do is to

- turn each declared type and data structure into a *sort* in the signature, that does not take any other sorts as arguments.
- turn each declared constant of type τ into a nullary 'function', whose return type is τ, in the signature.
- add each function to the signature in the obvious way.
- turn each predicate that takes arguments of types $\tau_1, \cdots, \tau_n$ into a function that takes the corresponding sorts, and returns **Bool**, in the signature.

- turn each atomic constructor of data structure δ into a nullary constructor function, whose return type is δ, in the signature.
- for each constructor, create a fresh tester function and add it to the signature.
- for each argument required by each constructor function, create a fresh selector function, and add it to the signature.

Some of the properties of the resultant siganture, namely, those involving testers, which 'tests' whether or not a term is constructed by a certain constructor, and selectors, which 'selects' a argument that is passed to the constructor when constructing a term, are neither present in our program language, not relevant to the semantics of our programs, but required to be present in the SMT-LIB signature. In definition 6.32, we mark these properties in grey and denote them by (~) so that they can be distinguished from the rest.

For example, consider contxt Γ:

$$\begin{aligned}\Gamma = \{&\texttt{Int} :: \texttt{Type},\\ &\texttt{List} :: \{nil, cons : \texttt{Int List}\},\\ &length : \texttt{List} \rightarrow \texttt{Int},\\ &P : \texttt{List List} \rightarrow \texttt{Prop}\}\end{aligned}$$

The assertion $ContextInSignature(\Gamma)$ would be satisfied by the signature Σ where:

$$\begin{aligned}
\Sigma^S &= \{\mathbf{Int}, \mathbf{List}, \mathbf{Bool}\}\\
\Sigma^F &= \{nil, cons, length, P, nil^t, cons^t, cons^s_1, cons^s_2\}\\
\Sigma^C &= \{nil, cons\} \quad \Sigma^T = \{nil^t, cons^t\} \quad \Sigma^G = \{cons^s_1, cons^s_2\}\\
R &= \{\langle nil, (\mathbf{List})\rangle, \langle cons, (\mathbf{Int\ List\ List})\rangle,\\
&\quad \langle nil^t, (\mathbf{List\ Bool}), \langle cons^t, (\mathbf{List\ Bool}),\\
&\quad \langle cons^s_1, (\mathbf{List\ Int}), \langle cons^s_2, (\mathbf{List\ List})\\
&\quad \langle length, (\mathbf{List\ Int})\rangle, \langle P, (\mathbf{List\ List\ Bool})\rangle, \}\\
ar(\mathbf{List}) &= 0 \quad ar(\mathbf{Int}) = 0 \quad ar(\mathbf{Bool}) = 0\\
\mathrm{con}_\Sigma(\mathbf{List}) &= \{nil, cons\}\\
\mathrm{tes}_\Sigma(nil) &= nil^t \quad \mathrm{tes}_\Sigma(cons) = cons^t\\
\mathrm{sel}_\Sigma(nil) &= () \quad \mathrm{sel}_\Sigma(cons) = (cons^s_1, cons^s_2)
\end{aligned}$$

We are not showing the assertion $ContextInSignature(\Gamma)$ here, because it is going to be extremely long.

The translation from typing statements to concrete SMT-LIB scripts is defined in definition A.1.

Formulae to SMT-LIB Terms

We then translate the Horn clause $\phi_i, \cdots, \phi_n$, and $\neg\phi'_c$, into SMT-LIB terms.

We define $[\![\cdot]\!]_T$, the translation from our terms to SMT-LIB terms, and $[\![\cdot]\!]_F$, the translation from our formulae to SMT-LIB terms of type **Bool**, as below:

Definition 6.33. Translation of terms into SMT-LIB terms

$$
\begin{aligned}
[\![c]\!]_T &\triangleq c && c \text{ is a constant} \\
[\![v]\!]_T &\triangleq v && v \text{ is a variable} \\
[\![f(t_1,\cdots,t_n)]\!]_T &\triangleq f\ [\![t_1]\!]_T \cdots [\![t_n]\!]_T
\end{aligned}
$$

Definition 6.34. Translation of formulae into SMT-LIB **Bool** terms

$$
\begin{aligned}
[\![P]\!]_F &\triangleq P && P \text{ is a proposition} \\
[\![\neg\phi]\!]_F &\triangleq \neg\ [\![\phi]\!]_F \\
[\![\phi_1\ C\ \phi_2]\!]_F &\triangleq C\ [\![\phi_1]\!]_F\ [\![\phi_2]\!]_F && C \text{ is a binary connective} \\
[\![Q\ t:\tau\ \phi]\!]_F &\triangleq Q\ t:\tau\ [\![\phi]\!]_F && Q \text{ is either } \forall \text{ or } \exists \\
[\![P(t_1,\cdots,t_n)]\!]_F &\triangleq P\ [\![t_1]\!]_T \cdots [\![t_n]\!]_T \\
[\![t_1 = t_2]\!]_F &\triangleq \approx\ [\![t_1]\!]_T\ [\![t_2]\!]_T
\end{aligned}
$$

The translation of terms is essentially an identity. Terms are translated into the directly corresponding SMT-LIB term, with each function application $f(t_1,\cdots,t_n)$ replaced by a SMT-LIB style function application $f\ t_1 \cdots t_n$. The difference is only syntactic.

For formulae, we translate each of them into an SMT-LIB of sort **Bool**. Thus a proposition is a constant, or nullary function, of sort **Bool**, and a predicate is a function that maps from some sorts to the sort **Bool**, the connectives are functions from **Bool**(s) to a **Bool**. Equality, denoted by $\approx$ in SMT-LIB, is treated as a predicate as well. Quantifications are special *binders* in which variables are bound, whose syntax is largely the same as our quantifications.

For `if` expressions, we may first translate them into their equivalent formula, or translate them using the `ite` function implemented by many SMT solvers:

Definition 6.35. Translation of terms into SMT-LIB terms [Extends definition 6.33]

$$
[\![\mathtt{if}(\phi_c, t_t, t_f)]\!]_T \triangleq \mathtt{ite}\ [\![\phi_c]\!]_F\ [\![t_t]\!]_T\ [\![t_f]\!]_T
$$

The `ite` function's sementics is similar to `if`. Its first argument, the 'condition', should have sort **Bool**. Its value is equal to the second argument if the condition is true, and equal to the third if it is not.

As an example, consider the following formula:

$$
\phi = \forall x : \mathtt{Int}. P(x) \land P(f(x)) = x
$$

According to the above definitions, we have:

$$
[\![\phi]\!]_F = \forall x : \mathbf{Int}.\Big(\land\ (P\ x)\ \big(\approx\ (P\ (f\ x))\ x\big)\Big)
$$

The translation from terms and formulae into SMT-LIB terms is defined in definitions A.2 and A.3.

Declare the Propositions

Since propositions are constants, they must also be declared in the signature Σ. We define the function *AllPropositionsDefined?*, that takes a set of formulae and returns the assertion that all propositions are defined in some signature Σ, as:

Definition 6.36. Define proposition

$$AllPropositionsDefined(\Phi) \triangleq \bigwedge_{\phi \in \Phi} Defined?(\phi)$$

where

$$\begin{aligned}
Defined?(P) &\triangleq P \in \Sigma^F \wedge \langle P, (\mathbf{Bool}) \rangle \in R \\
Defined?(\neg\phi) &\triangleq Defined?(\phi) \\
Defined?(\phi_1 \; C \; \phi_2) &\triangleq Defined?(\phi_1) \wedge Defined?(\phi_2) && C \text{ is a binary connective} \\
Defined?(Q \; t : \tau \; \phi) &\triangleq Defined?(\phi) && Q \text{ is either } \forall \text{ or } \exists \\
Defined?(P(t_1, \cdots, t_n)) &\triangleq \top \\
Defined?(t_1 = t_n) &\triangleq \top
\end{aligned}$$

This definition is used when we defined the signature under which the check is performed.

Perform the Check

We will then generate the signature Σ under which the check is performed. The signature Σ should satisfy the following property:

$$ContextInSignature(\Gamma) \wedge AllPropositionsDefined(\{\phi_1, \cdots, \phi_n, \neg\phi'_c\})$$

That is, all types, data structures, functions, and predicates are properly encoded in Σ, and all propositions are defined in Σ.

Once we have generated the signature, we handle the premises and the consequence. The SMT solver we consult has an *assertion stack*, which can store assertions, which are formulae (SMT-LIB terms on type **Bool**) that we wish to be true, at different levels. The levels are useful to control the lifetime, or scope, of assertions, but since we run the SMT solver once for each entailment check, we do not care about the levels here.

For each premise ϕ_i, we assert that the formula $[\![\phi_i]\!]_F$ is true. For the (unfolded) consequence ϕ'_c, we assert that its negation, $[\![\neg\phi'_c]\!]_F$, is true.

Finally, we ask the SMT solver to check the satisfiability of the current set of assertions, by giving the following SMT-LIB command to the SMT solver:

```
(check-sat)
```

The SMT solver will check the satisfiability, and return one of the three results:

`(unsat)` means that is it impossible for the consequence to be false when all premises are true. Therefore, the entailment *holds*.

`(sat)` means that even if all premises are true, it is still possible for the consequence to be false. Therefore, the entailment *does not hold*.

`(unknown)` means that the solver cannot determine the satisfiability of the assertions, so we do not know whether the entailment holds or not.

To ensure the soundness of EMMY, that is, to make sure that EMMY cannot prove anything that is not true, we deem the entailment to not hold when the satisfiability is unknown.

Sometimes, the SMT solver may go on and on and never finish solving. This is often related to inductions: the SMT solver may enter an infinite loop when checking inductive properties [72]. To ensure that our program will not stop indefinitely, we impose a time limit on the satisfiability checking. In our implementation, the default time limit is 200ms, but it can be adjusted using a command line argument. If the SMT solver cannot finish solving before the time limit, it will return '`(unknown)`' as its answer.

Chapter 7

Implementation

In this chapter, we discuss how EMMY is implemented as a computer program, as well as the choice of programming languages and libraries.

EMMY has three main components:

A proof checker which is a command line application that takes EMMY programs as inputs, checks them, and displays the results.

A web interface where users can write proofs, and send the proofs to the proof checker to be checked.

A server which handles the communication between the proof checker and the web interface.

In addition, we provided a 'reader' for `#lang emmy`, a DSL in which EMMY programs can be written.

The relation between the three main components, and the external SMT solver, can be summarised by the following diagram:

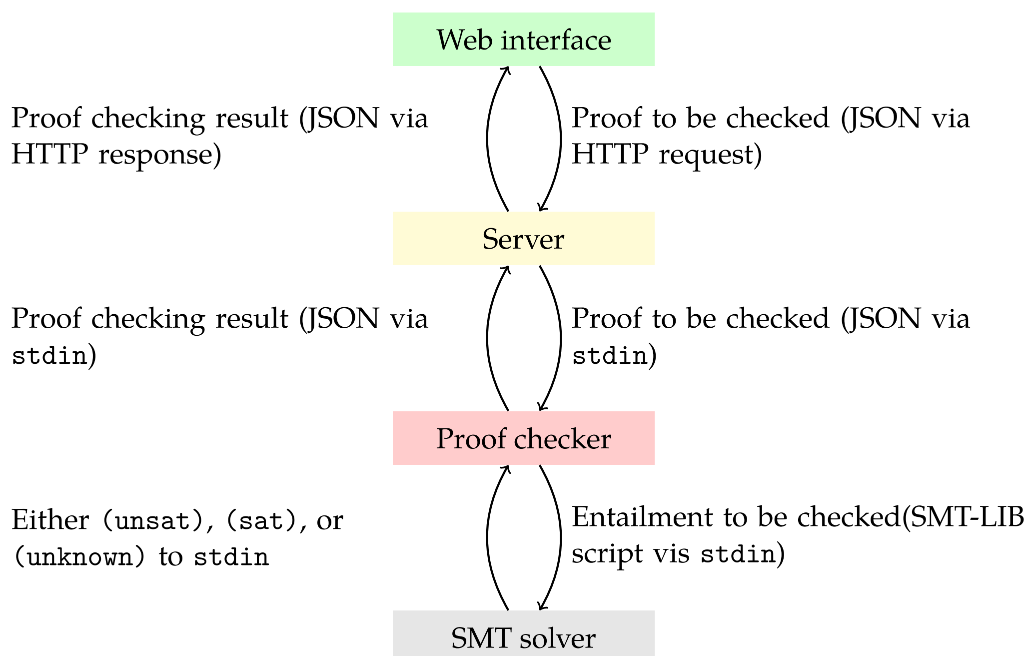


Figure 7.1: Structure of the project.

We shall now describe the implementation of each component in detail:

7.1 Proof Checker

The proof checker is a Racket program that takes Emmy programs as input via the standard input stream (`stdin`), checks the correctness of the program following the algorithm defined in chapter 6, and outputs the checking result as output via standard output (`stdout`). It will call the external SMT solver by spawning a subprocess, if needed.

We use the Racket language to write the proof checker for the following reasons:

- As a LISP (Scheme) dialect, Racket has excellent symbolic manipulation capabilities, making it easy to work with formulae expressed using s-expressions.
- The Racket parser can parse Emmy's s-expression based program language, which we will discuss in the next section, directly.
- While there is no well-maintained Racket binding for the SMT solvers we used, the fact that Racket can manipulate and write s-expressions means that we could very easily construct programs in SMT-LIB scrtips, and parse output from SMT solvers.
- Racket provides language building functionalities which allowed us to turn Emmy programs into program code that can be 'executed' directly, rather than data to be fed into another program.

By default, the proof checker accepts s-expression as the input format, which we will explain in detail in the next section. A command line argument can be used to instruct the proof checker to use JSON as the input format.

The core proof checking algorithm, described in chapter 6, is defined in a functional style. Our implementation is largely a translation of the definition into functional Racket code. The code is mostly pure, apart from the generation of fresh variable names which is stateful. This makes it easy for us to reason about the code, to debug and to structure the program in general.

7.1.1 Logic and Program Representation

In the proof checker, terms, formulae, proofs, and programs are represented using LISP's s-expressions [62]: a single item, which could be a term, a formula, a step, or a lemma section, is either a single *symbol*, such as '`x`', '`SomeProposition`', or a *list* of items, enclosed by brackets, like '`(f 10 x y)`'.

Like LISPs, we use Polish [12], or prefix notation, putting functions, operators, predicates, connectives, *et cetra*, before their arguments, so that $x+y$ would become as '`(+ x y)`', $f(x,1)$ would become '`(f x 1)`', and $A \to B$ would become as '`(Implies A B)`'.

The first element in a list can also serve as the 'tag' that indicates the type of that item. For example, we use '`[Function f Int -> Int]`' to represent a function declaration that says f : `Int` → `Int`, where '`Function`' tells us about the type of this declaration.

Notice that in the above example we used square brackets instead of normal brackets. In Emmy and Racket's concrete syntax, normal, square, and curly brackets are semantically equivalent.

We chose s-expression because the Racket parser can parse directly our programs written in s-expression, saving us from writing our own parser. The representation of programs, in

Racket code, is also identical to their representation in s-expression, reducing the overhead in maintaining multiple representations of programs.

The full, concrete syntax for the program language can be found in appendix D.

Example Program

The following program, adapted from `theorem-15.prf`, first declares the `List` type, two list functions, state a lemma related to the properties of the functions, then defines the two functions, and prove that $reverse(reverse(l)) = l$ (theorem B.15):

```
1  {Declare
2   [Data List [nil] [cons Int List]]
3   [Function append List List -> List]
4   [Function rev List -> List]
5   }
6
7  {Lemma
8   [LD (Forall List l1 (Forall List l2 (= (rev (append l1 l2))
9                                          (append (rev l2) (rev l1)))))]
10  }
11
12 {Define
13  [append-1 (append nil l) l]
14  [append-2 (append (cons i l1) l2)
15            (cons i (append l1 l2))]
16
17  [rev-1 (rev nil) nil]
18  [rev-2 (rev (cons i l)) (append (rev l) (cons i nil))]
19  }
20
21 {Proof (Forall List l (= l (rev (rev l))))
22   [G Induction (Forall List l (= l (rev (rev l))))
23      [Show (= nil (rev (rev nil)))
24       By (rev-1)]
25      [Take List l such that (= l (rev (rev l)))
26       Take Int x
27
28       Show (= (cons x l) (rev (rev (cons x l))))
29       By (append-1 append-2 rev-1 rev-2 LD %Hypo)]
30      ]
31   }
```

We provide more example programs in appendix E, all written in the concrete syntax.

JSON Representation

We also define the JSON [80] representation of programs, and have implemented the translation between the JSON representation and the canonical, s-expression-based representation. It is intended to be used when the proof checker is communicating with outside programs,

such as our web interface, and potentially, future native GUI proof assistants, or text editor plugins.

The JSON representation is more verbose, but it can be easily parsed by other programming languages, using existing libraries. The JavaScript language, in which EMMY's web interface is written, provides native functions for parsing JSON representations.

7.1.2 Proof Checking Workflow

The entry point of the proof checker is the `main` module in `main.rkt`. The `main` module calls the `check` procedure, which reads and parses the program from the standard input, and passes the input program, as a Racket list of sections, to the `check-loop` procedure, defined in `lang/check-loop.rkt`.

The `check-loop` procedure corresponds to the *Check* procedure, defined in definition 6.29. It iterates through the sections. For each section, `check-loop` first checks for errors using the procedures defined in `checker/error-checker.rkt`. Then, if it is a declaration, function definition, or lemma section, `check-loop` will add the content of the section to the collection of declarations, definitions, or lemmas seen so far. If it is a proof section, it calls the `checker/checker.rkt` to check the proof.

The `check-proof` procedure first calls `transform-definition`, which corresponds to *Translate* defined in definition 6.6, to translate function definitions into `Given` steps. The translated function definitions and lemmas are then added to Φ_C. `check-proof` then calls `check-steps`, which corresponds to *StepsCorrect* and *Correct*, to check the sequence of steps in the proof.

The `check-steps` procedure recursively checks the correctness of each step in a sequence (represented in Racket as a list) of steps, following the algorithm defined in *StepsCorrect* and *Correct*. ϕ_C, the context and the definitions are carried through recursive calls as function arguments. It calls itself in cases where a subproof needs to be checked, such as for assumptions and induction cases.

We check entailment by calling the `entails?` procedure, defined in `checker/checker.rkt`. It first checks syntactically as defined in definition 6.1. Then, if the check fails, it calls the `check-entailment` procedure, defined in `checker/smt.rkt`, to check the entailment using an SMT solver.

The `check-entailment` procedure calls various translation procedures to translate the formulae and context into SMT-LIB script representations. It then generates the complete SMT-LIB program and calls `call-solver` to spawn the solver process and perform the check.

After the checking results are ready, `check-loop` prints the result, either as s-expression or as JSON, depending on settings, via standard output.

7.2 Web Interface

We provide a web interface for writing and checking EMMY programs, that supports

- Writing, editing, loading, and saving EMMY program.
- Checking the proof and displaying the result, including displaying the correctness of induction principles.
- Annotating the program with comment sections.
- Automatically renumbering proof steps.

- Automatically generating induction principles for induction over data structures.

On the web interface, formulae are written using a syntax similar to the syntax used in the logics course (see section 2.1.1), except for function applications, which are written in the Haskell style, that is, `f x y` instead of $f(x,y)$.

For example, the formula

$$\forall x : \texttt{Int}.\forall y : \texttt{Int}.[P(x,y) \quad \rightarrow \quad f(x) = g(x,y) \wedge f(x) - 1 > y]$$

can be written on the web interface as

```
Forall x: Int. Forall y: Int. [
  P(x, y) -> f x = g x y And f x - 1 > y
]
```

After a formula is written down correctly, the web interface would render it in LaTeX style, to make the formula more concise and easier to read:

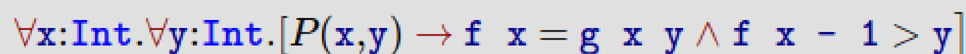


Figure 7.2: The above formula rendered on the web interface of Emmy.

We believe that by using a syntax similar to the one the students are already familiar with, we lowered the barrier of entry of Emmy, making it easier for new users to learn Emmy and become productive more quickly.

7.2.1 Technical Details

The web interface is an one-page web application written in Vanilla Javascript in a functional style: we made extensive use of immutable data structure operations, higher-order functions, and side-effect free code.

We used React [71], a popular JavaScript library for building interactive, stateful web applications. We use React to manage the state of the web interface, to dynamically render UI components, and to handle input events. We also used a number of other packages that extends React to provide additional features, such as rearranging program sections by dragging and dropping them, and LaTeX style formula rendering.

When the user clicks on the 'CHECK' button on the web interface, the application makes an request to a server, which we will describe in the next chapter, and display the proof checking result returned from the server.

7.2.2 Screenshots of the Web Interface

Here is a collection of the screenshots of Emmy's web interface.

Figure 7.3: A proof of theorem B.3 written using the web interface of Emmy, with a comment section.

Emmy Open Save HIDE ALL SHOW ALL CHECK

Declare HIDE X

x P: Atom -> Prop
x f: Atom -> Atom
x g: Atom -> Atom
x h: Atom -> Atom
ADD
The integer type (Int) and atom type (Atom) are already defined for you.

Proof HIDE X

Checked: Correct
Goal: $\forall$v:Atom.g v = h v

Ok 1 $\forall$u:Atom.g (f u) = h (f u) Given X + A+ I+ IA+ Ind+ E+
Ok 2 $\forall$z:Atom.$\exists$v:Atom.f v = z Given X + A+ I+ IA+ Ind+ E+

3 Take a: Atom X +I A+I I+I IA+I Ind+I E+I
Ok 4 $\exists$v:Atom.f v = a 2 X + A+ I+ IA+ Ind+ E+

5 Take b: Atom such that f b = a X +I A+I I+I IA+I Ind+I E+I
Ok 6 g (f b) = h (f b) 1 X + A+ I+ IA+ Ind+ E+
Ok 7 g a = h a 5 6 X + A+ I+ IA+ Ind+ E+
\+ A+ I+ IA+ Ind+ E+

Ok 8 g a = h a 4 7 X + A+ I+ IA+ Ind+ E+
\+ A+ I+ IA+ Ind+ E+

Ok 9 $\forall$v:Atom.g v = h v 8 X + A+ I+ IA+ Ind+ E+
Renumber

ADD COMMENT ADD PROOF ADD DECLARE ADD LEMMAS ADD DEFINE

Figure 7.4: A proof of theorem B.5 written using the web interface.

Emmy Open Save HIDE ALL SHOW ALL CHECK

Declare HIDE X

```
Data Code = lf Int | nd
Data Tree = node Tree Tree | leaf Int
Data ListC = nilC | consC Code ListC
Data ListT = nilT | consT Tree ListT
appendC: ListC ListC -> ListC
appendT: ListT ListT -> ListT
encd: Tree -> ListC
decd: ListC -> Tree
decdAux: ListC ListT -> Tree
```

ADD
The integer type (Int) and atom type (Atom) are already defined for you.

Lemmas HIDE X

```
DC    ∀xs:ListC.∀ys:ListC.∀zs:ListC.appendC xs (appendC ys zs) = appendC (appendC xs ys) zs
```

ADD

Define HIDE X

```
appendC-b     appendC nilC l = l
appendC-i     appendC (consC x xs) l = consC x (appendC xs l)
appendT-b     appendT nilT l = l
appendT-i     appendT (consT x xs) l = consT x (appendT xs l)
encd-l        encd (leaf i) = consC (lf i) nilC
encd-n        encd (node t1 t2) = appendC (encd t1) (appendC (encd t2) (consC nd nilC))
decd-d        decd cds = decdAux cds nilT
decdAux-b     decdAux nilC (consT t ts) = t
decdAux-l     decdAux (consC (lf i) cds) ts = decdAux cds (consT (leaf i) ts)
decdAux-n     decdAux (consC nd cds) (consT t1 (consT t2 ts)) = decdAux cds (consT (node t2 t1) ts)
```

ADD

Proof HIDE X

```
Checked: Correct
Goal: ∀t:Tree.decd (encd t) = t
  L   Prove ∀t:Tree.∀cds:ListC.∀ts:ListT.decdAux (appendC (encd t) cds) ts = decdAux cds (consT t ts)
      by induction on the definition of data structure Tree                         ->Function X
Induction principle is correct
  Base case
    Take x: Int
    Take cds: ListC
    Take ts: ListT
    P+I P+IA
              decdAux (appendC (encd (leaf x)) cds) ts                              X ->Subproof
    To show     =
              decdAux cds (consT (leaf x) ts)
    By appendC-b appendC-i appendT-b encd-l decdAux-l
  Inductive Step
                              ∀cds:ListC.∀ts:ListT.
                                decdAux (appendC (encd t1) cds) ts
    Take t1: Tree such that       =
                                decdAux cds (consT t1 ts)
                              ∀cds:ListC.∀ts:ListT.
                                decdAux (appendC (encd t2) cds) ts
    Take t2: Tree such that       =
                                decdAux cds (consT t2 ts)
    Take cds: ListC
    Take ts: ListT
    P+I P+IA
              decdAux (appendC (encd (node t1 t2)) cds) ts        X ->Trivial + A+ I+ IA+ Ind+ E+
    To show     =
              decdAux cds (consT (node t1 t2) ts)
    Ok 1 decdAux (appendC (encd (node t1 t2)) cds) ts
        = decdAux (appendC (appendC (encd t1) (appendC (encd t2) (consC nd nilC))) cds) ts   encd-n
        = decdAux (appendC (encd t1) (appendC (appendC (encd t2) (consC nd nilC)) cds)) ts   DC
        = decdAux (appendC (appendC (encd t2) (consC nd nilC)) cds) (consT t1 ts)            %Hypo
        = decdAux (appendC (encd t2) (appendC (consC nd nilC) cds)) (consT t1 ts)            DC
        = decdAux (appendC (consC nd nilC) cds) (consT t2 (consT t1 ts))                     %Hypo
        = decdAux (consC nd cds) (consT t2 (consT t1 ts))                                    appendC-b appendC-i
        = decdAux cds (consT (node t1 t2) ts)                                                decdAux-n
        +
  Add Case
  + A+ I+ IA+ Ind+ E+
Ok  G   Ind:[∀t:Tree.decd (encd t) = t]   appendC-b appendC-i appendT-b appendT-i encd-l encd-n decd-d
                                          decdAux-b decdAux-l decdAux-n L
Renumber
```

ADD COMMENT | ADD PROOF | ADD DECLARE | ADD LEMMAS | ADD DEFINE

Figure 7.5: A proof of theorem B.20 written using the web interface, using a large number of declarations and definitions, and an inductive step with an equalities step.

7.3 Server

The server handles the communication between the web interface and the proof checker. When the server receives a request from the web interface, it spawns a proof checker process, and passes the program sent by the web interface to the proof checker via the standard input stream (`stdin`). Once the proof checker finishes, the server passes the results from the proof checker back to the web interface.

The server is a very simple Node.js program written using the Express [39] library.

7.4 `#lang emmy`

We also provide the Emmy language, a DSL implemented as a Racket `#lang` language [25]. The users may write Emmy programs using the Emmy language, save the program in a file, and check the proof by 'runing' the file directly using the Racket interpreter.

The syntax of the Emmy language is the same as the s-expression based syntax described in section 7.1.1. However, in the beginning of a Emmy file, there must be the following line:

```
1 #lang emmy
```

To check the code, simply run the Racket interpreter on the code file:

```
racket name-of-file
```

When the Racket interpreter sees `#lang emmy` when 'running' an Emmy program, it knows that the code is written in the Emmy language, so it would use an 'reader' provided by us to read the code below. Our reader will then read the program, and pass the program to the `check-loop` procedure to check the correctness of the program.

Part III

Evaluation and Conclusion

Chapter 8

Soundness of Induction

When induction is performed in a proof, either by the use of an induction marker or in an induction step, Emmy generates an induction principle for the formula we wish to proof, and then proves the generated induction principle instead of the original formula in the normal way.

In this chapter, we prove that induction in Emmy is sound, that is, if formula ϕ' is the induction principle for some formula ϕ, then if ϕ' is true, ϕ must be true.

Formally, we define soundness of induction as:

Theorem 8.1. Induction principle is sound

For any formula ϕ, context Γ, function definition d, if there exists some ϕ' such that $\phi' = InductionPrinciple(\Gamma, \phi)$, then $\phi' \rightarrow \phi$, and if there exists some ϕ'' such that $\phi'' = FunctionInductionPrinciple(\Gamma, d, \phi)$, then $\phi'' \rightarrow \phi$.

We first prove the soundness of the induction principles generated by functions $InductionPrinciple$ and $FunctionInductionPrinciple$. These two proofs are based on the principle of well-founded orderings: we first identify the order, and then show that the set of terms for which the induction principle is unsound is empty, by the properties of well-founded orders. After that, we prove $Unfold$, which uses the two functions, is sound.

Our conclusion from the proofs is that:

- Emmy generates correct induction principle for inductions over data structures.
- Emmy generates correct induction principle for inductions over function definitions, *if* the function definition *terminates*.

We will then discuss how non-terminating functions affect the soundness, and how it effects the use of Emmy.

8.1 Induction Over Data Structures

Proof. Take some context Γ, variable t, and type δ such that $\Gamma \vdash \delta :: C$ for some set of constructors C. Take formula ϕ arbitrary. Take formula $\phi' = InductionPrinciple(\Gamma, \forall t : \delta.\phi)$, and $\phi'_1 \wedge \cdots \wedge \phi'_n = \phi'$. That is, ϕ' is the induction principle for ϕ, and ϕ'_i is a case in the induction principle for $i \in [1, n]$.

Assume that ϕ', the induction principle, holds.

We now assume that the induction principle is *not sound*. That is, there exists some term t_F of type δ, such that $[t_F/t]\phi$ does not hold under the assumption that ϕ' holds. Let us

denote the set of such terms as T'_δ, which is a subset of T_δ, the set of terms of type δ. Since all terms of type δ are in a well-founded order, as required by our data structure axioms, T'_δ has at least one minimal elements.

Take some term t' with type δ such that t' is the *minimal* term for which $[t'/t]\phi$ *does not* hold. That is, there does not exist any term t'' with type δ such that $t'' < t'$, by definition 4.13, and that $[t''/t]\phi$ is false.

We show that such a term t' cannot exist by contradiction:

Since t' is of type δ, then, according to the data structure axioms, t' is constructed by a constructor. t' must be either

1. An atomic constructor a.

   If $t' = a$, then $a \in C$. By the definition of $InductivePrinciple$, ϕ'_i, one of the cases that forms the induction principle, must be $[a/t]\phi$. If ϕ' is true, then $[a/t]\phi$ must be true, so is $[t'/t]\phi$. $[t'/t]\neg(\phi' \to \phi)$ will not hold. There is a contradiction.

2. A function constructor c applied to a sequence of terms $t'_1, \cdots, t'_m$, where $\Gamma \vdash c : \tau_1 \cdots \tau_m \to \delta$, $\Gamma \vdash t'_i : \tau_i$ for all $i \in [1, m]$.

   We first define I, the set of indices of recursive calls, as, $I = \{i \in [1, m] \mid \tau_i = \delta\}$.

   If $t' = c(t'_1, \cdots, t'_m)$, then $c : \tau_1 \cdots \tau_m \in C$. By the definition of $InductivePrinciple$, ϕ'_i, one of the cases that forms the induction principle, must be $\forall t_1 : \tau_1. \cdots t_m : \tau_m. \bigwedge_{i \in [1,n]} (\tau_1 = \delta \to [t_i/t]\phi) \to [c(t_1, \cdots, t_m)/t]\phi$, which, by the definition of I, is equivalent to $\forall t_1 : \tau_1. \cdots t_m : \tau_m. \bigwedge_{i \in I} [t_i/t]\phi \to [c(t_1, \cdots, t_m)/t]\phi$. If we eliminate the universal quantification, we may obtain $\bigwedge_{i \in I} [t'_i/t]\phi \to [c(t'_1, \cdots, t'_m)/t]\phi$.

   Now, there are two possibilities:

   (a) $\bigwedge_{i \in I} [t'_i/t]\phi$ is true, and $[c(t'_1, \cdots, t'_m)/t]\phi$ is true. However, this would contradict our assumption that $[t'/t]\phi$ is false.

   (b) $\bigwedge_{i \in I} [t'_i/t]\phi$ is false. This would mean that there exists some $i \in I$, such that $[t'_i/t]\phi$ is false. However, since $t' = c(t'_1, \cdots, t'_m)$, by definition 4.13, $t'_i < t'$. This contradicts with out assumption that t' is a minimal element of the set of terms for which ϕ does not hold.

Since we reach a contradiction in all cases, the minimal element t' of T'_δ does not exist. However, this would contradict the property of well-founded order that any non-empty subset of the set with a well-founded order has a minimal element. This means that T'_δ is empty. Therefore, if ϕ', the induction principle, is true, then ϕ, the original formula, is true, for all possible values of t. Hence the induction principle is sound.

□

8.2 Induction Over Function Definitions

As discussed in section 5.4.2, if we want to prove something that looks like:

$$\forall t_1 : \tau_1. \cdots \forall t_n : \tau_n. R(t_1, \cdots, t_n, f(t_1, \cdots, t_n)) \tag{8.1}$$

Then we should first prove that:

$$\forall t_1 : \tau_1. \cdots \forall t_n : \tau_n. \forall r : \sigma. [f(t_1, \cdots, t_n) = r \to R(t_1, \cdots, t_n, r)] \tag{8.2}$$

Then, if f terminates, then formula 8.1 follows from 8.2.

The formula $R(t_1,\cdots,t_n,f(t_1,\cdots,t_n))$ corresponds to our ϕ, which should be some formula that contains $f(t_1,\cdots,t_n)$ as a subterm, where f's definition is what we perform induction on, and $R(t_1,\cdots,t_n,r)$ would be $[r/f(t_1,\cdots,t_n)]\phi$.

We will split the proof into two parts:

1. We prove that for some ϕ, d, and Γ,

$$FunctionInductionPrinciple(\phi,d,\Gamma) \rightarrow \forall r:\sigma.[r = f(t_1,\cdots,t_n) \rightarrow [r/f(t_1,\cdots,t_n)]\phi]$$

   where $\{f(t_1,\cdots,t_n)\} = Calls_F(f,\phi)$ ($Call_F$ is defined in definition 5.14).

2. We prove that if f always terminates, then

$$\forall r:\sigma.[r = f(t_1,\cdots,t_n) \rightarrow [r/f(t_1,\cdots,t_n)]\phi] \quad \rightarrow \quad \phi$$

The first part proves that *FunctionInductionPrinciple* correctly generates the induction principle. The second part proves that if the function terminates, then the correct induction principle implies the original formula.

8.2.1 Correctness of *FunctionInductionPrinciple*

Proof. Take some context Γ, and function definition $d = \langle i, f(t_{a1},\cdots,t_{an}), C\rangle$, such that $\Gamma \vdash f : \tau_1\cdots\tau_n \rightarrow \sigma$. Let $t_1,\cdots,t_n$ be terms with types $\tau_1,\cdots\tau_n$. Let ϕ be a formula such that $\{f(t_1,\cdots,t_n)\} = Calls_F(f,\phi)$, that is, $f(t_1,\cdots,t_n)$ is the only unique call to f in formula ϕ.

We will use c_f to denote $f(t_1,\cdots,t_n)$ from now on.

We first assume that the function definition upon which we perform induction is defined for all arguments, or that it always terminates. That is,

$$\forall t'_1:\tau_1.\cdots\forall t'_n:\tau_n.\exists t':\sigma.t' = f(t'_1,\cdots,t'_n)$$

We then define the following relation:

> **Definition 8.1.** Recursive call order For terms $t'_1,\cdots,t'_n$ of types $\tau_1,\cdots,\tau_n$, if, by definition d, $f(t'_1,\cdots,t'_n) = t_b$, and that $f(t''_1,\cdots,t''_n) \in Calls_T(f,t_b)$, for terms $t''_1,\cdots,t''_n$ of types $\tau_1,\cdots,\tau_n$, that is, t_b contains $f(t''_1,\cdots,t''_n)$ as a subterm, then $\langle t''_1,\cdots,t''_n\rangle <_R \langle t'_1,\cdots,t'_n\rangle$

Intuitively, if we make the recursive call $f(t''_1,\cdots,t''_n)$ when calling $f(t'_1,\cdots,t'_n)$, then $\langle t'_1,\cdots,t'_n\rangle <_R \langle t''_1,\cdots,t''_n\rangle$ It is easy to see that $<_R$ is a well-founded partial order. Because f always terminates, it is impossible to have an infinite chain of recursive calls.

Let A be the set of all possible cartesian products of terms of types $\tau_1,\cdots,\tau_n$. We then define $A' \subseteq A$ as

$$A' = \{\langle t'_1,\cdots,t'_n\rangle \in A \mid \neg([t'_1/t_1],\cdots,[t'_n/t_n])\ \phi\}$$

That is, A' is the set of groups of terms $\langle t'_1,\cdots t'_n\rangle$, for which $([t'_1/t_1],\cdots,[t'_n/t_n])\ \phi$ does not hold. Since all elements in A are in the well-founded order $<_R$, A' must be either empty, or have at least one minimal elements.

Take formula $\phi' = FunctionInductionPrinciple(\Gamma,d,\phi)$, and $\phi'_1 \wedge\cdots\wedge \phi'_m = \phi'$. That is, ϕ' is the induction principle for ϕ, and ϕ'_i is a case in the induction principle for $i \in [1,m]$.

We assume that ϕ', the induction principle, holds.

Now, we wish to prove that

$$\forall r : \sigma.\big[r = c_f \rightarrow [r/c_f]\phi\big]$$

Take some r_0 and assume that $r_0 = c_f$. We now need to prove $[r_0/c_f]\phi$, which, according to the assumption, is equivalent to ϕ: that is, we now need to prove ϕ.

We then assume that $\langle t_1, \cdots, t_n\rangle$ is a *minimal* element in A'. This would imply that ϕ does not hold under the assumption that ϕ' holds. We shall then show by contradiction that $\langle t_1, \cdots, t_n\rangle$ cannot exist as a minimal element. Which implies that A' is empty, hence the ϕ would always be true.

According to lemma 6.1, we have, under the definition d, the following property:

$$c_f = f(t_1, \cdots, t_n) = S\ b_i \quad \wedge \quad S\ \phi_i \wedge \bigwedge_{j \in [1, i-1]} (\neg S\ \phi_j) \tag{8.3}$$

where $\langle \phi_i, b_i\rangle$ is the ith case in definition d, and sequence of substitution $S = ([t_1/t_{a1}], \cdots, [t_n/t_{an}])$. [1] Since the induction principle ϕ' is defined, the case body b_i must either

- contain no recursive call ($|Calls_T(f, b_i)| = 0$).

  If this is the case, then one of the cases in the induction principle must be

  $$S\ \phi_i \wedge \bigwedge_{j \in [1, i-1]} (\neg S\ \phi_j) \ \rightarrow \ [S\ b_i/c_f]\phi$$

  From the assumption that the induction principle is true, and property 8.3, we know that $[S\ b_i/c_f]\phi$ is true. However, since $c_f = S\ b_i$, $[S\ b_i/c_f]\phi$ would be the same as ϕ, which implies that ϕ is true, which contradicts the assumption that ϕ does not hold.

- contain one unique recursive call ($|Calls_T(f, b_i)| = 1$).

  If this is the case, then one of the cases in the induction principle must be

  $$\forall r : \sigma.\big[S\ \phi_i \wedge \bigwedge_{j \in [1, i-1]} (\neg S\ \phi_j) \ \rightarrow \ (S_b\ [r/c_f]\ \phi \wedge r = S_b\ c_f \rightarrow [S\ [r/t_r]\ b_i/c_f]\ \phi)\big]$$

  where $S_b = ([S\ t'_1/S\ t_{a1}], \cdots, [S\ t'_n/S\ t_{an}])$, and $\{t_r\} = \{f(t'_1, \cdots, t'_n)\} = Calls_T(f, b_i)$[2].

  Take a term r', such that $r' = S_b\ c_f$, which means r' is the result returned by the recursive call.

  By property 8.3, we now have:

  $$S_b\ [r/c_f]\ \phi \wedge r = S_b\ c_f \rightarrow [S\ [r/t_r]\ b_i/c_f]\ \phi$$

  Now, there are two possibilities:

  1. $S_b\ [r'/c_f]\ \phi$, the induction hypothesis, is true.
     From the induction principle, we then have $[S\ [r'/t_r]\ b_i/c_f]\ \phi$.
     By assumption, we have $r' = S_b\ c_f$. By definition of S_b and S, we then have $r' = S_b\ f(t_1, \cdots, t_n) = f(S\ t'_1, \cdots, S\ t'_n)$, which gives $r' = S\ f(t'_1, \cdots, t'_n) = S\ t_r$.
     This gives $[S\ [S\ t_r/t_r]\ b_i/c_f]\ \phi$, which is equivalent to $[S\ b_i/c_f]\ \phi$. By property 8.3, this gives $[c_f/c_f]\ \phi$, hence ϕ is true.
     This contradicts with our assumption that ϕ is not true.

[1] Notice that by this definition, $t_i \neq c_f$ for all $i \in [1, n]$. Also, since t_{a1} is required by definition 5.11 to be distinct from other variables, $S\ S\ \phi^*$ would be equal to $S\ \phi^*$ for all formula or term ϕ^*.

[2] By the definition of S, this means S_b is equivalent to $([S\ t'_1/t_1], \cdots, [S\ t'_n/t_n])$.

2. $S_b\ [r'/c_f]\ \phi$, the induction hypothesis, is false.
   Since $r' = S_b\ c_f$, we have $\neg S_b\ [S_b\ c_f/c_f]\ \phi$, which would be equivalent to $\neg S_b\ \phi$.
   By the definitions of S and S_b,we also have $\neg([S\ t'_1/t_1], \cdots, [S\ t'_n/t_n])\ \phi$, which gives $\langle S\ t'_1, \cdots, S\ t'_n \rangle \in A'$.
   Since $f(t_1, \cdots, t_n) \in Calls_F(f, \phi)$, we have $f(S\ t'_1, \cdots, S\ t'_n) \in Calls_F(f([S\ t'_1/t_1], \cdots, [S\ t'_n/t_n])\ \phi)$. which means that $\langle S\ t'_1, \cdots, S\ t'_n \rangle <_R \langle t_1, \cdots, t_n \rangle$.
   We have now found a member of A' that is less than $\langle t_1, \cdots, t_n \rangle$, which contradicts the assumption that $\langle t_1, \cdots, t_n \rangle$ is a minimal element.

□

8.2.2 The Original Formula is True if Function Terminates

Proof. Take some context Γ, and function f, such that $\Gamma \vdash f : \tau_1 \cdots \tau_n \to \sigma$. Let $t_1, \cdots, t_n$ be terms with types $\tau_1, \cdots \tau_n$. Let ϕ be a formula such that $\{f(t_1, \cdots, t_n)\} = Calls_F(f, \phi)$, that is, $f(t_1, \cdots, t_n)$ is the only unique call to f in formula ϕ.

We will use c_f to denote $f(t_1, \cdots, t_n)$ from now on.

Assume that f always terminates, that is,

$$\forall t'_1 : \tau_1. \cdots \forall t'_n : \tau_n. \exists r : \sigma. f(t'_1, \cdots, t'_n) = r$$

Then, assume that

$$\forall r : \sigma. [r = c_f \to [r/c_f]\phi] \tag{8.4}$$

Because f always terminates, we can take some r' such that $r' = c_f$.
From the assumption 8.4, we have $[r'/c_f]\phi$. From the assumption that $r' = c_f$, we have ϕ.
Therefore, $\forall r : \sigma. [r = f(t_1, \cdots, t_n) \to [r/f(t_1, \cdots, t_n)]\phi] \to \phi$.

□

8.3 Unfolding of Induction Markers

By definition 6.31, $Unfold$ will only alter (sub)formulae that are induction markers, by applying either $InductionPrinciple$ or $FunctionInductionPrinciple$ to the formula in the induction marker. Since we have proven that those two functions are sound under certain conditions, we can prove that $Unfold$ is sound under those conditions, by straightforward induction.

8.4 Non-terminating Functions

The above proofs about soundness of induction over function definitions are written under the assumption that f, the function we apply induction on, terminates. If f does not terminate, then we could prove incorrect results from induction.

Consider the following definition of f:

```
f :: Int -> Int
f x | x < 10     =  10
    | otherwise =  f (x + 1)
```

$$\langle \texttt{f-def}, f(x), (\langle x < 10, 10 \rangle, \langle \top, f(x+1) \rangle) \rangle$$

Say we have some $\Gamma \vdash x : \texttt{Int}$, and a formula

$$\phi = f(x) > 0 \tag{8.5}$$

We cannot prove ϕ because x could be greater than or equal to 10, which would make ϕ meaningless.

However, we could generate the following induction principle for ϕ:

$$\begin{aligned} &FunctionInductionPrinciple(\Gamma, d, \phi) = \\ &\quad (x < 10 \rightarrow 1 > 0) \;\; \wedge \;\; \forall r : \texttt{Int}.[\neg x < 10 \;\; \rightarrow \;\; r > 0 \wedge r = f(x+1) \;\; \rightarrow \;\; r > 0] \end{aligned} \tag{8.6}$$

Since if $x \geq 10$, $f(x+1)$ is undefined, $r > 0 \wedge r = f(x+1)$ cannot be true. Hence the induction step in the induction principle is vacuously true, which means that the entire induction principle is true, although the original formula ϕ cannot be proven to be true.

Chapter 9

Evaluation

In this chapter, we provide an evaluation of Emmy in relation to the various objectives of the project. Namely, we will focus on how the capabilities of Emmy impact the use of Emmy in the teaching of the reasoning about programs course, making reference to course materials when relevant.

9.1 Language Support

Emmy supports the following language features:

- Formula and terms in propositional and (many-sorted) first-order logic.
- Declaration of custom types and (non-mutually recursive) data structures. which allows as to express most basic data structures that are used in Haskell.
- Strong, static typing. This allows Emmy to ensure the well-formedness formulae a proof before checking the proof.
- Recursive function definitions.
- Pattern matching in function definitions, which is crucial when defining functions about data structures.
- Guards in function definitions, which is useful when defining recursive functions.
- `if` expressions.
- Built-in support for integers and basic arithmetic operations, which is built upon the SMT solvers' native support for arithmetic.

Some important language features that Emmy *cannot* express directly are:

- Integer division, which is not total.
- Declaration and reasoning of mutually recursive data structures, like:

  ```
  data Yin  = Yin1  | Yin2 Yang
  data Yang = Yang1 | Yang2 Yin
  ```

- Generic data structures and functions, like:

```
data GenericList a = GenericNil | GenericCons a GenericList
```

and

```
id :: a -> a
```

- Higher-order functions that take other function as it arguments, such as

  ```
  map :: (a -> b) -> [a] -> [b]
  filter :: (a -> Bool) -> [a] -> [a]
  ```

  Notice that these two functions are also generic functions.

- Lambda expressions, like `\x -> x + 1`.

- Shorthand notations, including list construction (`[1, 2, 3]`) and pair construction (`(1, 2)`).

Although Emmy can only express a small subset of the Haskell language, it is already enough for expressing a large range of data structures, functions, and algorithms. As we will show in the next section, Emmy is in fact capable of expressing most theorems a student may encounter in the reasoning course. However, due to the lack of generic data structures, the expression of certain theorems may be cumbersome.

We will discuss how some of the missing features can be implemented in section 10.1.

9.2 Proving Power

We now demonstrate the proving power of Emmy, by using Emmy to prove theorems that appeared in the course materials. In this way, we surveyed the extent to which Emmy can help with the teaching and learning of the reasoning about programs course.

We identified 26 theorems in total, and proved 24 of them. The proofs were checked twice, first time using Z3 version 4.8.5, and then using CVC4 version 1.6 as the underlying SMT solver, with a 200ms time limit per call. The full set of theorems are listed in appendix B.1, and the proofs, written in the `#lang` Emmy language (see section 7.4), are placed in the `examples` folder in the code directory.

9.2.1 Logic

Although Emmy focuses on proving properties of computer programs, it is also capable of proving theorems in propositional and first-order logic. We took 3 propositional theorems[1] from a tutorial sheet for the logic course [74], and an additional 5 first-order theorems [2] from another tutorial sheet [52] that are difficult or tricky to prove using natural deduction, and wrote proofs of them using Emmy.

We followed the principles listed below when formulating and proving the theorems in Emmy:

- The first letters in the names of all propositions and predicates are capitalised.

[1] Question 2g, 2k, and 2m from the tutorial sheet. We chose these questions because they were supposed to be discussed during tutorial sessions.

[2] Queston 51 to 54, and question 57. We omitted 55 and 56 because they are more about translating statements than writing proofs.

- To prove a theorem of the form

  $$\phi_1, \cdots, \phi_n \vdash \phi'$$

  we first write formulae $\phi_1, \cdots, \phi_n$ as steps that are '`Given`' to be true. Then, we prove ϕ' using these '`Given`' steps.

- For first-order theorems, we use `Atom` as the type for all terms.

For each theorem, we produce at least two proofs: one longer proof in 'natural deduction style', and another short one where the goal follows directly from the premises. We then try to produce new proofs by removing steps from the longer proof. We wish to show that our tool is not only capable of proving these theorems, but also capable of handling proofs in different styles, including cases where we do not follow strictly natural deduction rules, and rely on our intuition instead.

Results

We proved all 8 theorems using both SMT solvers. In addition, we have shown that all the theorems can be proven by EMMY directly in one step.

We have also shown that all 8 theorems can be proven by stylised proofs that are shorter than the 'natural deduction style' proofs, without sacrificing correctness. We argue that the automated checking power of EMMY may alleviate the tediousness of writing proofs in 'pure' natural deduction.

It worths mentioning that many of the theorems involves properties of functions in discrete mathematics. For example, theorem B.5 involves the property of injective functions, and theorem B.7 proves that functions cannot be one-to-many or many-to-many. While we did not intend to use EMMY in the teaching of discrete mathematics, we now believe that there is a potential for such application.

9.2.2 Reasoning about Program

To demonstrate the proving power of EMMY in relation to the syllabus of the reasoning about programs course, we identified some theorems about the properties of functions and data structures that appeared in the course materials, and then proved of them using EMMY.

There are also questions and examples about finding the induction principle or writing down the proof schema in the course materials. However, it is not clear how to show that our tool can be used for such tasks. We therefore focus on proving the theorems we identified instead.

Due to the various limitations of EMMY's logic, many theorems in the course materials cannot be translated into EMMY's language directly. We tried to preserve the meaning of the original theorems as much as we could, and added extra constraints such as the expected range of function arguments, in cases where the original theorem is not clear enough. In general, we followed the following rules:

- A generic list with Haskell type `[a]` will become a list of integers in our proof, with *nil*, which constructs an empty list and corresponds to `[]` in Haskell, and *cons*, which takes one integer and another list, and corresponds to the `:` operator in Haskell, as its two constructors.

  For example, the Haskell list `[1, 2, 3]`, which is equal to `1 : 2 : 3 : []`, would become $cons(1, cons(2, cons(3, nil)))$ in our theorems.

Lists of specific types are defined individually, with suffixes added to type names and constructor names where ambiguity might arise. For example, for theorem B.20, we declared two list types, `ListC` and `ListT`, as lists of `Code`s and lists of `Tree`s.

- The ++ infix operator in Haskell, which contatenates two lists, are represented by an *append* function, defined as:

$$append(nil, ys) = ys$$
$$append(cons(x, xs)) = cons(x, append(xs, ys))$$

  which is the same as the definition of ++ in the Haskell prelude [22].

  For example, the Haskell expression `xs ++ ys` would become $append(xs, ys)$.

  When there are lists of specific types, we also define individual *append* functions for those list types.

- We define a `Bool` type to stand for booleans, with t and f as its two atomic constructors. We use $b = t$ to mean 'b is true', and $b = f$ to mean 'b is false'.

  Logic operators (connectives) are declared and defined in the usual manners when they are needed.

- A `Pair` data structure would be declared in cases where they are needed. The type of its two elements depends on the theorem. The fst and snd accessors would also be declared and defined if needed.

- All natural numbers, except those in proofs about Peano numbers, were given the `Int` type, and an extra assumption that they are non-negative.

  For example, the formula $\forall x : \mathbb{N}.P(x)$ would become $\forall x : \texttt{Int}.[x \geq 0 \rightarrow P(x)]$.

- All natural numbers that occurs in theorems in terms of zeros and successor functions are expressed by a `Nat` type, which corresponds to Peano numbers, with two constructors, z and s.

- All non-total function definitions are replaced with total ones that behave the same in the 'correct' cases.

Some of the questions in the course materials comes with a set of lemmas. We use them when applicable.

We found 18 theorems in the lecture notes, exercises, and tutorial sheets of the reasoning about programs course. The majority of them involve inductive property or function definitions, and require explicit application of induction to prove. None of the theorems contain higher-order functions, which Emmy cannot express.

For each theorem that can be expressed using Emmy, we first check whether or not Emmy can prove them directly from the definitions without performing any induction or supplying any additional lemmas not included in the course material. Then, we write a proof for it by hand.

Results

We proved 16 of the 18 theorems. The below table gives a summary of the results:

Theorem number	Z3 Proven?	Z3 Directly?	CVC4 Proven?	CVC4 Directly?
Theorem B.9	✓		✓	✓
Theorem B.10	✓		✓	✓
Theorem B.11	✓		✓	
Theorem B.12	✓		✓	✓
Theorem B.13	✓		✓	
Theorem B.14	✓		✓	✓
Theorem B.15	✓		✓	✓
Theorem B.16	✓		✓	
Theorem B.17	✓			
Theorem B.18	✓		✓	✓
Theorem B.19	✓		✓	
Theorem B.20	✓		✓*	
Theorem B.21	✓		✓	
Theorem B.22	✓		✓	
Theorem B.23				
Theorem B.24				
Theorem B.25	✓		✓	✓
Theorem B.26	✓	✓	✓	✓
Count: 18	16	1	15*	8

Table 9.1: The results from proving the reasoning about programs theorems.

Theorem B.20 cannot be proven when we use CVC4 when using a 200ms time limit. It could be proven if we raise the limit to 2000ms.

We were unable to prove two of the theorems, for the following reasons:

Theorem B.23 is about the property of a power function, defined in terms of exponentiation in mathematics, which is not available 'natively' in the SMT solvers we use. This means that we have to write our own exponent function if we want to express the proof, which, we believe, defeats the purpose of proving the property of a power function.

Theorem B.24 requires induction on the definitions of two functions, but Emmy only supports induction on the definition of one function at a time.

When we use CVC4, theorem B.17 cannot be proven. This is likely related to CVC4's handling of data structure properties, which we will discuss in section 9.3.1.

Most of our proofs are short: we can either prove the theorem in one induction step, or prove a lemma in one step first, and then show that the goal follows directly from the lemma. Only a few rather complicated theorems required more than that to prove.

From the above results, we draw the following conclusions:

- Emmy is capable of proving most theorems a student may encounter in the reasoning course. In particular, it can prove all data structure related theorems that appear in the course materials.
- Z3 is perhaps more suitable for Emmy. as can be seen in the results, it proves

9.3 Proving Powers of SMT Solvers

The proving powers of the underlying SMT solvers have a large impact on the power of Emmy. The more powerful the SMT solvers are, the fewer steps we need to write in order to obtain our goal.

We shall now discuss some issues with the SMT solvers' proving power we discovered during the course of the development of Emmy, and their relation with Emmy's functionality. We will also compare the two solvers we use with regard to these issues

9.3.1 Data Structure Properties

Consider this property, defined upon the definition of `List` described in section 9.2.2:

Theorem 9.1. Non-empty list must be constructed from an integer and a list

$$\forall xs : \texttt{List}.[xs \neq nil \quad \rightarrow \quad \exists y : \texttt{Int}.\exists ys : \texttt{List}.xs = cons(y, ys)]$$

While theorem 9.1 itself is not very interesting, it can be used to prove theorems such as

Theorem 9.2. Empty list is the only left identity of *append*

$$\forall xs : \texttt{List}.\forall ys : \texttt{List}.[append(xs, ys) = ys \quad \leftrightarrow \quad xs = nil]$$

However, we found out that it is impossible to prove theorem 9.1 using CVC4 as the underlying solver [3]. The solver would keep running, until the time limit is reached and it returns `unknown`. As a result, certain theorems related to theorem 9.2, such as step `I4` in the proof of theorem B.17, is not provable when using CVC4.

Z3, however, can prove theorem 9.1 directly.

One possible way to get around this issue is to generate lemmas like theorem 9.1 for all non-atomic data structure constructors, and include all of them in entailment checks.

9.3.2 Induction over Data Structure

In general, SMT solvers have limited inductive reasoning capabilities. Many of them are unable to prove anything that requires non-trivial induction [72]. It was clearly stated in the Z3 guide that Z3 'will not prove inductive facts' [46]. CVC4, however, has implemented support for inductive reasoning [72] [58].

The vast majority of the theorems we proved in section 9.2.2 are inductive. As can be seen in table 9.1, when using Z3, Emmy can only prove one of the theorems directly, but when using CVC4, 8 can be proven directly.

Most of the inductive theorems that cannot be proven directly even when using CVC4 involves the use of lemmas. We speculate that although CVC4 can perform induction over data structures, it cannot effectively generate lemmas when proving.

However, we argue that the inductive reasoning power of the SMT solvers is not very important for Emmy, because Emmy can generate induction principles before calling the SMT solvers. Also, we have shown that Emmy is already capable of proving most theorems when using Z3, which does not support induction, as the underlying solver.

[3] Curiously, while theorem 9.1 cannot be proven when using CVC4, an equivalent theorem can be proven by CVC4 directly:

Theorem 9.3. Empty list is the only left identity of *append* [Variant of theorem 9.2]

$$\forall xs : \texttt{List}.\forall ys : \texttt{List}.[append(xs, ys) = ys \quad \leftrightarrow \quad xs = nil]$$

9.3.3 Entailment between Equivalent Formulae

We identified some pairs of formulae that are semantically equivalent but bear trivial syntactic differences, where Z3 cannot prove that one formula entails another.

One instance, encountered when proving theorem B.25, is that for the following semantically equivalent formulae

$$\phi = \forall k : \texttt{Nat}.\forall i : \texttt{Nat}.\forall j : \texttt{Nat}.[k = min(i,j) \to k = min(j,i)]$$
$$\phi' = \forall i : \texttt{Nat}.\forall j : \texttt{Nat}.\forall k : \texttt{Nat}.[k = min(i,j) \to k = min(j,i)]$$

where the only difference between ϕ and ϕ' is the ordering of quantified formulae, the entailment

$$\Gamma, \Delta, \phi \vdash \phi'$$

does not hold, when checked using Z3.

Another instance was encountered when proving the following theorem:

Theorem 9.4. Length of concatenation of two lists is the same as sum of two lists' length

$$\forall l1 : \texttt{List}.\forall l2 : \texttt{List}.length(append(l1,l2)) = length(l1) + length(l2)$$

A proof of theorem 9.4 is available in `example-data-3.prf`.

The proof would require induction. The induction principle of theorem 9.4 could be

$$\begin{aligned}
&\forall l2 : \texttt{List}.length(append(nil,l2)) = length(nil) + length(l2) \\
&\quad\wedge \\
&\forall l1 : \texttt{List}.\Big[\forall l2 : \texttt{List}.length(append(l1,l2)) = length(l1) + length(l2) \\
&\qquad\to \\
&\quad\forall i : \texttt{Int}.\forall l2 : \texttt{List}.length(append(cons(i,l1),l2)) = length(cons(i,l1)) + length(l2)\Big]
\end{aligned} \tag{9.1}$$

If we swap the order of the base case and inductive step, and rename the quantified variable $l1$ in the inductive step, we may obtain the semantically equivalent formula:

$$\begin{aligned}
&\forall l1' : \texttt{List}.\Big[\forall l2 : \texttt{List}.length(append(l1',l2)) = length(l1') + length(l2) \\
&\qquad\to \\
&\quad\forall i : \texttt{Int}.\forall l2 : \texttt{List}.length(append(cons(i,l1'),l2)) = length(cons(i,l1')) + length(l2)\Big] \\
&\quad\wedge \\
&\forall l2 : \texttt{List}.length(append(nil,l2)) = length(nil) + length(l2)
\end{aligned} \tag{9.2}$$

If we denote formula 9.2 by ϕ and formula 9.1 by ϕ', then the entailment

$$\Gamma, \Delta, \phi \vdash \phi'$$

cannot be proven when using Z3.

Interestingly, if we add the (translated) definitions of *length* and *append* to the set of premises to the left of the turnstile, then the entailment will be deemed correct, although it should be correct regardless of the definitions of the functions. Also, the entailment can be proven when we only rename the quantified variable, or when we only switch the order of cases, but not when we do both.

We provide the SMT-LIB scripts generated when proving all variations of the second example, the output from running the SMT solvers, along with a small summary of results, in appendix C.

Entailments in both the first and second example can be proven when we use CVC4.

We only identified and documented two such cases, but we believe that similar cases are very common. The users may become frustrated when the equivalence between two obviously equivalent formulae cannot be checked. There is also a problem with inductions: the user could have written the correct induction principle, but EMMY may consider it incorrect because the aforementioned problems.

One way to partially solve this problem is to normalise formulae, so that such pairs of syntactically similar formulae can be represented by the same normal form. This is one of the future extensions of EMMY listed in section 10.1.

9.4 Non-terminating Functions

In section 5.4.1, we have shown how we could define non-terminating functions in EMMY. Consider the following definition:

```
f :: Int -> Int
f x | x < 10     =  10
    | otherwise =  f (x + 1)
```

$$\langle \texttt{f-def}, f(x), (\langle x < 10{,}10\rangle, \langle \top, f(x+1)\rangle)\rangle$$

It is obvious that f does not terminate for any argument $x \geq 10$, so we could say that if $x \geq 10$, then $f(x)$ is undefined, hence the following property is false:

$$\forall x : \texttt{Int}. \exists r : \texttt{Int}. r = f(x) \tag{9.3}$$

However, when the translated function definition `f-def` is used as a justification, EMMY would prove formula 9.3 to be true.

This counterintuitive result stems from how we translate function definitions: the definition `f-def` would be represented, according to definition 6.6, by the property:

$$\forall x : \texttt{Int}. f(x) = \texttt{if}(x < 10, 10, f(x+1)) \tag{9.4}$$

which is equivalent to

$$\forall x : \texttt{Int}. \big[(x < 10 \to f(x) = 10) \;\; \wedge \;\; (x \geq 10 \to f(x) = f(x+1)\big] \tag{9.5}$$

The property 9.5 only says that $f(x) = f(x+1)$ when $x \geq 10$. If we define $f(x)$ to be 0 for all $x \geq 10$, as

```
f' :: Int -> Int
f' x | x < 10     =  10
     | otherwise =  0
```

$$\langle \texttt{fp-def}, f'(x), (\langle x < 10{,}10\rangle, \langle \top, 0\rangle)\rangle$$

then 9.5 would also be satisfied.

Since property 9.4, the generated property that should represent the definition of f, does not forbid such an interpretation, EMMY would consider 9.3 to be true.

We have shown that the semantics of the definition of a non-terminating recursive function in EMMY deviates from the same definition in programming languages, and the fact that

property 9.3 can be proven by Emmy is the consequence of such a deviation. To mitigate this issue, the users can write definitions that ensures termination. Alternatively, they could also add constraints to the arguments in their arguments, so that the function always terminates if the constraints are satisfied:

$$\forall x : \texttt{Int}.\exists r : \texttt{Int}.[x < 10 \to r = f(x)]$$

9.5 User Feedback

We invited students from two tutorial groups to test-use the web interface of Emmy, and received two responses from them. Due to the very small number of participants, no quantitative studies were conducted.

The two test-users are both first-year students who have very recently sat the reasoning about programs exam, and are therefore familiar with the course material. We did not provide any prior information about Emmy to the test-users.

9.5.1 Procedure

During test-using, we first showed the test-users an example proof. We walked through the proof, showing the syntax for writing declarations, definitions, and formulae using the online interface, as well as some of the basic operations, such as editing formulae and checking proofs.

We then showed the test-users a question about theorem B.12, which is about the property of a recursive function over inductive data structures, and requires structural induction to prove. We asked the test-users to prove this theorem using Emmy. Two lemmas, namely $add(i, 0) = i$, and $add(s(i), j) = add(i, s(j))$, could be used without proving. We did not provide extensive guidance when the test-users were writing the proofs, but helps were offered when they were stuck on specific UI issues.

9.5.2 Results

We observed the following from the test-users:

- Both test-users produced correct proofs for theorem B.12. They both used one induction step with one base case, and one induction step.

- The base case, which follows from a lemma we gave them, was proven by both test-users directly in one step.

  For the inductive step, the test-users used multiple steps to obtain the equality $add(succ(i), j) = add(j, succ(i))$.

- The syntax for writing declaration using the web interface, which is similar to Haskell syntax, but differs in cases such as the naming of data structure constructors, the use of single instead of double colons for typing statements, caused minor confusions among the test users, but they were quick to adapt to our syntax.

- They learned the syntax for writing formulae using the web interface, which is very similar to the syntax of normal logic formula, quickly. Some points of confusion include the dot following the typing statement in a quantification, and whether or not brackets are needed in some cases.

- The various buttons used for adding steps, induction cases, et cetra. were confusing, and their functionalities were hard to discover.

- One test-user commented that it would be better if EMMY could check proofs and update the result as he worked through the proof, which is currently infeasible as proof checking takes a long time.

Although both test-users used multiple steps in the inductive step to prove $add(succ(i), j) = add(j, succ(i))$, EMMY can actually prove it directly. We speculate that the test-users were not aware of the full proving power of EMMY, so their proofs were more conservative than necessary.

One test-user who had more free time proceeded to prove theorem B.18. He was able to prove it correctly without much troubles.

9.5.3 Improvements Made in Response to User Feedback

Many UI bugs and imperfections were identified during the tests and later fixed. This includes:

- The user may edit more than one elements at the same time. For example, they may be editing both the formula and number of a step at once. However, if they finish editing any one of the elements, their edits made to the other elements would be lost.

  This was fixed by making all text inputs save their content once the user stops editing them, by, for example, clicking elsewhere.

- There were buttons that adds the wrong types of steps to the proof. They were fixed by changing the arguments supplied to relavent functions.

We also added some new features as response to how the test-users used EMMY. For example, we added a ‘copy’ button that copies the raw string of a formula, after noticing the test-users clicking on the formulae and copying the raw strings from the editor.

Chapter 10

Conclusion

In this project, we have developed a proof system, defined formally its syntax and semantics, and built an implementation of the said system as a computer program. The result is Emmy, a proof assistant that strives to provide the full power of automated reasoning tools to students who have just starting learning to reason about programs.

Despite a number of limitations in terms of expressiveness and automatic proving power, we have demonstrated that Emmy is sufficiently powerful to prove most theorems students may encounter when learning the reasoning about programs course. From the test-using results of Emmy, we are also confident that Emmy has met our expectation that it should be approachable and easy to learn.

We believe that Emmy is now in a stable state, and is ready to be deployed in the teaching of the reasoning about program course. We are excited to be able to bring Emmy into existence, and we are looking forward to its use and appreciation by future users.

10.1 Future Works

Here we suggest some features that may be added to Emmy by future contributors.

Induction over Relations Apart from induction over data structures and function definitions, we may also implement induction over recursively defined relations, or predicates. Consider the following predicate *Odd*, define for natural (Peano) numbers:

$$Odd(s(z))$$
$$\forall x : \texttt{Nat}.[Odd(x) \rightarrow Odd(s(s(x)))]$$

Then, if we want to prove that some property P holds for all odd numbers:

$$\forall x : \texttt{Nat}.P(x)$$

We may perform induction, and obtain the following induction principle.

$$\begin{array}{l} P(s(z)) \\ \quad\wedge \\ \forall x : \texttt{Nat}.[P(x) \wedge Odd(x) \quad \rightarrow \quad P(s(s(x)))] \end{array}$$

Notice that before we implement induction principle generation for induction over relations, we also need to be able to define relations. Both features should be straightforward to implement, as they are similar to function definitions and their inductions.

Higher-order Functions Functions in Emmy are first-order, they neither take functions as arguments not return functions. Common higher-order Haskell functions, such as `map` and `foldl`, cannot be expressed in our system.

One way to extend Emmy to include higher-order functions without changing the underlying SMT solver is to apply defunctionalisation: [20] we first turn functions into terms, and then use a special 'apply' function to carry out function application. Alternatively, we may use a higher-order SMT solver [4] [13], in which higher-order functions can be expressed directly.

While we considered adding support for higher-order functions during the course of the project, we decided to omit this feature due to time constraints and the fact that none of the theorems in the course materials involve higher-order functions.

Generic Types Currently, Emmy does not have any support for generic types. When we need generic versions of certain data structures and functions, we must declare them individually for each type we need. For example, when proving theorem B.20, we had to define `List` data structures and related operations for each concrete `List` type we intend to use.

Since the most recent version of the SMT-LIB standard already supports type parameters in data structures [6], declaration of generic data structures can be expressed easily. However, there is no support for genric functions. We may need to generate concrete instances of generic functions when translating our programs to SMT-LIB scripts.

Formula Normalisation Currently, when checking proofs, we do not alter the structure of formulae in any way except for the unfolding of induction principles. However, we may develop a way to 'normalise' formulae, so that two formulae that are semantically equivalent, but have minor syntactic differences, can be expressed by the same *normal form*.

For example, we may define an ordering of quantifications, and normalise nested quantifications by sorting them in ascending order. Suppose we order quantification by the name of the quantified variable, then we may have something like:

$$Normalise(\forall b:\tau.\forall a:\tau.\forall c:\tau.\phi) = \forall a:\tau.\forall b:\tau.\forall c:\tau.\phi$$
$$Normalise(\forall c:\tau.\forall b:\tau.\forall a:\tau.\phi) = \forall a:\tau.\forall b:\tau.\forall c:\tau.\phi$$
$$Normalise(\forall a:\tau.\forall b:\tau.\forall c:\tau.\phi) = \forall a:\tau.\forall b:\tau.\forall c:\tau.\phi$$

This could mitigate, though not eliminate, some of the problems we identified in section 9.3.

Incremental Proof Checking The current proof checking implementation checks an entire program at once, which may take a long time when the program is long, or if there are unprovable formulae that causes the SMT solver to timeout. This makes it prohibitive to provide 'live' updates about the correctness of individual steps.

An incremental approach to proof checking would allow us to check individual steps, without touching other parts of the proof. This might require a stateful proof checking implementation, that stores the previously checked proof, and only check the parts that have been changed since last check. This would result in higher CPU and memory load for the server, but would provide a more seamless experience for users of the web interface.

Part IV

Bibliography and Appendices

Bibliography

[1] Michael Ameri and Carlo A. Furia. "Why Just Boogie?" In: *Integrated Formal Methods*. Ed. by Erika Ábrahám and Marieke Huisman. Cham: Springer International Publishing, 2016, pp. 79–95. ISBN: 978-3-319-33693-0.

[2] Michael Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 135–150. ISBN: 978-3-642-25379-9.

[3] David Aspinall. "Proof General: A Generic Tool for Proof Development". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Susanne Graf and Michael Schwartzbach. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 38–43. ISBN: 978-3-540-46419-8.

[4] Haniel Barbosa et al. "Extending SMT Solvers to Higher-Order Logic". In: 2019.

[5] Mike Barnett et al. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387. ISBN: 978-3-540-36750-5.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2017.

[7] Clark Barrett, Leonardo de Moura, and Aaron Stump. "SMT-COMP: Satisfiability Modulo Theories Competition". In: *Computer Aided Verification*. Ed. by Kousha Etessami and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 20–23. ISBN: 978-3-540-31686-2.

[8] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.0*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2010.

[9] Clark Barrett et al. "6 Years of SMT-COMP". In: *Journal of Automated Reasoning* 50.3 (Mar. 2013), pp. 243–277. ISSN: 1573-0670. DOI: 10.1007/s10817-012-9246-5. URL: https://doi.org/10.1007/s10817-012-9246-5.

[10] Clark Barrett et al. "CVC4". In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Snowbird, Utah. Springer, July 2011, pp. 171–177. URL: http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf.

[11] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* 2004. ISBN: 3540208542. DOI: 10.1007/978-3-662-07964-5.

[12] Simon Blackburn, ed. *The Oxford Dictionary of Philosophy*. 2 rev ed. Oxford University Press, 2008.

[13] Sascha Böhme. "Proving Theorems of Higher-Order Logic with SMT Solvers". Dissertation. München: Technische Universität München, 2012.

[14] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda – A Functional Language with Dependent Types". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78. ISBN: 978-3-642-03359-9.

[15] Krysia Broda et al. "Pandora: A Reasoning Toolbox using Natural Deduction Style". In: *Logic Journal of the IGPL* 15.4 (2007), pp. 293–304. DOI: 10.1093/jigpal/jzm020. eprint: /oup/backfile/content-public/journal/jigpal/15/4/10.1093/jigpal/jzm020/2/jzm020.pdf. URL: http://dx.doi.org/10.1093/jigpal/jzm020.

[16] R. M. Burstall. "Proving Properties of Programs by Structural Induction". In: *The Computer Journal* 12.1 (1969), pp. 41–48. DOI: 10.1093/comjnl/12.1.41. eprint: /oup/backfile/content-public/journal/comjnl/12/1/10.1093/comjnl/12.1.41/2/12-1-41.pdf. URL: http://dx.doi.org/10.1093/comjnl/12.1.41.

[17] Samuel R. Buss. "On Herbrand's theorem". In: *Logic and Computational Complexity*. Ed. by Daniel Leivant. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 195–209. ISBN: 978-3-540-44720-7.

[18] M. Clint and C. A. R. Hoare. "Program proving: Jumps and functions". In: *Acta Informatica* 1.3 (Sept. 1972), pp. 214–224. ISSN: 1432-0525. DOI: 10.1007/BF00288686. URL: https://doi.org/10.1007/BF00288686.

[19] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.7*. Oct. 2017. URL: http://coq.inria.fr.

[20] Olivier Danvy and Lasse R. Nielsen. "Defunctionalization at Work". In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '01. Florence, Italy: ACM, 2001, pp. 162–174. ISBN: 1-58113-388-X. DOI: 10.1145/773184.773202. URL: http://doi.acm.org/10.1145/773184.773202.

[21] *Data.Bool*. URL: http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Bool.html#v:otherwise.

[22] *Data.List*. URL: http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-List.html.

[23] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: http://doi.acm.org/10.1145/321033.321034.

[24] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: Introduction and Applications". In: *Commun. ACM* 54.9 (Sept. 2011), pp. 69–77. ISSN: 0001-0782. DOI: 10.1145/1995376.1995394. URL: http://doi.acm.org/10.1145/1995376.1995394.

[25] "Defining new #lang Languages". In: *The Racket Guide*. Chap. 17.3. URL: https://docs.racket-lang.org/guide/hash-languages.html.

[26] David Delahaye. "A Tactic Language for the System Coq". In: *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*. LPAR'00. Reunion Island, France: Springer-Verlag, 2000, pp. 85–95. ISBN: 3-540-41285-9. URL: http://dl.acm.org/citation.cfm?id=1765236.1765246.

[27] Sophia Drossopoulou and Mark Wheelhouse. *Inductively Defined Functions*. 2019.

[28] Sophia Drossopoulou and Mark Wheelhouse. *Structural induction over Haskell data types*. 2019.

[29] Sophia Drossopoulou and Mark Wheelhouse. *Stylised Proofs*. 2019.

[30] Sophia Drossopoulou and Mark Wheelhouse. *Week 4 Assessed PMT - Structural Induction*. 2019.

[31] Sophia Drossopoulou and Mark Wheelhouse. *Week 4 PMT - Structural Induction*. 2019.

[32] Sophia Drossopoulou and Mark Wheelhouse. *Week 4 Tutorial - Structural Induction*. 2019.

[33] Sophia Drossopoulou and Mark Wheelhouse. *Week 5 Assessed PMT - Structural Induction*. 2019.

[34] Sophia Drossopoulou and Mark Wheelhouse. *Week 5 PMT - Induction over Recursively Defined Sets, Relations and Functions*. 2019.

[35] Sophia Drossopoulou and Mark Wheelhouse. *Week 5 Tutorial - Induction over Recursively Defined Relations*. 2019.

[36] Bruno Dutertre. “Yices 2.2”. In: *Computer-Aided Verification (CAV'2014)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, July 2014, pp. 737–744.

[37] Burak Ekici et al. “SMTCoq: A Plug-In for Integrating SMT Solvers into Coq”. In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 126–133. ISBN: 978-3-319-63390-9.

[38] *Emacs Mode —Agda 2.5.4.2 documentation*. 2018. URL: https://agda.readthedocs.io/en/v2.5.4.2/tools/emacs-mode.html.

[39] *Express - Node.js web application framework*. URL: https://expressjs.com/.

[40] Daniel P. Friedman and David Thrane Christiansen. *The Little Typer*. The MIT Press, 2018.

[41] Daniel P. Friedman and Carl Eastlund. *The Little Prover*. The MIT Press, 2015.

[42] Jean H Gallier. *Logic for computer science : foundations of automatic theorem proving*. eng. Harper & Row computer science and technology series. New York ; London: Harper & Row, 1986. ISBN: 0060422254.

[43] Olivier Gasquet, François Schwarzentruber, and Martin Strecker. “Panda: A Proof Assistant in Natural Deduction for All. A Gentzen Style Proof Assistant for Undergraduate Students”. In: *Tools for Teaching Logic*. Ed. by Patrick Blackburn et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 85–92. ISBN: 978-3-642-21350-2.

[44] Gerhard Gentzen. “Investigations into Logical Deduction”. In: *American Philosophical Quarterly* 1.4 (1964), pp. 288–306. ISSN: 00030481. URL: http://www.jstor.org/stable/20009142.

[45] Gerhard Gentzen. “Untersuchungen über das logische Schließen. I”. In: *Mathematische Zeitschrift* 39.1 (Dec. 1935), pp. 176–210. ISSN: 1432-1823. DOI: 10.1007/BF01201353. URL: https://doi.org/10.1007/BF01201353.

[46] *Getting Started with Z3: A Guide*. URL: https://rise4fun.com/z3/tutorial.

[47] *GitHub - the-little-prover/j-bob*. URL: https://github.com/the-little-prover/j-bob.

[48] *GitHub - the-little-typer/pie: The Pie language, which accompanies The Little Typer by Friedman and Christiansen*. 2018. URL: https://github.com/the-little-typer/pie.

[49] Michael J. Gordan, Arthur J. Milner, and Christopher Wadsworth. *Edinburgh LCF*. Lecture Notes in Computer Science. Springer, 1979. ISBN: 978-3-540-09724-2. DOI: https://doi.org/10.1007/3-540-09724-4.

[50] Tim Green et al. *iProve - Final Report*. 2019.

[51] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[52] Ian Hodkinson. *Extra logic exercises*. 2018.

[53] Alfred Horn. “On Sentences Which are True of Direct Unions of Algebras”. In: *The Journal of Symbolic Logic* 16.1 (1951), pp. 14–21. ISSN: 00224812. URL: http://www.jstor.org/stable/2268661.

[54] *How do I enable Java in my web browser?* URL: https://java.com/en/download/help/enable-browser.xml.

[55] Michael Huth and Mark Ryan. *Logic in Computer Science. Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[56] Stanisław Jaśkowski. “On the Rules of Suppositions in Formal Logic”. In: *Itepmccall1967*. Oxford at the Clarendon Press, 1934.

[57] M. Kaufmann and J. Strother Moore. “ACL2: an industrial strength version of Nqthm”. In: *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96*. June 1996, pp. 23–34. DOI: 10.1109/CMPASS.1996.507872.

[58] K. Rustan M. Leino. "Automating Induction with an SMT Solver". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Viktor Kuncak and Andrey Rybalchenko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 315–331. ISBN: 978-3-642-27940-9.

[59] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: -78-3-642-17511-4.

[60] J. Mccarthy. "Towards a Mathematical Science of Computation". In: *In IFIP Congress*. North-Holland, 1962, pp. 21–28.

[61] John McCarthy. "A Basis For a Mathematical Theory of Computation". In: *Computer Programming and Formal Systems*. Ed. by Paul Braffort and David Hirschberg. North Holland, 1963, pp. 33–70.

[62] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782. DOI: `10.1145/367177.367199`. URL: `http://doi.acm.org/10.1145/367177.367199`.

[63] Matthew W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Annual Design Automation Conference*. DAC '01. Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2. DOI: `10.1145/378239.379017`. URL: `http://doi.acm.org/10.1145/378239.379017`.

[64] Leonardo de Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: An Appetizer". In: *Formal Methods: Foundations and Applications*. Ed. by Marcel Vinícius Medeiros Oliveira and Jim Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–36. ISBN: 978-3-642-10452-7.

[65] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[66] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Edinburgh LCF*. Lecture Notes in Computer Science. Springer, 2002. ISBN: 978-3-540-43376-7. DOI: `https://doi.org/10.1007/3-540-45949-9`.

[67] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3-540-43376-7.

[68] Giuseppe Peano. *Arithmetices principia: nova methodo exposita*. Fratres Bocca, 1889.

[69] Francis Jeffry Pelletier and Allen P. Hazen. "A History of Natural Deduction". In: *Logic: A History of its Central Concepts*. Ed. by Dov M. Gabbay, Francis Jeffry Pelletier, and John Woods. Vol. 11. Handbook of the History of Logic. North-Holland, 2012, pp. 341–414. DOI: `https://doi.org/10.1016/B978-0-444-52937-4.50007-1`. URL: `http://www.sciencedirect.com/science/article/pii/B9780444529374500071`.

[70] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, 1965.

[71] *React –A JavaScript library for building user interfaces*. URL: `https://reactjs.org/`.

[72] Andrew Reynolds and Viktor Kuncak. "Induction for SMT Solvers". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 80–98. ISBN: 978-3-662-46081-8.

[73] Andrew Reynolds and Viktor Kuncak. *Induction for SMT Solvers (VMCAI 2015 submission)*. URL: `http://lara.epfl.ch/~reynolds/VMCAI2015-ind/`.

[74] Alessandra Russo. *140 Logic exercises 4*. 2018.

[75] Alessandra Russo and Ian Hodkinson. *Lecture notes: C140 Logic*. 2018.

[76] K. Rustan and M. Leino. “Developing Verified Programs with Dafny”. In: *Verified Software: Theories, Tools, Experiments*. Ed. by Rajeev Joshi, Peter Müller, and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 82–82. ISBN: 978-3-642-27705-4.

[77] Peter Simons. “Jan Łukasiewicz”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2017. Metaphysics Research Lab, Stanford University, 2017.

[78] *SMT-COMP 2018 Results*. URL: http://smtcomp.sourceforge.net/2018/results-toc.shtml.

[79] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. “Zeno: An Automated Prover for Properties of Recursive Data Structures”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Cormac Flanagan and Barbara König. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 407–421. ISBN: 978-3-642-28756-5.

[80] Ed. T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. 2017. URL: https://tools.ietf.org/html/rfc8259.

[81] Robert W. Floyd. “Assigning Meanings to Programs”. In: *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics* 19 (1967). DOI: 10.1090/psapm/019/0235771.

[82] Makarius Wenzel. “Isabelle/jEdit – A Prover IDE within the PIDE Framework”. In: *Intelligent Computer Mathematics*. Ed. by Johan Jeuring et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 468–471. ISBN: 978-3-642-31374-5.

[83] Frank Zenker et al. “Designing an Introductory Course to Elementary Symbolic Logic within the Blackboard E-learning Environment”. In: *Tools for Teaching Logic*. Ed. by Patrick Blackburn et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 249–255. ISBN: 978-3-642-21350-2.

Appendix A

Translation into SMT-LIB Script

Notice that when names of functions, terms, types, propositions, and predicates are translated into SMT-LIB script, we add prefixes `func-`, `term-`, `type-`, `prop-`, and `pred-` to them, to avoid name clashing.

Definition A.1. Translation of typing statements into SMT-LIB script

$$
\begin{aligned}
[\![\tau :: \texttt{Type}]\!]_{SMT} &\triangleq \texttt{(declare-sort type-}\tau\texttt{ 0)} \\
[\![t : \tau]\!]_{SMT} &\triangleq \texttt{(declare-const term-}t\texttt{ type-}\tau\texttt{)} \\
[\![f : \tau_1 \cdots \tau_n \to \sigma]\!]_{SMT} &\triangleq \texttt{(declare-fun func-}f\texttt{ (type-}\tau_1\ \cdots\ \texttt{type-}\tau_n\texttt{) type-}\sigma\texttt{)} \\
[\![P : \tau_1 \cdots \tau_n \to \texttt{Prop}]\!]_{SMT} &\triangleq \texttt{(declare-fun func-}f\texttt{ (type-}\tau_1\ \cdots\ \texttt{type-}\tau_n\texttt{) Bool)} \\
[\![\delta :: C]\!]_{SMT} &\triangleq \texttt{(declare-datatype type-}\delta\texttt{ (}[\![cons_1]\!]_c\ \cdots\ [\![cons_n]\!]_c\texttt{))}
\end{aligned}
$$

where $C = \{cons_1, \cdots, cons_n\}$, and

$$
\begin{aligned}
[\![a]\!]_c &\triangleq \texttt{(term-}a\texttt{)} \\
[\![c : \tau_1 \cdots \tau_n]\!]_c &\triangleq \texttt{(func-}c\texttt{ (}c\texttt{-}0\texttt{ type-}\tau_1\texttt{)}\ \cdots\ \texttt{(}c\texttt{-}(n-1)\texttt{ type-}\tau_n\texttt{))}
\end{aligned}
$$

Definition A.2. Translation of terms into SMT-LIB script

$$
\begin{aligned}
[\![c]\!]_{SMT} &\triangleq \texttt{term-}c && c \text{ is a constant} \\
[\![v]\!]_{SMT} &\triangleq \texttt{term-}v && v \text{ is a variable} \\
[\![f(t_1, \cdots, t_n)]\!]_{SMT} &\triangleq \texttt{(func-}f\ [\![t_1]\!]_{SMT}\ \cdots\ [\![t_n]\!]_{SMT}\texttt{)} \\
[\![\texttt{if}(\phi_C, t_t, t_f)]\!]_{SMT} &\triangleq \texttt{(ite }[\![\phi_C]\!]_{SMT}\ [\![t_t]\!]_{SMT}\ [\![t_f]\!]_{SMT}\texttt{)}
\end{aligned}
$$

Definition A.3. Translation of formulae into SMT-LIB script

$$
\begin{aligned}
[\![P]\!]_{SMT} &\triangleq \texttt{prop-}P && P \text{ is a proposition} \\
[\![\neg\phi]\!]_{SMT} &\triangleq \texttt{(not }[\![\phi]\!]_{SMT}\texttt{)} \\
[\![\phi_1 \wedge \phi_2]\!]_{SMT} &\triangleq \texttt{(and }[\![\phi_1]\!]_{SMT}\ [\![\phi_2]\!]_{SMT}\texttt{)} \\
[\![\phi_1 \vee \phi_2]\!]_{SMT} &\triangleq \texttt{(or }[\![\phi_1]\!]_{SMT}\ [\![\phi_2]\!]_{SMT}\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
[\![\phi_1 \rightarrow \phi_2]\!]_{SMT} &\triangleq \texttt{(=> } [\![\phi_1]\!]_{SMT}\ \ [\![\phi_2]\!]_{SMT}\texttt{)} \\
[\![\phi_1 \leftrightarrow \phi_2]\!]_{SMT} &\triangleq \texttt{(= } [\![\phi_1]\!]_{SMT}\ \ [\![\phi_2]\!]_{SMT}\texttt{)} \\
[\![\forall t : \tau\ \phi]\!]_{SMT} &\triangleq \texttt{(forall ((term-}t\texttt{ type-}\tau\texttt{)) } [\![\phi]\!]_{SMT}\texttt{)} \\
[\![\exists t : \tau\ \phi]\!]_{SMT} &\triangleq \texttt{(exists ((term-}t\texttt{ type-}\tau\texttt{)) } [\![\phi]\!]_{SMT}\texttt{)} \\
[\![P(t_1, \cdots, t_n)]\!]_{SMT} &\triangleq \texttt{(pred-}P\ \ [\![t_1]\!]_{SMT}\ \ \cdots\ \ [\![t_n]\!]_{SMT}\texttt{)} \\
[\![t_1 = t_2]\!]_{SMT} &\triangleq \texttt{(= } [\![t_1]\!]_{SMT}\ \ [\![t_2]\!]_{SMT}\texttt{)}
\end{aligned}
$$

Appendix B

Theorems

B.1 Logic

Theorem B.1. $P \to Q \vdash \neg Q \to \neg P$

Proof. Proven in `example-propositional-3.prf`. □

Theorem B.2. $\neg(\neg P \land \neg Q) \vdash P \lor Q$

Proof. Proven in `example-propositional-4.prf`. □

Theorem B.3. $\vdash (P \to Q) \lor (Q \to P)$

Proof. Proven in `example-propositional-5.prf`. □

Theorem B.4. $\forall x.\exists y.[P(y) \land x \neq y]$ is equivalent to $\exists x.\exists y.[P(x) \land P(y) \land x \neq y]$

Proof. Proven in `example-first-order-4.prf`. □

Theorem B.5. $\forall u.g(f(u)) = h(f(u)), \forall z.\exists v.f(v) = z \vdash \forall v.g(v) = h(v)$

Proof. Proven in `example-first-order-5.prf`. □

Theorem B.6. $\forall x.\forall y.[f(g(x)) = f(g(y)) \to x = y] \vdash \forall u.\forall v.[g(u) = g(v) \to u = v]$

Proof. Proven in `example-first-order-6.prf`. □

Theorem B.7. $\vdash \forall x.\forall y.[x = y \to f(x) = f(y)]$

Proof. Proven in `example-first-order-7.prf`. □

Theorem B.8. $\forall x.[x = a \lor x = b], g(a) = b, \forall x.\forall y.[g(x) = g(y) \to x = y] \vdash g(g(a)) = a$

Proof. Proven in `example-first-order-8.prf`. □

B.2 Reasoning about Programs

In this chapter we list the full set of theorems we have extracted from the reasoning about programs course material.

For the theorems we have proven, we give the name of the proof file, which is included in `examples` folder in the code repository. For the others, we explain the reason why our system is incapable of proving them.

Theorems from 'Structural induction over Haskell data types' [28]

Theorem B.9. *subList* removes elements from a list

$$\forall xs : \texttt{List}.\forall ys : \texttt{List}.\forall z : \texttt{Int}.[elem(z, ys) = t \rightarrow elem(z, subList(xs, ys)) = f]$$

where

$$\begin{aligned} elem(x, nil) &= f \\ elem(x, cons(y, ys)) &= \begin{cases} t & \text{if } x = y \\ elem(x, ys) & \text{otherwise} \end{cases} \\ subList(nil, ys) &= nil \\ subList(cons(x, xs), ys) &= \begin{cases} subList(xs, ys) & \text{if } elem(x, ys) = t \\ cons(x, subList(xs, ys)) & \text{otherwise} \end{cases} \end{aligned}$$

Proof. Proven in `theorem-9.prf`. □

Theorem B.10. Append then reverse is equal to reverse then append

$$\forall xs : \texttt{List}.\forall ys : \texttt{List}.reverse(append(xs, ys)) = append(reverse(ys), reverse(xs))$$

where

$$\begin{aligned} reverse(nil) &= nil \\ reverse(cons(x, xs)) &= append(reverse(xs), cons(x, nil)) \end{aligned}$$

Proof. Proven in `theorem-10.prf`. A detailed walkthrough of the proof is available in section E.4.1. □

Theorem B.11. *sum* and *sumTr* are equal

$$\forall is : \texttt{List}.sum(is) = sumTr(is, 0)$$

where

$$\begin{aligned} sum(nil) &= 0 \\ sum(cons(i, is)) &= i + sum(is) \\ sumTr(nil, k) &= k \\ sumTr(cons(i, is), k) &= sumTr(is, i + k) \end{aligned}$$

Proof. Proven in `theorem-11.prf`. A detailed walkthrough of the proof is available in section E.4.1. □

Theorems from 'Week 4 Tutorial - Structural Induction' [32]

Theorem B.12. Addition of natural numbers is commutative

$$\forall m : \texttt{Nat}.\forall n : \texttt{Nat}.add(m, n) = add(n, m)$$

where

$$\begin{aligned} add(z, j) &= j \\ sum(s(i), j) &= s(add(i, j)) \end{aligned}$$

Proof. Proven in `theorem-12.prf`. □

Theorem B.13. Two evaluation strategies are the same

$$\begin{aligned}&\forall tm : \texttt{Term}.ValPositive(tm)\\&\quad\rightarrow\\&\forall tm : \texttt{Term}.eval(tm) = sign(eval(rip(tm)), pos(tm))\end{aligned}$$

where

$$\begin{aligned}
\texttt{Term} &:: \{val : \texttt{Int}, min : \texttt{Term}, mul : \texttt{Term Term}\}\\
ValPositive(val(i)) &\leftrightarrow i \geq 0\\
ValPositive(min(tm)) &\leftrightarrow ValPositive(tm)\\
ValPositive(mul(t1, t2)) &\leftrightarrow ValPositive(t1) \wedge ValPositive(t2)\\
eval(val(i)) &= i\\
eval(min(tm)) &= -eval(tm)\\
eval(mul(t1, t2)) &= eval(t1) * eval(t2)\\
rip(val(i)) &= val(i)\\
rip(min(tm)) &= rip(tm)\\
rip(mul(t1, t2)) &= mul(rip(t1), rip(t2))\\
pos(val(i)) &= \begin{cases} t & \text{if } x \geq 0\\ f & \text{otherwise}\end{cases}\\
pos(min(tm)) &= not(pos(tm))\\
pos(mul(t1, t2)) &= iff(pos(t1), pos(t2))\\
sign(i, b) &= \begin{cases} i & \text{if } b = t\\ -i & \text{if } b = f\end{cases}
\end{aligned}$$

Proof. Proven in `theorem-13.prf`. □

Theorems from 'Week 4 PMT - Structural Induction' [31]

Theorem B.14. 'RevConcat'

$$\forall xs : \texttt{List}.\forall x : \texttt{Int}.reverse(append(xs, cons(x, nil))) = cons(x, reverse(xs))$$

where *reverse* is as defined in theorem B.10.

Proof. Proven in `theorem-14.prf`. □

Theorem B.15. 'RevRev'

$$\forall xs : \texttt{List}.reverse(reverse(xs)) = xs$$

where *reverse* is as defined in theorem B.10.

Proof. Proven in `theorem-15.prf`. □

Theorem B.16. *reverseA* can do what *reverse* does

$$\forall xs : \texttt{List}.reverseA(xs, nil) = reverse(xs)$$

where *reverse* is as defined in theorem B.10, and

$$\begin{aligned} reverseA(nil, ys) &= ys \\ reverseA(cons(x, xs), ys) &= reverseA(xs, cons(x, ys)) \end{aligned}$$

Proof. Proven in `theorem-16.prf`. □

Theorem B.17. $g4$ checks whether or not a list is a palindrome

$$\forall xs : \texttt{List}.g4(xs) = t \leftrightarrow Pal(xs)$$

where *reverse* is as defined in theorem B.10, and

$$\begin{aligned} \forall xs : \texttt{List}.Pal(xs) \leftrightarrow \exists ys : \texttt{List}.xs &= append(ys, reverse(ys)) \\ &\vee \exists y : \texttt{Int}.xs = append(ys, append(cons(y, nil), reverse(ys))) \\ g4(xs) &= g5(xs, nil) \\ g5(nil, ys) &= f \\ g5(cons(x, xs), ys) &= \begin{cases} t & \text{if } cons(x, xs) = ys \\ t & \text{if } xs = ys \\ g5(xs, cons(x, ys)) & \text{otherwise} \end{cases} \end{aligned}$$

Proof. Proven in `theorem-17.prf`. □

Theorems from 'Week 4 Assessed PMT - Structural Induction' [30]

Theorem B.18. *size* and *length* does the same

$$\forall t : \texttt{Tree}.size(t) = length(enc(t))$$

where a `List` is a list of `Code`s, and

$$\begin{aligned} \texttt{Tree} &:: leaf : \texttt{Int}, node : \texttt{Tree Tree} \\ \texttt{Code} &:: nd, lf : \texttt{Int} \\ enc(leaf(x)) &= cons(leaf(x), nil) \\ enc(node(t1, t2)) &= append(cons(nd, enc(t1)), enc(t2)) \\ size(leaf(x)) &= 1 \\ size(node(t1, t2)) &= 1 + size(t1) + size(t2) \end{aligned}$$

Proof. Proven in `theorem-18.prf`. □

Theorem B.19. *dec* is reverse of *enc*

$$\forall t : \texttt{Tree}.t = dec(enc(t))$$

where `Tree`, `Code`, and *enc* are as defined in the previous theorem, and

$$\texttt{Pair} :: p : \texttt{Tree List}$$

$$fst(p(x,y)) = x \qquad snd(p(x,y)) = y$$

$$dec(t) = \begin{cases} fst(decAux(t)) & \text{if } snd(decAux(t)) = nil \\ leaf(0) & \text{otherwise} \end{cases}$$

$$decAux(cons(lf(i),cds)) = p(leaf(i),cds)$$

$$decAux(cons(nd,cds)) = p(node(fst(decAux(cds)), fst(decAux(snd(decAux(cds))))), snd(decAux(snd(dexAux(cds)))))$$

Proof. Proven in `theorem-19.prf`. □

Theorem B.20. *decd* is reverse of *encd*

$$\forall t : \texttt{Tree}.t = decd(encd(t))$$

where `Tree`, `Code`, are as defined in the previous theorems, and

$$\texttt{ListC} :: nilC, consC : \texttt{Code ListC}$$

$$\texttt{ListT} :: nilT, consT : \texttt{Tree ListT}$$

appendC and *appendT* are defined in the obvious way for `ListC` and `ListT`

$$encd(leaf(i)) = consC(lf(i), nilC)$$

$$encd(node(t1,t2)) = appendC(encd(t1), appendC(encd(t2), cons(nd, nilC)))$$

$$decd(cds) = decdAux(cds, nilT)$$

$$decdAux(nilC, consT(t,ts)) = t$$

$$decdAux(consC(lf(i),cds),ts) = decdAux(cds, cons(leaf(i),ts))$$

$$decdAux(consC(nd,cds), consT(t1, consT(t2,ts))) = decdAux(cds, consT(node(t1,t2),ts))$$

Proof. Proven in `theorem-20.prf`. □

Theorems from 'Inductively Defined Functions' [27]

Theorem B.21. g performs multiplication

$$\forall i : \texttt{Int}.\forall j : \texttt{Int}.[i \geq 0 \wedge j \geq 0 \ \rightarrow \ g(i,j) = i * j]$$

where

$$g(i,j) = g'(i,j,0,0)$$

$$g'(i,j,cnt,acc) = \begin{cases} acc & \text{if } cnt \geq i \\ g'(i,j,1+cnt,j+acc) & \text{otherwise} \end{cases}$$

Proof. Proven in `theorem-21.prf`. A detailed walkthrough of the proof is available in section E.4.2. □

Theorem B.22. *divMod* finds the quotient and remainder

$$\forall m : \texttt{Int}.\forall n : \texttt{Int}.\forall k1 : \texttt{Int}.\forall k2 : \texttt{Int}.[p(k1,k2) = divMod(m,n) \quad \rightarrow \quad m = k1 * n + k2 \wedge k2 < n]$$

where

$$\begin{aligned} \texttt{Pair} :: p : \texttt{Tree List} \\ fst(p(x,y)) = x \qquad snd(p(x,y)) = y \\ divMod(m,n) = dm(m,n,0,0) \\ dm(m,n,cnt,acc) = \begin{cases} p(cnt, m - acc) & \text{if } acc + n > m \\ dm(m,n,cnt+1,acc+n) & \text{otherwise} \end{cases} \end{aligned}$$

Proof. Proven in `theorem-22.prf`. □

Theorem B.23. m is a power function

$$\forall x : \texttt{Int}.[x \geq 0 \rightarrow m(x) = 2^x]$$

where

$$\begin{aligned} m(x) = m'(x,0,1) \\ m'(i,cnt,acc) = \begin{cases} acc & \text{if } cnt \leq i \\ m'(i,cnt+1,2*acc) & \text{otherwise} \end{cases} \end{aligned}$$

We did not produce a proof as SMT-LIB does not offer the power operator. Any proof would thus require us to define out own 'trusted' power operation, which would defeat the purpose of proving the correctness of a power function.

Theorems from 'Week 5 Tutorial - Induction over Recursively Defined Relations' [35]

Theorem B.24. f and g are equivalent

$$\forall n : \texttt{Int}.[n \geq 0 \rightarrow f(n) = g(n)]$$

where

$$\begin{aligned} f(n) = \begin{cases} 10 + 10 * n & \text{if } 0 \leq n \wedge n < 3 \\ f(n-1) * f(n-3) & \text{otherwise} \end{cases} \\ g(n) = \begin{cases} 10 + 10 * n & \text{if } 0 \leq n \wedge n < 3 \\ h(n,2,30,20,10) & \text{otherwise} \end{cases} \\ h(n,cnt,k1,k2,k3) = \begin{cases} k1 & \text{if } n = cnt \\ h(n,cnt+1,k1*k3,k1,k2) & \text{otherwise} \end{cases} \end{aligned}$$

We cannot produce a proof because the proof would require induction the definitions of both f and h, which is not supported by Emmy.

Theorems from 'Week 5 PMT - Induction over Recursively Defined Sets, Relations and Functions' [34]

Theorem B.25. *min* is commutative

$$\forall i : \texttt{Nat}.\forall j : \texttt{Nat}.\forall k : \texttt{Nat}.[k = min(i,j) \;\; \rightarrow \;\; k = min(j,i)]$$

where

$$\forall n : \texttt{Nat}.min(n,z) = z$$
$$\forall n : \texttt{Nat}.min(z,n) = z$$
$$\forall n : \texttt{Nat}.\forall n' : \texttt{Nat}.\forall k : \texttt{Nat}.[k = min(n,n') \;\; \rightarrow \;\; s(k) = min(s(n),s(n'))]$$

The proof requires an induction over k, which is not easy to perform as k is quantified inside two other quantifications. We therefore prove

$$\forall k : \texttt{Nat}.\forall i : \texttt{Nat}.\forall j : \texttt{Nat}.[k = min(i,j) \;\; \rightarrow \;\; k = min(j,i)]$$

instead.

Proof. Proven in `theorem-25.prf`. □

The way in which *min* is defined makes the proof somehow complicated. We included an alternative, simpler proof in `theorem-25.prf`, written using function definitions.

Theorem B.26. Induction principle of odd numbers implies induction principle of natural numbers

$$\begin{aligned}
&\Big[P(s(z)) \;\; \wedge \;\; \forall m : \texttt{Nat}.[Odd(m) \wedge P(m) \;\; \rightarrow \;\; P(s(s(m)))] \\
&\qquad \rightarrow \forall n : \texttt{Nat}.[Odd(n) \rightarrow P(n)]\Big] \\
&\quad \rightarrow \\
&\Big[[Odd(z) \rightarrow P(z)] \;\; \wedge \;\; \forall m : \texttt{Nat}.\big[[Odd(m) \rightarrow P(m)] \;\; \rightarrow \;\; [Odd(s(m)) \rightarrow P(s(m))]\big] \\
&\qquad \rightarrow \forall n : \texttt{Nat}.[Odd(n) \rightarrow P(n)]\Big]
\end{aligned}$$

where

$$Odd(s(z))$$
$$\forall n : \texttt{Nat}.[Odd(n) \rightarrow Odd(s(n))]$$

Proof. Proven in `theorem-26.prf`. □

Theorems from 'Week 5 Assessed PMT' [33]

This coursework asks for a proof of theorem B.22, which we have already proven.

Appendix C

Example Programs for Section 9.3.3

An induction principle for theorem 9.4, defined in section 9.3.3, can be written as:

$$\begin{aligned}
&\forall l2 : \texttt{List}.length(append(nil, l2)) = length(nil) + length(l2) \\
&\quad\wedge \\
&\forall l1 : \texttt{List}.\Big[\forall l2 : \texttt{List}.length(append(l1, l2)) = length(l1) + length(l2) \\
&\qquad\quad\rightarrow \\
&\qquad \forall i : \texttt{Int}.\forall l2 : \texttt{List}.length(append(cons(i, l1), l2)) = length(cons(i, l1)) + length(l2)\Big]
\end{aligned} \tag{C.1}$$

We denote C.1 by ϕ.

We produced some semantically equivalent variations of ϕ, with minor syntactic differenced with ϕ. We denote the ith variation as ϕ_i'. We then used Emmy to check whether or not $\Gamma, \Delta, \phi \rightarrow \phi_i'$ holds, using Z3 as the underlying SMT solver.

The summary of results from all variations is as below:

	Rename?	Swap?	Definition?	Proven?
1				✓*
2			+	✓
3		+		✓
4		+	+	✓
5	+			✓*
6	+		+	✓
7	+	+		
8	+	+	+	✓

Table C.1: Summary of results.

Where

Rename means that we renamed the quantified variables in ϕ', like from $\forall x : \tau.\phi$ to $\forall y : \tau.[y/x]\phi$.

Swap means that we swapped the induction cases in ϕ'. like from $A \wedge B$ to $B \wedge A$.

Definition means that we included function definitions in the justifications when proving.

Proven means that the entailment is proven. '✓*' means that EMMY did not call the SMT solver because the two formulae are equivalent according to definition 4.7.

In a nutshell, the entailment is only *not* proved when we renamed quantified variables, swapped induction cases, but did not provide function definitions as justifications.

1. ϕ is exactly the same as ϕ_1'. SMT solver is not called.

2.
```
(set-option :timeout 2000)
(set-logic AUFNIA)
(declare-sort type-Atom 0)
(declare-datatype
 type-List
 ((term-nil) (func-cons (cons-0 Int) (cons-1 type-List))))
(declare-fun func-append (type-List type-List) type-List)
(declare-fun func-length (type-List) Int)
(assert
 (and (forall
       ((term-l2 type-List))
       (=
        (func-length (func-append term-nil term-l2))
        (+ (func-length term-nil) (func-length term-l2))))
      (forall
       ((term-l1 type-List))
       (=>
        (forall
         ((term-l2 type-List))
         (=
          (func-length (func-append term-l1 term-l2))
          (+ (func-length term-l1) (func-length term-l2))))
        (forall
         ((term-i Int))
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append (func-cons term-i term-l1) term-l2))
           (+
            (func-length (func-cons term-i term-l1))
            (func-length term-l2))))))))))
(assert
 (forall
  ((term-l+0 type-List))
  (= (func-append term-nil term-l+0) term-l+0)))
(assert
 (forall
  ((term-l2+0 type-List))
  (forall
   ((term-l1+0 type-List))
   (forall
    ((term-i+0 Int))
    (=
     (func-append (func-cons term-i+0 term-l1+0) term-l2+0)
     (func-cons term-i+0 (func-append term-l1+0 term-l2+0)))))))
(assert (= (func-length term-nil) 0))
(assert
 (forall
  ((term-l+1 type-List))
  (forall
   ((term-i+1 Int))
   (=
```

```
    (func-length (func-cons term-i+1 term-l+1))
    (+ 1 (func-length term-l+1))))))
(assert
 (not
  (and (forall
        ((term-l2 type-List))
        (=
         (func-length (func-append term-nil term-l2))
         (+ (func-length term-nil) (func-length term-l2))))
       (forall
        ((term-l1 type-List))
        (=>
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append term-l1 term-l2))
           (+ (func-length term-l1) (func-length term-l2))))
         (forall
          ((term-i Int))
          (forall
           ((term-l2 type-List))
           (=
            (func-length (func-append (func-cons term-i term-l1) term-l2))
            (+
             (func-length (func-cons term-i term-l1))
             (func-length term-l2))))))))))
(check-sat)
```

Output: (unsat)

3.
```
(set-option :timeout 2000)
(set-logic AUFNIA)
(declare-sort type-Atom 0)
(declare-datatype
 type-List
 ((term-nil) (func-cons (cons-0 Int) (cons-1 type-List))))
(declare-fun func-append (type-List type-List) type-List)
(declare-fun func-length (type-List) Int)
(assert
 (and (forall
       ((term-l2 type-List))
       (=
        (func-length (func-append term-nil term-l2))
        (+ (func-length term-nil) (func-length term-l2))))
      (forall
       ((term-l1 type-List))
       (=>
        (forall
         ((term-l2 type-List))
         (=
          (func-length (func-append term-l1 term-l2))
          (+ (func-length term-l1) (func-length term-l2))))
        (forall
         ((term-i Int))
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append (func-cons term-i term-l1) term-l2))
           (+
            (func-length (func-cons term-i term-l1))
            (func-length term-l2)))))))))
```

```
(assert
 (not
  (and (forall
        ((term-l1 type-List))
        (=>
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append term-l1 term-l2))
           (+ (func-length term-l1) (func-length term-l2))))
         (forall
          ((term-i Int))
          (forall
           ((term-l2 type-List))
           (=
            (func-length (func-append (func-cons term-i term-l1) term-l2))
            (+
             (func-length (func-cons term-i term-l1))
             (func-length term-l2)))))))
       (forall
        ((term-l2 type-List))
        (=
         (func-length (func-append term-nil term-l2))
         (+ (func-length term-nil) (func-length term-l2)))))))
(check-sat)
```

Output: (unsat)

4.
```
(set-option :timeout 2000)
(set-logic AUFNIA)
(declare-sort type-Atom 0)
(declare-datatype
 type-List
 ((term-nil) (func-cons (cons-0 Int) (cons-1 type-List))))
(declare-fun func-append (type-List type-List) type-List)
(declare-fun func-length (type-List) Int)
(assert
 (and (forall
       ((term-l2 type-List))
       (=
        (func-length (func-append term-nil term-l2))
        (+ (func-length term-nil) (func-length term-l2))))
      (forall
       ((term-l1 type-List))
       (=>
        (forall
         ((term-l2 type-List))
         (=
          (func-length (func-append term-l1 term-l2))
          (+ (func-length term-l1) (func-length term-l2))))
        (forall
         ((term-i Int))
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append (func-cons term-i term-l1) term-l2))
           (+
            (func-length (func-cons term-i term-l1))
            (func-length term-l2)))))))))
(assert
 (forall
```

```
  ((term-l+0 type-List))
  (= (func-append term-nil term-l+0) term-l+0)))
(assert
 (forall
  ((term-l2+0 type-List))
  (forall
   ((term-l1+0 type-List))
   (forall
    ((term-i+0 Int))
    (=
     (func-append (func-cons term-i+0 term-l1+0) term-l2+0)
     (func-cons term-i+0 (func-append term-l1+0 term-l2+0)))))))
(assert (= (func-length term-nil) 0))
(assert
 (forall
  ((term-l+1 type-List))
  (forall
   ((term-i+1 Int))
   (=
    (func-length (func-cons term-i+1 term-l+1))
    (+ 1 (func-length term-l+1))))))
(assert
 (not
  (and (forall
        ((term-l1 type-List))
        (=>
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append term-l1 term-l2))
           (+ (func-length term-l1) (func-length term-l2))))
         (forall
          ((term-i Int))
          (forall
           ((term-l2 type-List))
           (=
            (func-length (func-append (func-cons term-i term-l1) term-l2))
            (+
             (func-length (func-cons term-i term-l1))
             (func-length term-l2)))))))
       (forall
        ((term-l2 type-List))
        (=
         (func-length (func-append term-nil term-l2))
         (+ (func-length term-nil) (func-length term-l2)))))))
(check-sat)
```

Output: (unsat)

5. ϕ is exactly the same as ϕ'_5. SMT solver is not called.

6.
```
(set-option :timeout 2000)
(set-logic AUFNIA)
(declare-sort type-Atom 0)
(declare-datatype
 type-List
 ((term-nil) (func-cons (cons-0 Int) (cons-1 type-List))))
(declare-fun func-append (type-List type-List) type-List)
(declare-fun func-length (type-List) Int)
(assert
 (and (forall
```

```
      ((term-l2 type-List))
      (=
       (func-length (func-append term-nil term-l2))
       (+ (func-length term-nil) (func-length term-l2))))
     (forall
      ((term-l1 type-List))
      (=>
       (forall
        ((term-l2 type-List))
        (=
         (func-length (func-append term-l1 term-l2))
         (+ (func-length term-l1) (func-length term-l2))))
       (forall
        ((term-i Int))
        (forall
         ((term-l2 type-List))
         (=
          (func-length (func-append (func-cons term-i term-l1) term-l2))
          (+
           (func-length (func-cons term-i term-l1))
           (func-length term-l2)))))))))
(assert
 (forall
  ((term-l+0 type-List))
  (= (func-append term-nil term-l+0) term-l+0)))
(assert
 (forall
  ((term-l2+0 type-List))
  (forall
   ((term-l1+0 type-List))
   (forall
    ((term-i+0 Int))
    (=
     (func-append (func-cons term-i+0 term-l1+0) term-l2+0)
     (func-cons term-i+0 (func-append term-l1+0 term-l2+0)))))))
(assert (= (func-length term-nil) 0))
(assert
 (forall
  ((term-l+1 type-List))
  (forall
   ((term-i+1 Int))
   (=
    (func-length (func-cons term-i+1 term-l+1))
    (+ 1 (func-length term-l+1))))))
(assert
 (not
  (and (forall
        ((term-l2 type-List))
        (=
         (func-length (func-append term-nil term-l2))
         (+ (func-length term-nil) (func-length term-l2))))
       (forall
        ((term-l- type-List))
        (=>
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append term-l- term-l2))
           (+ (func-length term-l-) (func-length term-l2))))
         (forall
```

```
          ((term-i- Int))
          (forall
           ((term-l2 type-List))
           (=
            (func-length (func-append (func-cons term-i- term-l-) term-l2))
            (+
             (func-length (func-cons term-i- term-l-))
             (func-length term-l2))))))))))
(check-sat)
```

Output: (unsat)

7.
```
(set-option :timeout 2000)
(set-logic AUFNIA)
(declare-sort type-Atom 0)
(declare-datatype
 type-List
 ((term-nil) (func-cons (cons-0 Int) (cons-1 type-List))))
(declare-fun func-append (type-List type-List) type-List)
(declare-fun func-length (type-List) Int)
(assert
 (and (forall
       ((term-l2 type-List))
       (=
        (func-length (func-append term-nil term-l2))
        (+ (func-length term-nil) (func-length term-l2))))
      (forall
       ((term-l1 type-List))
       (=>
        (forall
         ((term-l2 type-List))
         (=
          (func-length (func-append term-l1 term-l2))
          (+ (func-length term-l1) (func-length term-l2))))
        (forall
         ((term-i Int))
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append (func-cons term-i term-l1) term-l2))
           (+
            (func-length (func-cons term-i term-l1))
            (func-length term-l2)))))))))
(assert
 (not
  (and (forall
        ((term-l- type-List))
        (=>
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append term-l- term-l2))
           (+ (func-length term-l-) (func-length term-l2))))
         (forall
          ((term-i- Int))
          (forall
           ((term-l2 type-List))
           (=
            (func-length (func-append (func-cons term-i- term-l-) term-l2))
            (+
             (func-length (func-cons term-i- term-l-))
```

```
             (func-length term-l2)))))))
       (forall
        ((term-l2 type-List))
        (=
         (func-length (func-append term-nil term-l2))
         (+ (func-length term-nil) (func-length term-l2)))))))
(check-sat)
```

Output: (unknown)

8.
```
(set-option :timeout 2000)
(set-logic AUFNIA)
(declare-sort type-Atom 0)
(declare-datatype
 type-List
 ((term-nil) (func-cons (cons-0 Int) (cons-1 type-List))))
(declare-fun func-append (type-List type-List) type-List)
(declare-fun func-length (type-List) Int)
(assert
 (and (forall
       ((term-l2 type-List))
       (=
        (func-length (func-append term-nil term-l2))
        (+ (func-length term-nil) (func-length term-l2))))
      (forall
       ((term-l1 type-List))
       (=>
        (forall
         ((term-l2 type-List))
         (=
          (func-length (func-append term-l1 term-l2))
          (+ (func-length term-l1) (func-length term-l2))))
        (forall
         ((term-i Int))
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append (func-cons term-i term-l1) term-l2))
           (+
            (func-length (func-cons term-i term-l1))
            (func-length term-l2)))))))))
(assert
 (forall
  ((term-l+0 type-List))
  (= (func-append term-nil term-l+0) term-l+0)))
(assert
 (forall
  ((term-l2+0 type-List))
  (forall
   ((term-l1+0 type-List))
   (forall
    ((term-i+0 Int))
    (=
     (func-append (func-cons term-i+0 term-l1+0) term-l2+0)
     (func-cons term-i+0 (func-append term-l1+0 term-l2+0)))))))
(assert (= (func-length term-nil) 0))
(assert
 (forall
  ((term-l+1 type-List))
  (forall
   ((term-i+1 Int))
```

```
   (=
    (func-length (func-cons term-i+1 term-l+1))
    (+ 1 (func-length term-l+1))))))
(assert
 (not
  (and (forall
       ((term-l- type-List))
       (=>
        (forall
         ((term-l2 type-List))
         (=
          (func-length (func-append term-l- term-l2))
          (+ (func-length term-l-) (func-length term-l2))))
        (forall
         ((term-i- Int))
         (forall
          ((term-l2 type-List))
          (=
           (func-length (func-append (func-cons term-i- term-l-) term-l2))
           (+
            (func-length (func-cons term-i- term-l-))
            (func-length term-l2)))))))
      (forall
       ((term-l2 type-List))
       (=
        (func-length (func-append term-nil term-l2))
        (+ (func-length term-nil) (func-length term-l2)))))))
(check-sat)
```

Output: `(unsat)`

Notice that we raised the time limit to 2000ms when checking these entailments.

Appendix D

Program Language

Here we specify the concrete syntax of the language described in chapter 5, using (extended) BNF notation, where '+' means the repeating a rule for one or more times, and '`*`' means the repetition a rule for zero or more times.

The syntax of the program language is based on LISP's s-expressions [62]. Due to the heavy use of brackets, we omit all quotation marks around literal. We also use LISP symbols to denote logic connectives, mathematical operators, and step names. Such terms are displayed using a `monospaced typeface` without any quotation marks. Terms (enclosed by brackets) must have whitespaces between them.

Comment

⟨*Comment*⟩ ::= A markdown text enclosed by double quotation marks

Terms Notice that in the rule for Conditional terms we reference the rule for Formula.

⟨*TermIdentifier*⟩ ::= Strings that starts with a lower-case letter, and contains only letters, digits, and hyphens

⟨*FunctionIdentifier*⟩ ::= ⟨*TermIdentifier*⟩ | `+` | `-` | `*` | `/`

⟨*Digit*⟩ ::= '`0`' to '`9`'

⟨*Sign*⟩ ::= '+' | '−' | ⟨*Empty*⟩

⟨*Integer*⟩ ::= ⟨*Sign*⟩⟨*Digit*⟩+

⟨*Conditional*⟩ ::= (`if` ⟨*Formula*⟩ ⟨*Term*⟩ ⟨*Term*⟩)

⟨*FunctionApplication*⟩ ::= (⟨*FunctionIdentifier*⟩ ⟨*Term*⟩+)

⟨*Term*⟩ ::= ⟨*TermIdentifier*⟩ | ⟨*Integer*⟩ | ⟨*Conditional*⟩ | ⟨*FunctionApplication*⟩

Formulae

⟨*PropositionIdentifier*⟩ ::= Strings that starts with an upper-case letter, and contains only letters, digits, and hyphens

⟨*TypeIdentifier*⟩ ::= ⟨*PropositionIdentifier*⟩

⟨*PredicateIdentifier*⟩ ::= ⟨*PropositionIdentifier*⟩ | `=` | `<` | `>`

⟨*PredicateApplication*⟩ ::= (⟨*PredicateIdentifier*⟩ ⟨*Term*⟩+)

⟨*Negation*⟩ ::= (`Not` ⟨*Formula*⟩)

⟨*BinaryConnective*⟩ ::= `And` | `Or` | `Implies` | `If-and-only-if`

⟨*BinaryFormula*⟩ ::= (⟨*BinaryConnective*⟩ ⟨*Formula*⟩ ⟨*Formula*⟩)

⟨*Quantifier*⟩ ::= `Forall` | `Exists`

⟨*Quantification*⟩ ::= (⟨*Quantifier*⟩ ⟨*TypeIdentifier*⟩ ⟨*Term*⟩ ⟨*Formula*⟩)

⟨*InductionMarker*⟩ ::= (`Ind:` ⟨*StepNumber*⟩ ⟨*Formula*⟩) | (`Ind:` ⟨*Formula*⟩)

⟨*Formula*⟩ ::= ⟨*PropositionIdentifier*⟩ | ⟨*PredicateApplication*⟩ | ⟨*Negation*⟩
| ⟨*BinaryFormula*⟩ | ⟨*Quantification*⟩ | ⟨*InductionMarker*⟩

Steps

⟨*StepNumber*⟩ ::= A string of letters, numbers, and hyphens

⟨*Rule*⟩ ::= A string that ends with '`:`'

⟨*Justification*⟩ ::= `Given` | `?`
| (⟨*Rule*⟩ ⟨*StepNumber*⟩*) | (⟨*StepNumber*⟩*)

⟨*SimpleStep*⟩ ::= [⟨*StepNumber*⟩ ⟨*Formula*⟩ ⟨*Justification*⟩]

⟨*AssumptionStep*⟩ ::= [⟨*StepNumber*⟩ `Assume` ⟨*Formula*⟩ ⟨*Step*⟩+]

⟨*IntroductionStep*⟩ ::= [⟨*StepNumber*⟩ `Take` ⟨*TypeIdentifier*⟩ ⟨*TermIdentifier*⟩ ⟨*Step*⟩+]

⟨*IntroductionWithAssumptionStep*⟩ ::= [⟨*StepNumber*⟩ `Take` ⟨*TypeIdentifier*⟩ ⟨*TermIdentifier*⟩ `such that` ⟨*Formula*⟩ ⟨*Step*⟩+]

⟨*InductionHypothesis*⟩ ::= `Take` ⟨*TypeIdentifier*⟩ ⟨*TermIdentifier*⟩ |
| `Take` ⟨*TypeIdentifier*⟩ ⟨*TermIdentifier*⟩ `such that` ⟨*Formula*⟩

⟨*InductionCaseBody*⟩ ::= ⟨*Step*⟩* | `By` ⟨*Justification*⟩

⟨*InductionCase*⟩ ::= [⟨*InductionHypothesis*⟩* `Show` ⟨*Formula*⟩ ⟨*InductionCaseBody*⟩]

⟨*InductionStep*⟩ ::= [⟨*StepNumber*⟩ `Induction` ⟨*Formula*⟩ ⟨*InductionCase*⟩*]
| [⟨*StepNumber*⟩ `Induction on` ⟨*StepNumber*⟩ ⟨*Formula*⟩ ⟨*InductionCase*⟩*]

⟨*Equality*⟩ ::= `=` ⟨*Term*⟩ ⟨*Justification*⟩

⟨*EqualitiesStep*⟩ ::= [⟨*StepNumber*⟩ `=` ⟨*Term*⟩ ⟨*Equality*⟩+]

⟨*Step*⟩ ::= ⟨*SimpleStep*⟩ | ⟨*AssumptionStep*⟩ | ⟨*IntroductionStep*⟩ | ⟨*IntroductionWithAssumptionStep*⟩ | ⟨*InductionStep*⟩ | ⟨*EqualitiesStep*⟩

Declarations

⟨*TypeDeclaration*⟩ ::= [`Type` ⟨*TypeIdentifier*⟩]

⟨*TermDeclaration*⟩ ::= [`Term` ⟨*TypeIdentifier*⟩ ⟨*TermIdentifier*⟩]

⟨*DataCase*⟩ ::= [⟨*TermIdentifier*⟩ ⟨*TypeIdentifier*⟩*]

⟨*DataStructureDeclaration*⟩ ::= [`Data` ⟨*TypeIdentifier*⟩ ⟨*DataCase*⟩+]

⟨*FunctionDeclaration*⟩ ::= [`Function` ⟨*TermIdentifier*⟩ ⟨*TypeIdentifier*⟩+ `->` ⟨*TypeIdentifier*⟩]

⟨*PredicateDeclaration*⟩ ::= [`Predicate` ⟨*PropositionIdentifier*⟩ ⟨*TypeIdentifier*⟩+]

⟨*Declaration*⟩ ::= ⟨*TypeDeclaration*⟩ | ⟨*TermDeclaration*⟩ | ⟨*DataStructureDeclaration*⟩
 | ⟨*FunctionDeclaration*⟩ ⟨*PredicateDeclaration*⟩

Lemmas

⟨*Lemma*⟩ ::= [⟨*StepNumber*⟩ ⟨*Formula*⟩]

Definitions

⟨*DefinitionCaseCondition*⟩ ::= ⟨*Formula*⟩ | `otherwise`

⟨*DefinitionCase*⟩ ::= [⟨*DefinitionCaseCondition*⟩ ⟨*Term*⟩]

⟨*Definition*⟩ ::= [⟨*StepNumber*⟩ ⟨*FunctionApplication*⟩ ⟨*Term*⟩]
 | [⟨*StepNumber*⟩ ⟨*FunctionApplication*⟩ `cases` ⟨*DefinitionCase*⟩+]

Program

⟨*Section*⟩ ::= { `Declare` ⟨*Declaration*⟩* }
 | { `Lemma` ⟨*Lemma*⟩* }
 | { `Define` ⟨*Definition*⟩* }
 | { `Proof` ⟨*Formula*⟩ ⟨*Step*⟩* }
 | ⟨*Comment*⟩

⟨*Program*⟩ ::= ⟨*Section*⟩*

We use rounded brackets (( )) for things that resemble function applications, such as terms and formulae. We use square brackets ([ ]) for items in a list, such as declarations and steps. We use curly brackets ({ }) for program sections. We use different brackets for mere cosmetic reasons. There is *no semantic difference* between these three types of brackets, and the Racket parser treats them as the same.

Appendix E

Example Proofs

Here are some example proofs that demonstrate the range of theories tha can be proven using Emmy, as well as how Emmy's approach compares to exsting proof methods. This chapter may also be used as a guide to writing proofs in Emmy, especially because we prove a number of examples from the lecture notes.

First, we present a proof carried out in the 'traditional' way, either by 'box' style natural deduction (see 2.1.2), or by a stylised proof (see 2.1.3) We then provide a Emmy program that proves the same thing, written in the concrete syntax of Emmy, which means that they are valid inputs to Emmy and can be automatically checked.

All proofs in this chapter are included in the `examples` directory in the project repository.

E.1 Propositional Logic

For proofs in propositional logic, we can write everything in one proof section. There are no terms not predicates, so nothing needs to be declared or defined. We may use a lemma section to provide existing, proven, lemmas, if we wish.

E.1.1 Peirce's Law

Theorem E.1. Peirce's Law
$\big((P \to Q) \to P\big)$.

Using 'box' style natural deduction, we can prove that Peirce's Law holds, as below:

1.	$(P \to Q) \to P$	Assume
2.	$\neg P$	Assume
3.	P	Assume
4.	$\bot$	$\bot$-Intro 3 2
5.	Q	$\bot$-Elim 4
6.	$P \to Q$	$\to$-Intro 5 3
7.	P	$\to$-Elim 3 1
8.	$\bot$	$\bot$-Intro 2 7
9.	$\neg\neg P$	$\neg$-Intro 2 8
10.	P	$\neg\neg$-Elim 9
11.	$\big((P \to Q) \to P\big) \to P$	$\to$-Intro 1 10

We can replace each assumption with an assumption step, and drop all invocations of natual deduction rules, to obtain the following proof in our language:

```
1  {Proof (Implies (Implies (Implies P Q) P) P)
2
3    [1 Assume (Implies (Implies P Q) P)
4       [2 Assume (Not P)
5          [3 Assume P
6             [4 Bottom (3 2)]
7             [5 Q (4)]]
8          [6 (Implies P Q) (5 3)]
9          [7 P (3 1)]
10         [8 Bottom (2 7)]]
11      [9 (Not (Not P)) (2 8)]
12      [10 P (9)]]
13   [G (Implies (Implies (Implies P Q) P) P) (1 10)]
14   }
```

We changed the number of the last step to 'G', which is the idiomatic name for the last step, which is required to be equivalent to the goal.

Notice in the above proof, we still make reference to the assumption, when we take things out of an assumption, that is, when performing an arrow introduction. As described in section 6.4.2, they are not needed. We could drop those references in justifications, and rewrite the proof as:

```
1  {Proof (Implies (Implies (Implies P Q) P) P)
2
3    [1 Assume (Implies (Implies P Q) P)
4       [2 Assume (Not P)
5          [3 Assume P
6             [4 Bottom (3 2)]
7             [5 Q (4)]]
8          [6 (Implies P Q) (5)]
9          [7 P (3 1)]
10         [8 Bottom (2 7)]]
11      [9 (Not (Not P)) (8)]
12      [10 P (9)]]
13   [11 (Implies (Implies (Implies P Q) P) P) (10)]
14   }
```

However, since we do not use natural deduction rules at all, it is possible to greatly simplify the proof, for example, we may obtain $\perp$ from the assumptions $(P \to Q) \to P$ and $\neg P$ directly:

```
1  {Proof (Implies (Implies (Implies P Q) P) P)
2
3    [1 Assume (Implies (Implies P Q) P)
4       [2 Assume (Not P)
```

```
        [8 Bottom (1 2)]]
      [9 (Not (Not P)) (8)]
      [10 P (9)]]
  [G (Implies (Implies (Implies P Q) P) P) (10)]
  }
```

Or, since Peirce's law is valid, we can skip all steps, and obtain the result in one step:

```
{Proof (Implies (Implies (Implies P Q) P) P)
  [G (Implies (Implies (Implies P Q) P) P) ()]
  }
```

This proof is included in `example-propositional-1.prf`.

E.2 First-order Logic

E.2.1 Non-sorted First-order Logic

In classical, non-sorted first-order logic, we have the following theorem:

Theorem E.2. Not forall is equivalent to exists not
$\neg\forall x.P(x)$ and $\exists x.\neg P(x)$ are equivalent, for any predicate P.

It can be proven by:

1.	$\neg\forall x.P(x)$	Assume
2.	$\neg\exists x.\neg P(x)$	Assume
3.	a	$\forall$-Intro const
4.	$\neg P(a)$	Assume
5.	$\exists x.\neg P(x)$	$\exists$-Intro 4
6.	$\bot$	$\bot$-Intro 2 5
7.	$\neg\neg P(a)$	$\neg$-Intro 4 6
8.	$P(a)$	$\neg\neg$-Elim 7
9.	$\forall x.P(x)$	$\forall$-Intro 3 8
10.	$\bot$	$\bot$-Intro 1 9
11.	$\neg\neg\exists x.\neg P(x)$	$\neg$-Intro 2 10
12.	$\exists x.\neg P(x)$	$\neg\neg$-Elim 11
13.	$\exists x.\neg P(x)$	Assume
14.	$\neg P(a)$	Assume
15.	$\forall x.P(x)$	Assume
16.	$P(a)$	$\forall$-Elim 15
17.	$\bot$	$\bot$-Intro 14 16
18.	$\neg\forall x.P(x)$	$\neg$-Intro 15 17
19.	$\neg\forall x.P(x)$	$\exists$-Elim 13 14 18
20.	$\neg\forall x.P(x) \leftrightarrow \exists x.\neg P(x)$	$\leftrightarrow$-Intro 1 12 13 19

The above proof is of special interest to us as it uses all four natural deduction rules involving first-order quantifiers.

To prove a theory in non-sorted first-order logic in our system, we assign the `Atom` type to all constants and variables occuring in the original theory, and declare each predicate as one predicate that takes a certain number of `Atom`s. The above proof can thus be translated into:

```
{Declare
 [Predicate P Atom]}

{Proof (If-and-only-if (Not (Forall Atom x (P x)))
                       (Exists Atom x (Not (P x))))
  [1 Assume (Not (Forall Atom x (P x)))
     [2 Assume (Not (Exists Atom x (Not (P x))))
        [3 Take Atom a
           [4 Assume (Not (P a))
              [5 (Exists Atom x (Not (P x))) (4)]
              [6 Bottom (2 5)]]
           [7 (Not (Not (P a))) (6)]
           [8 (P a) (7)]]
        [9 (Forall Atom x (P x)) (8)]
        [10 Bottom (9 1)]]
     [11 (Not (Not (Exists Atom x (Not (P x))))) (10)]
     [12 (Exists Atom x (Not (P x))) (11)]]

  [13 Assume (Exists Atom x (Not (P x)))
      [14 Take Atom a such that (Not (P a))
          [15 Assume (Forall Atom x (P x))
              [16 (P a) (15)]
              [17 Bottom (16 14)]]
          [18 (Not (Forall Atom x (P x))) (17)]]
      [19 (Not (Forall Atom x (P x))) (13 18)]]

  [20 (If-and-only-if (Not (Forall Atom x (P x)))
                      (Exists Atom x (Not (P x))))
      (12 19)]
 }
```

When we introduce an arbitrary variable to prove that some property holds for all variables of that type, we use an introduction step to take an arbitrary variable. When we introduce an variable and assume that it satisfies some properties to perform ∃-Elimination, we use an introduction step with assumptions, and refer to the step in which the existential quantification if proved in the step where we perform ∃-Elimination.

Notice that we have also removed all unnecessary references to assumptions.

Since theorem E.2 is true, we can also skip to the result in one step:

```
{Declare
 [Predicate P Atom]}
```

```
3
4  {Proof (If-and-only-if (Not (Forall Atom x (P x)))
5                         (Exists Atom x (Not (P x))))
6    [G (If-and-only-if (Not (Forall Atom x (P x)))
7                       (Exists Atom x (Not (P x))))
8       ()]
9    }
```

This proof is included in `example-first-order-1.prf`.

E.2.2 Stylised Proof in Non-sorted First-order Logic

Theorem E.3. Green Dragons are Happy

Given that the following facts about dragons are true:

1. A dragon is happy if all of its children can fly.
2. All green dragons can fly.
3. Something is green if at least one of its parents is green.
4. All of the children of a dragon are also dragons.
5. If y is a child of x, then x is a parent of y.

and that, *all green dragons are happy*

Theorem E.3 appeared in the reasoning about programs lecture notes, as an exercise for constructing stylised proofs [29]. We will not present the natural deduction proof here as it has over 30 steps, but it is available in the lecture notes and included in the example program `example3.prf`. A stylised proof in the lecture notes uses proves the same theorem with only 11 steps. Formulated in our language, the stylised proof would be:

```
1  {Declare
2   [Predicate Dragon Atom]
3   [Predicate Green Atom]
4   [Predicate Fly Atom]
5   [Predicate Happy Atom]
6   [Predicate Parent-of Atom Atom]
7   [Predicate Child-of Atom Atom]}
8
9  {Lemma
10  [1 (Forall Atom x
11       (Implies (And (Dragon x)
12                     (Forall Atom y
13                       (Implies (Child-of y x) (Fly y))))
14                (Happy x)))]
15  [2 (Forall Atom x
16       (Implies (And (Green x)
17                     (Dragon x))
18                (Fly x)))]
```

```
19   [3 (Forall Atom x
20        (Implies (Exists Atom y
21                     (And (Parent-of y x)
22                          (Green y)))
23                   (Green x)))]
24   [4 (Forall Atom x
25        (Forall Atom y
26          (Implies (And (Child-of x y)
27                          (Dragon y))
28                     (Dragon x))))]
29   [5 (Forall Atom x
30        (Forall Atom y
31          (Implies (Child-of x y)
32                     (Parent-of y x))))]
33   }
34
35  {Proof (Forall Atom x (Implies (Dragon x)
36                                   (Implies (Green x) (Happy x))))
37
38    [I Take Atom smaug
39       [ass1 Assume (Dragon smaug)
40             [ass2 Assume (Green smaug)
41                   [6 (Forall Atom x (Forall Atom y (Implies (And (Parent-of y x)
42                                                                  (Green y))
43                                                             (Green x))))
44                      (3)]
45                   [7 (Forall Atom x (Forall Atom y (Implies (And (Child-of x y)
46                                                                  (Green y))
47                                                             (Green x))))
48                      (5 6)]
49                   [8 (Forall Atom x (Implies (Child-of x smaug)
50                                               (Green x)))
51                      (7 ass2)]
52                   [9 (Forall Atom x (Implies (Child-of x smaug)
53                                               (Dragon x)))
54                      (4 ass1)]
55                   [10 (Forall Atom x (Implies (Child-of x smaug)
56                                                (And (Green x) (Dragon x))))
57                       (8 9)]
58                   [11 (Forall Atom x (Implies (Child-of x smaug)
59                                                (Fly x)))
60                       (10 2)]
61                   [12 (Happy smaug) (1 ass1 11)]]
62             [13 (Implies (Green smaug) (Happy smaug)) (12)]]
63       [14 (Implies (Dragon smaug) (Implies (Green smaug) (Happy smaug))) (13)]]
64    [G (Forall Atom x
65         (Implies (Dragon x) (Implies (Green x) (Happy x))))
66       (14)]
67    }
```

The ‘facts about dragons’, which are ‘given’ to us, are put into a lemma section to separate them from the rest of the proof. Its is also possible to write them as steps `Given` to be true, but it would use more space.

Our representation uses slightly more steps because we need to take results out from assumptions, one assumption at a time. If we allow access to results obatined within nested assumptions, as defined in definition 6.14, then we could also omit steps 13 and 14.

Notice that E.3 can be proven directly in one step as well:

```
68 {Proof (Forall Atom x (Implies (Dragon x)
69                                 (Implies (Green x) (Happy x))))
70   [G (Forall Atom x (Implies (Dragon x)
71                               (Implies (Green x) (Happy x))))
72      (1 2 3 4 5)]
73   }
```

This proof is included in `example-first-order-2.prf`.

E.2.3 Stylised Proof in Many-sorted First-order Logic

The lecture note formalises the theorem using non-sorted logic, where the predicate $Dragon(x)$ is used to say that ‘x is a dragon’. We could represent the same statement using a type `Dragon`, so that if a term has type `Dragon`, it is a dragon. The proof could then be written as:

```
1  {Declare
2   [Type Dragon]
3
4   [Predicate Green Dragon]
5   [Predicate Fly Dragon]
6   [Predicate Happy Dragon]
7   [Predicate Parent-of Dragon Dragon]
8   [Predicate Child-of Dragon Dragon]}
9
10 {Lemma
11  [1 (Forall Dragon x (Implies (Forall Dragon y (Implies (Child-of y x) (Fly y)))
12                               (Happy x)))]
13  [2 (Forall Dragon x(Implies (Green x) (Fly x)))]
14  [3 (Forall Dragon x (Implies (Exists Dragon y (And (Parent-of y x) (Green y)))
15                               (Green x)))]
16  [4 (Forall Dragon x (Forall Dragon y (Implies (Child-of x y) (Parent-of y x))))]
17  }
18
19 {Proof (Forall Dragon x (Implies (Green x) (Happy x)))
20
21   [I Take Dragon smaug
22      [ass1 Assume (Green smaug)
23            [5 (Forall Dragon x (Forall Dragon y
24                                 (Implies (And (Parent-of y x) (Green y))
```

```
25                                              (Green x))))
26              (3)]
27           [6 (Forall Dragon x (Forall Dragon y
28                                   (Implies (And (Child-of x y)
29                                                 (Green y))
30                                            (Green x))))
31              (4 5)]
32           [7 (Forall Dragon x (Implies (Child-of x smaug)
33                                        (Green x)))
34              (6 ass1)]
35           [8 (Forall Dragon x (Implies (Child-of x smaug)
36                                        (Fly x)))
37              (7 2)]
38           [9 (Happy smaug) (1 ass1 8)]]
39     [10 (Implies (Green smaug) (Happy smaug)) (9)]]
40  [G (Forall Dragon x (Implies (Green x) (Happy x))) (10)]
41  }
```

Since we now represent some of the information in the theorem with types, the length of the proof is even shorter.

This proof is included in `example-first-order-3.prf`.

E.3 Arithmetics

Both SMT solvers we use have extensive support for

Theorem E.4.

$$\forall x, y : \mathbb{Z}.(x+y)^2 = x^2 + y^2 + 2xy$$

E.4 Induction

Inductive reasoning is extremely useful when reasoning about functional programs, where recursively defined functions and data structures are used extensively. However, the SMT solvers we researched have limited inductive reasoning capailities: Z3 cannot prove 'inductive facts' [46], and as we will show in this section, neither Z3 nor CVC4 could perform induction over function definitions.

Therefore, Emmy cannot directly prove inductive statements using the underlying solvers. Instead, the user must explicitly instruct our system to perform induction, and our system will generate suitable induction principles for this purpose. In this section we present how induction is performed, how inductive proofs are structured in our system, and how do they compare to stylised proofs.

We discuss the exact proving abilities of SMT solvers in section 9.3.

E.4.1 Over Recursive Data Structures

For recursive data structures, we

Consider the following definitions, adapted from the lecture notes [28] [1]:

```
data List = nil | cons Int List

append :: List -> List -> List
rev :: List -> List

append nil l = l
append (cons x xs) l = cons x (append xs l)

rev nil = nil
rev (cons x xs) = append (rev xs) (cons x nil)
```

We wish to prove that

Theorem E.5. For any `List`s `l1` and `l2`, `rev (append l1 l2)` is equal to `append (rev l2) (rev l1)`.

We can use the following lemmas for the proof:

Lemma E.1. `append` has right identity
For all `List` `l`, `append l nil` is equal to `l`

Lemma E.2. `append` is associative
For all `List` `l1`, `l2`, and `l3`, `append (append l1 l2) l3` is equal to `append l1 (append l2 l3)`.

Before we proceed to prove E.5, we express the data structure declarations, function declarations, and function definitions in our language:

```
1  {Declare
2   [Data List [nil] [cons Int List]]
3   [Function append List List -> List]
4   [Function rev List -> List]}
5
6  {Define
7   [append-1 (append nil l) l]
8   [append-2 (append (cons x xs) l) (cons x (append xs l))]
9
10  [rev-1 (rev nil) nil]
11  [rev-2 (rev (cons x xs)) (append (rev xs) (cons x nil))]}
```

Then, we put the two lemmas available into a lemma section:

```
12 {Lemma
13  [append-ident (Forall List l (= (append l nil) l))]
14  [append-assoc (Forall List l1
15                  (Forall List l2
16                    (Forall List l3 (= (append (append l1 l2) l3)
17                                       (append l1 (append l2 l3))))))]}
```

[1] The lecture notes do not give explicit definitions for `[]` and `++`, which correspondes to `List` and `append` in our definitions. Instead, it uses a few lemmas about the properties of `++`. Our definition of `append` is the same as the definition of `++` in the Haskell prelude [22].
Also notice that our `List`s are not polymorphic, because we have not implemented parametric polymorphism.

We start the proof by stating our goal:

```
18 {Proof (Forall List l1 (Forall List l2
19                         (= (rev (append l1 l2))
20                            (append (rev l2) (rev l1)))))
```

The actual proof clearly requires structural induction. We choose to perform induction over the definition l1, and split the goal into two cases: one base case and one inductive case.

To begin, we write an induction step. Since we will be able to prove the goal of the proof in one step, we name this step G, for 'goal:

```
21   [G Induction (Forall List l1 (Forall List l2
22                                 (= (rev (append l1 l2))
23                                    (append (rev l2) (rev l1)))))
```

Then comes the base case, in which we do not make any induction hypothesis, and show directly, by definition and lemma E.1, that the theorem E.5 holds when l1 is replaced by nil.

```
24      [Show (Forall List l2 (= (rev (append nil l2))
25                               (append (rev l2) (rev nil))))
26
27       By (append-ident append-1 rev-1)]
```

For the induction step, we make some assumptions. We first take an List l1, and make the inductive hypothesis that theorem E.5 holds for l1. Then, we take an arbitrary Int x, and prove that the property holds for cons x l1.

```
28      [Take List l1 such that (Forall List l2 (= (rev (append l1 l2))
29                                                 (append (rev l2) (rev l1))))
30       Take Int x
31       Show (Forall List l2 (= (rev (append (cons x l1) l2))
32                               (append (rev l2) (rev (cons x l1)))))
```

The subproof for this case is relatively straightforward. We build up a chain of equalities to show that the term rev (append (cons x l1) l2) is equal to some intermediate result, and eventually, that it is equal to append (rev l2) (rev (cons x l1)), which is the goal we wish to prove in this case.

```
33       [II Take List l2
34           [I = (rev (append (cons x l1) l2))
35              = (rev (cons x (append l1 l2)))               (append-2)
36              = (append (rev (append l1 l2)) (cons x nil))  (rev-2)
37              = (append (append (rev l2) (rev l1))
38                        (cons x nil))                       (%Hypo)
39              = (append (rev l2)
```

```
40                          (append (rev l1) (cons x nil)))     (append-assoc)
41                  = (append (rev l2) (rev (cons x l1)))       (rev-2)]]]
42        ]
43    }
```

Notice that in the third equality, we refer to the induction hypothesis that the property holds for `l1`, by using `%Hypo` as a justification.

Notice that we cannot prove theorem E.5 without explicitly stating that we wish to perform induction. As neither Z3 nor CVC4 seem to be able to prove E.5 in its original form directly. The correctness of the following proof cannot be determinined, so it would be considered incorrect by out system, although semantically it is correct.

```
18  {Proof (Forall List l1 (Forall List l2
19                              (= (rev (append l1 l2))
20                                 (append (rev l2) (rev l1)))))
21
22    [G (Forall List l1 (Forall List l2
23                           (= (rev (append l1 l2))
24                              (append (rev l2) (rev l1)))))
25       (append-1 append-2 rev-1 rev-2 append-ident append-assoc)]
26    }
```

However, we can use an induction marker to instruct the system to perform induction over `l1`. Emmy will generate induction principles for all formulae marked by induction markers, so we are now asking the underlying solver to prove the generated inductive principle, instead of the original formula.

The following program, which puts the entire goal inside an induction marker, will be deemed correct:

```
18  {Proof (Forall List l1 (Forall List l2
19                              (= (rev (append l1 l2))
20                                 (append (rev l2) (rev l1)))))
21
22    [G (Ind: (Forall List l1 (Forall List l2
23                                 (= (rev (append l1 l2))
24                                    (append (rev l2) (rev l1))))))
25       (append-1 append-2 rev-1 rev-2 append-ident append-assoc)]
26    }
```

This proof is included in `theorem-10.prf`.

Another Example: Two Different `sum` Functions

Consider another set of definitions adapted from the lecture notes [28]:

```
data List = nil | cons Int List

sum :: List -> Int
```

```
sum nil = 0
sum (cons i is) = i + sum is

sumTr :: List -> Int -> Int
sumTr nil k = k
sumTr (cons i is) k = sumTr is (i + k)
```

We wish to prove that

Theorem E.6. `sum is` and `sumTr is 0` are equal
For all `List is`, `sum is` is equal to `sumTr is 0`.

We begin the proof by declaring the types and functions, and defining the functions:

```
1  {Declare
2   [Data List [nil] [cons Int List]]
3   [Function sum List -> Int]
4   [Function sumTr List Int -> Int]}
5
6  {Define
7   [sum-B (sum nil) 0]
8   [sum-I (sum (cons i is)) (+ i (sum is))]
9
10  [sumTr-B (sumTr nil k) k]
11  [sumTr-I (sumTr (cons i is) k) (sumTr is (+ i k))]}
```

It is mostly a straightforward translation from the Haskell definitions.
We first state our proof goal:

```
12 {Proof (Forall List is (= (sum is) (sumTr is 0)))
```

Directly proving theorem E.6 is not possible. We need to first prove a stronger theorem which entails theorem E.6:

Theorem E.7. `x + sum is` and `sumTr is x` are equal
For all `List is`, and integer `x`, `x + sum is` is equal to `sumTr is x`.

We prove theorem E.7 in an induction step.

```
13   [L Induction
14      (Forall List is (Forall Int x (= (+ x (sum is)) (sumTr is x))))
15
16      [Take Int x
17
18       Show (= (+ x (sum nil)) (sumTr nil x))
19       By (sum-B sumTr-B)]
20
21      [Take List is such that (Forall Int x (= (+ x (sum is)) (sumTr is x)))
22       Take Int i
```

```
23        Take Int x
24
25        Show (= (+ x (sum (cons i is))) (sumTr (cons i is) x))
26        By (sum-I sumTr-I %Hypo)]]
```

The goal would follow directly.

```
27    [G (Forall List is (= (sum is) (sumTr is 0))) (L)]}
```

This proof is included in `theorem-11.prf`.

E.4.2 Over Recursive Function Definitions

When proving properties of recursively defined functions, we split

Consider the following definitions of functions `g` and `g'`, taken from the lecture notes [27]:

```
g :: Int -> Int -> Int
g i j = gp i j 0 0

g' :: Int -> Int -> Int -> Int -> Int
g' i j cnt acc
  | cnt >= i    = acc
  | otherwise   = gp i j (1 + cnt) (j + acc)
```

Which, mathematically, is equivalent to:

$$g(i,j) = g'(i,j,0,0)$$

$$g'(i,j,cnt,acc) = \begin{cases} acc & \text{if } cnt \geq i \\ g'(i,j,1+cnt,j+acc) & \text{otherwise} \end{cases}$$

Notice that although in 'normal' cases,

The lecture notes then gives the following theorem and proves it via induction in a stylised proof:

Theorem E.8. `g` performs multiplication

For all non-negative integers x, y, `g x y` is equal to `x * y`.

We need to limit the domain to positive integers, because while

To prove the above, we need to prove the following lemma:

Lemma E.3. `g'` performs multiplication

For all non-negative integers `x` and `y`, and integers `cnt` and integer `acc`, if `x` is greater than or equal to `cnt`, then `g' x y cnt acc` is equal to `(x - cnt) * y + acc`.

As before, we limit the range of `x` and `y` by assuming them to be non-negative. We also assume `cnt` to be less than or equal to `x`, because we only care what happens if we call `g' x y 0 0` when calling `g x y`. We don't care what happens if `g'` is applied other arguments.

It is easy to see that if lemma E.3 is true, then `g' x y 0 0` would be equal to `x * y`, and theorem E.8 would hold according to the definition of `g`.

To begin with our proof, as before, we first declare the functions:

```
1  {Declare
2   [Function g Int Int -> Int]
3   [Function gp Int Int Int Int -> Int]}
```

We renamed `g'` to gp as our syntax does not support `'` in identifiers. As our built-in `Int` type includes negative integers, we will add range constraints as assumptions when formulating the theorem and in our proof.

The definition of `g` is straightforward. While for `g'`, we use a function definition with cases that corresponds with the cases in the Haskell definition we wrote with guards.

```
4  {Define
5   [g-1 (g i j) (gp i j 0 0)]
6   [gp-1 (gp i j cnt acc)
7         cases
8         [(>= cnt i) acc]
9         [otherwise (gp i j (+ 1 cnt) (+ j acc))]]
10  }
```

Our proof goal would be theorem E.8. We state the non-negativity of `x` and `y` explicitly, by making it the antecedent of the theorem we wish to prove.

```
11 {Proof (Forall Int x
12         (Forall Int y
13           (Implies (And (>= x 0) (>= y 0))
14                    (= (g x y) (* x y)))))
```

We first prove lemma E.3 in one induction step. We state that we are performing induction on the definition of gp-1. Then, we use an induction marker in the goal of the induction step to mark the 'body' of lemma E.3 inside the quantifiers. Like before, we state that `x` and `y` are non-negative explicitly.

```
15   [L Induction on gp-1
16      (Forall Int x (Implies (>= x 0)
17        (Forall Int y (Implies (>= y 0)
18          (Forall Int cnt
19            (Forall Int acc
20              (Implies (>= x cnt)
21                       (Ind: (= (gp x y cnt acc)
22                                (+ (* (- x cnt) y) acc))))))))))
```

We name this step `L` as we prove the lemma in this step.

The definition gp-1 contains two cases. We first deal with the base case, where `x` is less than or equal to `cnt`, and `gp x y cnt acc` is equal to `acc`:

```
23      [Take Int x such that (>= x 0)
24       Take Int y such that (>= y 0)
```

```
25        Take Int cnt
26        Take Int acc
27
28        Show (Implies (= cnt x)
29                      (= acc (+ (* (- x cnt) y) acc)))
30        By ()]
```

Notice that since the case condition that `x` ≤ `cnt` and `x` ≥ `cnt` both holds, we only need to prove that the property holds when `x` is equal to `cnt`. Emmy is good at handling minor variations across equivalent arithmetic properties.

The goal of the base case can be easily proven without any justification, so we just write `By ()`.

We then move on to the induction step. In the definition of `g'`, the case condition for the second case is `otherwise`. However, since we require function definition cases to be exhaustive, and that there are only two cases, we can write `cnt` < `x` as the case condition, which is the negation of the condition of the first case.

In an inductive step, we need to introduce an 'result' variable `r`, whose value is equal to the result of the recursive call. We also assume that the lemma E.3, the inductive property we wish to prove, holds for r. Then, we proceed to prove that under these assumptions, the value of the call to g' would satisfy the property as well.

```
31       [Take Int x such that (>= x 0)
32        Take Int y such that (>= y 0)
33        Take Int cnt
34        Take Int acc
35        Take Int r such that (And (= r (gp x y (+ 1 cnt) (+ y acc)))
36                                  (= r (+ (* (- x (+ 1 cnt)) y) (+ y acc))))
37
38        Show (Implies (< cnt x)
39                      (= r (+ (* (- x cnt) y) acc)))
40        By (%Hypo)]]
```

In this case, the goal of this case follows from the induction hypothesis, denoted by `%Hypo`.

The goal would trivially follow from lemma E.3 and the definition of `g`.

```
41    [G (Forall Int x
42         (Forall Int y
43           (Implies (And (>= x 0) (>= y 0))
44                    (= (g x y) (* x y)))))
45       (L g-1)]
46    }
```

Lemma E.3 can also be proven in one simple step automatically, given that we use an induction marker to state that we wish to perform induction on the definition `gp-1`:

```
11 {Proof (Forall Int x
12          (Forall Int y
13            (Implies (And (>= x 0) (>= y 0))
```

```
14                     (= (g x y) (* x y)))))
15
16   [L (Forall Int x (Implies (>= x 0)
17       (Forall Int y (Implies (>= y 0)
18         (Forall Int cnt
19           (Forall Int acc
20             (Implies (>= x cnt)
21                      (Ind: gp-1 (= (gp x y cnt acc)
22                                   (+ (* (- x cnt) y) acc))))))))))
23     (gp-1)]
24
25   [G (Forall Int x
26       (Forall Int y
27         (Implies (And (>= x 0) (>= y 0))
28                  (= (g x y) (* x y)))))
29     (L g-1)]
30   }
```

Without the induction marker, it cannot be proven automatically.

This proof is included in `theorem-21.prf`.