Development and Verification of a Proof Assistant

Alexander Birch Jensen

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

The thesis describes the development of a formalization in Isabelle of a first-order logic proof system, which serves as the inference kernel of a proof assistant developed by John Harrison. By verifying the soundness of the kernel, we verify the soundness of Harrison's proof assistant as all of its functionalities are derived from the kernel. From the verified formalization of the kernel, we generate executable code to obtain a verified implementation of Harrison's proof assistant. We test the implementation by replacing the existing kernel by generated kernel to gain confidence in the correctness of the formalization.

Preface

This thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring an MSc in Computer Science and Engineering. The thesis deals with formalization and verification in the proof assistant Isabelle of a first-order logic inference kernel.

The thesis is for 30 ECTS in the period from 18th of March 2016 to 18th of August 2016. I was supervised by Jørgen Villadsen, and co-supervised by Anders Schlichtkrull.

Prior to the start of my work on this thesis, I had only basic knowledge of Isabelle. I have had courses on many different topics in logic, including artificial intelligence, multi-agent systems and logic programming. Also during my semester abroad at TU Wien, I worked on developing a tactic language for the proof theory framework GAPT (General Architecture for Proof Theory), where I was supervised by Stefan Hetzl and Alexander Leitsch.

Prior to submission of this thesis, I did collaborative work with Jørgen Villadsen and Anders Schlichtkrull on the paper *Verification of an LCF-Style First-Order Prover with Equality*, which describes the verification presented in my thesis. The formalization has since then received numerous minor improvements. The paper can be found at the web page `http://www21.in.tum.de/~nipkow/Isabelle2016/`.

I worked on a project called NaDeA (Natural Deduction Assistant) in collaboration with Jørgen Villadsen and Anders Schlichtkrull, where I was mainly responsible for the implementation of the tool. NaDeA is a simple proof assistant in which first-order formulas can be proved in a verified natural deduction proof system by manually single-stepping through the rules. The tool can be found at the web page `https://nadea.compute.dtu.dk`.

I would like to thank my supervisor Jørgen Villadsen for his tireless efforts to provide help and feedback in all aspects of my work on this thesis.

I would also like to thank my co-supervisor Anders Schlichtkrull; especially for helping me to overcome hurdles during the development of the formalization, and for comments on my work.

Finally, I would also like to Jasmin Christian Blanchette for providing remarkably helpful and honest comments on my work during the critical final stages.

Lyngby, 18-08-2016

Alexander Birch Jensen

Contents

Chapter 1

Introduction

The reader is expected to have at least a basic background in logic, including proof systems. It is beneficial, although not strictly required, to have experience with functional programming.

1.1 Motivation

Proof assistants are computer programs that provide interactive guidance for constructing proofs. In proof assistants, we can formalize and verify mathematical theorems, algorithms and computer systems. Proof assistants usually take advantage of built-in ATPs (automated theorem provers) to assist in finding proofs. Recent developments include the verification of the seL4 Microkernel [KEH+09]. The functional correctness of the microkernel has been verified. This means that the microkernel has been proved to function correctly in regard to its abstract specification. The seL4 microkernel runs on various current processors, some of which control critical parts of smartphones today, i.e. sending and receiving antenna signals. Verification of critical hardware and software components are prime examples of the usefulness of proof assistants.

Proof systems define how to construct a proof for a given logic, most commonly by rules and axioms. Proof systems are used as the underlying machinery of proof checking in proof assistants. Certain proof assistants, like Isabelle, require that all proofs are built up from a small inference kernel. An inference kernel is in essence an implementation of a proof system. The most commonly used instance of Isabelle is based on higher-order logic (HOL).

> 1. Proofs in Isabelle, even those generated from automated tactics, are justified by a minimal inference kernel. In contrast to ATPs, which are complex pieces of software, it is far less likely that a kernel-certified proof is unsound. 2. Isabelle's premier logic, HOL, has seen decades of development of rich mathematical libraries and formalizations such as Archive of Formal Proofs. Proofs carried out in Isabelle have access to this knowledge, which means that there is a greater potential for reuse of existing developments. [HK16, p. 2]

Examples of proof assistants that have a verified kernel are Harrison's verification of HOL Light [Har06] and its extension by Kumar, Arthan, Myreen and Owens [KAMO16].

We will use the proof assistant Isabelle. Isabelle features a declarative language that is inspired by mathematical practice. The language is composed of logic formulas and structured proofs. Consider the classical first-order logic (FOL) theorem known as the drinker paradox:

$$\exists x.\, D(x) \longrightarrow (\forall x.\, D(x))$$

The formula looks slightly different in HOL, which is used in Isabelle, as we write $D(x)$ as $D\ x$. Here is a rather detailed proof in Isabelle:

```
theorem ∃ x. D x ⟶ (∀ x. D x)
proof (cases ∀ x. D x)
  case True
  then have P ⟶ (∀ x. D x) for P ..
  then show ?thesis ..
next
  case False
  then have ∃ x. ¬ D x unfolding not-all .
  then obtain a where ¬ D a ..
  then have ¬ D a ∨ (∀ x. D x) ..
  then have D a ⟶ (∀ x. D x) unfolding disj-not1 .
  then show ?thesis ..
qed
```

We consider the cases for the universally quantified formula; either it is true or false. If $\forall x.\, D\ x$ is true, then the implication is true regardless of the left-hand side, which makes the whole formula true. In the case where $\forall x.\, D\ x$ is false, there must exist an individual a such that $\neg D\ a$. Choosing this a as witness to the existential quantification makes the left-hand side of the implication false, which makes the whole formula true. We can save the labor of writing out the proof structure by using the automated simplification method:

theorem $\exists\, x.\ D\ x \longrightarrow (\forall\, x.\ D\ x)$
by *simp*

Following formalization in Isabelle, it is possible to generate executable code for the defined data types and functions in different programming languages, including Standard ML (SML). If the data types and functions we generate code for has been verified, we know that the verified properties also hold for the generated code.

1.2 Problem Statement

In John Harrison's *Handbook of Practical Logic and Automated Reasoning*, an axiomatic proof system based on FOL is the kernel of a proof assistant that is implemented in OCaml [Har09, p. 477].

The aim of this thesis is to:

(1) based on Harrison's code, develop in Isabelle a formalization of the kernel of a proof assistant based on FOL,

(2) verify the formalization in Isabelle, and

(3) from the formalization generate SML code. To gain confidence that the formalization is correct, we match the results of the generated code against the existing code using a test suite.

The implementation also has an SML version developed by Schlichtkrull and Villadsen [SV] .We will replace the code for the kernel of the SML version with generated code.

1.3 Structure of the Thesis

The contents of the remaining chapters are summarized below:

- Chapter 2 defines the syntax and semantics of FOL. The addition of equality to our logic is described and discussed.
- Chapter 3 discusses proof systems. We define Harrison's axiomatic proof system. The soundness and completeness of such an axiomatic proof system is defined and discussed.
- Chapter 4 serves as a brief introduction to the proof assistant Isabelle. In particular, the programming language of Isabelle/HOL and the proof language Isar (Intelligible Semi-Automated Reasoning).
- Chapter 5 formalizes FOL and the proof system in Isabelle in relation to the OCaml code in Harrison's handbook.
- Chapter 6 presents a formalized soundness proof in Isabelle.
- Chapter 7 shows how to generate executable code for the formalized proof system. Tests of the generated code are compared to the SML version of Harrison's proof assistant.
- Chapter 8 presents and discusses experiments with the declarative proof language in Harrison's proof assistant.
- Chapter 9 concludes on the final results and reflects on further work.

Chapter 2

First-Order Logic

This chapter briefly discusses first-order logic from a historical perspective, and then proceeds to define the syntax and semantics of first-order logic in a format that can serve as the starting point for our formalization.

Initially, the study of logic was the study of valid reasoning.

> Logic formalizes valid methods of reasoning. The study of logic was begun by the ancient Greeks whose educational system stressed competence in reasoning and in the use of language. [BA12, p. 1]

In particular, the logical nature of arguments in natural languages were studied. Consider as an example the following two statements that give rise to intuitive logical reasoning:

(1) If it rains the sewers will flood.

(2) It rains.

If we accept those statements, it must follow that the sewers will flood. We can formalize the example statements in propositional logic. However, for more complex statements atomic propositions are insufficient:

(1) All movies produced by Michael Bay has explosions.

(2) The movie Transformers is produced by Michael Bay.

We cannot formalize the statements in propositional logic as the statement (1) requires the use of variables and quantifiers, as it holds for all movies, and (2) states a fact about one movie in particular.

> Propositional logic is not sufficiently expressive for formalizing mathematical theories such as arithmetic. An arithmetic expression such as $x + 2 > y - 1$ is neither true nor false: (a) its truth depends on the values of the variables x and y; (b) we need to formalize the meaning of the operators + and − as functions that map a pair of numbers to a number; (c) relational operators like $>$ must be formalized as mapping pairs of numbers into truth values. The system of logic that can be interpreted by values, functions and relations is called first-order logic (also called predicate logic or the predicate calculus). [BA12, p. 3]

Today first-order logic is in many ways the standard for teaching in mathematics and computer science, but it was not before Hilbert and Skolem that first-order logic came into existence.

Interestingly, first-order logic did not receive appreciation from mathematicians and computer scientists about a hundred years ago when it was first introduced. Before the rise of first-order logic, higher-order logic was the most widely used. Higher-order logic can not only quantify over individuals, but sets of individuals, sets of sets of individuals, and so on. Higher-order logic has later also proved itself useful in applications of computer science, in particular for proof assistants.

> Experience with HOL over decades has demonstrated that higher-order logic is widely applicable in many areas of mathematics and computer science. In a sense, Higher-Order Logic is simpler than First-Order Logic, because there are fewer restrictions and special cases. Note that HOL is weaker than FOL with axioms for ZF set theory, which is traditionally considered the standard foundation of regular mathematics, but for most applications this does not matter. [W+16]

In 1986, Pelletier gave seventy-five problems to test the capabilities of an automated theorem prover. Problem 43 is to define set equality as having exactly the same members, and prove that set equality is symmetric. [Pel86, p. 201]:

$$(\forall x.\, \forall y.\, Q(x,y) \longleftrightarrow \forall z.\, P(z,x) \longleftrightarrow P(z,y)) \longrightarrow \forall x.\, \forall y.\, Q(x,y) \longleftrightarrow Q(x,y)$$

$P(x,y)$ is set membership ($x \in y$) and $Q(x,y)$ is set equality ($x = y$). The formula may look deceptively simple, but automatic proof procedures often run into problems. For a proof assistant like Isabelle the formula is easily proved by full automation.

lemma $(\forall x.\ \forall y.\ (Q\ x\ y) \longleftrightarrow (\forall z.\ (P\ z\ x) \longleftrightarrow (P\ z\ y)))$
$\longrightarrow (\forall x.\ \forall y.\ (Q\ x\ y) \longleftrightarrow (Q\ y\ x))$
by *auto*

We will try to prove the symmetry of set equality using Harrison's proof assistant in Chapter 8.

2.1 Syntax

In first-order logic, the syntax describes how to properly build formulas. Formulas may contain terms, namely variables and functions. Therefore, we first define the terms of first-order logic.

Definition 2.1 A *term* is defined as either:

- A variable x, where x is the variable symbol.

- A function $f(t_1, t_2, \ldots, t_n)$, $n \geq 0$, where f is the function symbol, and the terms $t_1, t_2, \ldots, t_n$ are the arguments to the function. Constants are functions with $n = 0$. ◇

We then define the formulas of first-order logic.

Definition 2.2 We use the letters A and B to denote arbitrary formulas.

A *formula* is defined as either:

- Truth $\top$.
- Falsity $\bot$.
- A predicate $P(t_1, t_2, \ldots, t_n)$, $n \geq 0$, where P is the predicate symbol, and the terms $t_1, t_2, \ldots, t_n$ are the arguments to the predicate. Propositional symbols are predicates with $n = 0$.
- An implication $A \longrightarrow B$.
- A bi-implication $A \longleftrightarrow B$.
- A conjunction $A \wedge B$.
- A disjunction $A \vee B$.
- A negation $\neg A$.
- An existentially quantification $\exists x.\, A$.
- A universal quantification $\forall x.\, A$. ◇

Our syntax is similar to Ben-Ari's in [BA12], but we leave out a few uncommon logical operators that are expressible by other operators. Furthermore, we also allow the constants truth and falsity. It should be noted that right-associativity is used for implication, i.e. $p \longrightarrow (q \longrightarrow r)$ can be written $p \longrightarrow q \longrightarrow r$.

2.2 Semantics

The semantics of first-order logic is the assignment of truth values to formulas. The meaning of a formula depends on the meaning of the used variables, predicates and functions.

A variable denotation assigns values to variables. As known from mathematics, a variable is a placeholder for some element. The universe defines the type of the elements that the variables point to. As an example, the variables could range over natural numbers.

We define our semantics similarly to [BA12], but we explicitly introduce the variable denotation, function denotation and predicate denotation.

Definition 2.3 A *variable denotation* E, also called environment, maps each variable symbol to an element of the universe. ◇

Just as variables are interpreted as elements of the universe, so are function symbols (with arguments).

Definition 2.4 A *function denotation* F maps a function symbol with arguments to an element of the universe. ◇

Lastly, we consider the interpretation of predicates. The predicates are the atomic propositions of FOL and can be assigned a truth value.

Definition 2.5 A *predicate denotation* G maps a predicate symbol with arguments to a boolean value. ◇

We are now ready to define the semantics of terms and formulas. When referring to the semantics of a given expression, we enclose it in double brackets $[\![$ and $]\!]$. The semantics of terms is defined recursively with regard to their structure.

Definition 2.6 Given a variable denotation E and a function denotation F, the *semantics of a term* is defined as follows:

- $[\![x]\!]_F^E = E(x)$, if x is a variable symbol.
- $[\![f(t_1, t_2, \ldots, t_n)]\!]_F^E = F(f, [\![t_1]\!]_F^E, [\![t_2]\!]_F^E, \ldots, [\![t_n]\!]_F^E)$, if f is a function symbol, and t_k for $k = 1, 2, \ldots, n$ is a term. ◇

The semantics of formulas is defined in a similar way.

Definition 2.7 We use the letters A and B to denote arbitrary formulas.

Given a variable denotation E, a function denotation F, and a predicate denotation G, the *semantics of a formula* is defined as follows:

- $[\![\top]\!]^E_{(F,G)} = \mathit{true}$
- $[\![\bot]\!]^E_{(F,G)} = \mathit{false}$
- $[\![P(t_1, t_2, \ldots, t_n)]\!]^E_{(F,G)} = G\left(P, [\![t_1]\!]^E_F, [\![t_2]\!]^E_F, \ldots, [\![t_n]\!]^E_F\right)$, if P is a predicate symbol, and t_k for $k = 1, 2, \ldots, n$ is a term
- $[\![A \longrightarrow B]\!]^E_{(F,G)} = \begin{cases} [\![B]\!]^E_{(F,G)}, & \text{if } [\![A]\!]^E_{(F,G)} \text{ is true} \\ \mathit{true}, & \text{otherwise} \end{cases}$
- $[\![A \longleftrightarrow B]\!]^E_{(F,G)} = \begin{cases} \mathit{true}, & \text{if } [\![A]\!]^E_{(F,G)} \text{ and } [\![B]\!]^E_{(F,G)} \text{ are equal} \\ \mathit{false}, & \text{otherwise} \end{cases}$
- $[\![A \wedge B]\!]^E_{(F,G)} = \begin{cases} [\![B]\!]^E_{(F,G)}, & \text{if } [\![A]\!]^E_{(F,G)} \text{ is true} \\ \mathit{false}, & \text{otherwise} \end{cases}$
- $[\![A \vee B]\!]^E_{(F,G)} = \begin{cases} \mathit{true}, & \text{if } [\![A]\!]^E_{(F,G)} \text{ is true, or if } [\![B]\!]^E_{(F,G)} \text{ is true} \\ \mathit{false}, & \text{otherwise} \end{cases}$
- $[\![\neg A]\!]^E_{(F,G)} = \begin{cases} \mathit{true}, & \text{if } [\![A]\!]^E_{(F,G)} \text{ is false} \\ \mathit{false}, & \text{otherwise} \end{cases}$
- $[\![\exists x.\, A]\!]^E_{(F,G)} = \begin{cases} \mathit{true}, & \text{if for some } e \in U : [\![A]\!]^{E[x \leftarrow e]}_{(F,G)} \text{ is true} \\ \mathit{false}, & \text{otherwise} \end{cases}$
- $[\![\forall x.\, A]\!]^E_{(F,G)} = \begin{cases} \mathit{true}, & \text{if for all } e \in U : [\![A]\!]^{E[x \leftarrow e]}_{(F,G)} \text{ is true} \\ \mathit{false}, & \text{otherwise} \end{cases}$ ◇

Some formulas are true for some variable denotations, function denotations and predicate denotations, but not in all. Those are the satisfiable formulas. However, we are mostly interested in formulas that are true for all possible denotations.

Definition 2.8 If $[\![A]\!]^E_{(F,G)} = \mathit{true}$ for all E, F and G then A is *valid*. ◇

A valid formula is thus also satisfiable. The valid formulas can tell us about truths that hold regardless of the context. In particular those formulas containing implications are of interest, as an implication can be understood as a rule.

2.3 Equality

So far we have considered first-order logic without equality where the binary predicate $=$ does not have a predefined semantics. Having equality available in the logic is desirable from a mathematical point of view. For the remainder of this thesis we will be considering first-order logic with equality. We can introduce equality either as a binary predicate with predefined semantics, or as an atomic logical operator. To follow the approach by Harrison, we introduce equality as a binary predicate, but where the usual infix syntax from mathematics $s = t$ can be used instead of $=(s, t)$.

We want equality to have the properties of reflexivity, symmetry and transitivity:

$$\forall x.\, x = x$$
$$\forall x.\, \forall y.\, x = y \longleftrightarrow y = x$$
$$\forall x.\, \forall y.\, \forall z.\, x = y \wedge y = z \longrightarrow x = z$$

We achieve these properties by axioms for reflexivity and congruence for functions and predicates. We present the equality axioms in Chapter 3.

Chapter 3

Proof Systems

We introduce the proof system HAPS (Harrison's Axiomatic Proof System) that is the kernel of the proof assistant implemented in Harrison's handbook. Furthermore, we discuss its soundness and completeness.

A proof system defines how to construct proofs in the logic, usually by means of rules and axioms. In particular, we will consider an axiomatic proof system that combines a large number of axioms and just two rules of inference to construct proofs.

Given an arbitrary first-order logic formula it is either provable or not in the proof system. It is provable if we can derive the formula by use of the available axioms and rules.

Definition 3.1 5 If a formula A is *provable* in a proof system P, we denote it $\vdash_P A$. The subscript may be dropped when the proof system in question is clear. ◇

3.1 Harrison's Axiomatic Proof System

We now present HAPS (Harrison's Axiomatic Proof System) of first-order logic [Har09, p. 477] in formal notation.

Rules of inference

modus ponens

$$\frac{p \longrightarrow q \qquad p}{q}$$

generalization

$$\frac{p}{\forall x.\, p}$$

Logical axioms

axiom_addimp

$$\frac{}{p \longrightarrow q \longrightarrow p}$$

axiom_distribimp

$$\frac{}{(p \longrightarrow q \longrightarrow r) \longrightarrow (p \longrightarrow q) \longrightarrow p \longrightarrow r}$$

axiom_doubleneg

$$\frac{}{((p \longrightarrow \bot) \longrightarrow \bot) \longrightarrow p}$$

axiom_allimp

$$\frac{}{(\forall x.\, p \longrightarrow q) \longrightarrow (\forall x.\, p) \longrightarrow (\forall x.\, q)}$$

axiom_impall

$$\frac{}{p \longrightarrow (\forall x.\, p)} \quad \neg \text{free\_in } x\ p$$

axiom_existseq

$$\frac{}{\exists x.\, x = t} \quad \neg \text{occurs\_in } x\ t$$

axiom_eqrefl

$$\frac{}{t = t}$$

axiom_funcong

$$\frac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}$$

axiom_predcong

$$\frac{}{s_1 = t_1 \longrightarrow \cdots \longrightarrow s_n = t_n \longrightarrow P(s_1, \ldots, s_n) \longrightarrow P(t_1, \ldots, t_n)}$$

axiom_iffimp1

$$\frac{}{(p \longleftrightarrow q) \longrightarrow p \longrightarrow q}$$

axiom_iffimp2

$$\frac{}{(p \longleftrightarrow q) \longrightarrow q \longrightarrow p}$$

axiom_impiff

$$\frac{}{(p \longrightarrow q) \longrightarrow (q \longrightarrow p) \longrightarrow (p \longleftrightarrow q)}$$

axiom_true

$$\frac{}{\top \longleftrightarrow (\bot \longrightarrow \bot)}$$

axiom_not

$$\frac{}{\neg p \longleftrightarrow (p \longrightarrow \bot)}$$

axiom_and

$$\frac{}{(p \wedge q) \longleftrightarrow ((p \longrightarrow q \longrightarrow \bot) \longrightarrow \bot)}$$

axiom_or

$$\frac{}{(p \vee q) \longleftrightarrow \neg(\neg p \wedge \neg q)}$$

axiom_exists

$$\frac{}{(\exists x.\, p) \longleftrightarrow \neg(\forall x.\, \neg p)}$$

Table 3.1: Harrison's Axiomatic Proof System (HAPS)

Consider below a proof of the formula $p \longrightarrow p$:

Example 3.1 Proof of $p \longrightarrow p$ in HAPS.

1	$p \longrightarrow p \longrightarrow p$	addimp
2	$(p \longrightarrow (p \longrightarrow p) \longrightarrow p) \longrightarrow (p \longrightarrow p \longrightarrow p) \longrightarrow (p \longrightarrow p)$	distribimp
3	$p \longrightarrow (p \longrightarrow p) \longrightarrow p$	addimp
4	$(p \longrightarrow p \longrightarrow p) \longrightarrow (p \longrightarrow p)$	MP 2, 3
5	$p \longrightarrow p$	MP 4, 1

3.2 Soundness and Completeness

We consider here the soundness and completeness of HAPS.

An unsound proof system may derive invalid formulas, whereas a sound proof system does not.

Definition 3.2 A proof system is *sound* if it derives only valid formulas. ◇

We will prove the soundness of HAPS in Chapter 6.

Lastly, we consider completeness.

Definition 3.3 A proof system is *complete* if it derives all valid formulas. ◇

Completeness is desirable, but is a practically useless result without soundness. In Chapter 7 and in Chapter 8 we provide ample practical evidence for the completeness of HAPS despite not providing a mathematical proof.

Chapter 4

Isabelle

This chapter introduces the proof assistant Isabelle. Furthermore, it discusses the LCF-style approach for proof assistants. This chapter also serves as a brief introduction to programming in Isabelle, where we cover the relevant technical aspects of our formalization.

Mathematical proofs may contain mistakes, and software may have hidden bugs. In proof assistants we can computer-check the correctness of mathematical theorems and software. The proof assistant assists by offering guidance in the development formalization and verification. Even proof assistants are themselves subject to verification as they are also just pieces of software.

4.1 LCF-Style Provers and Proof Assistants

In the proof assistants Isabelle and Coq, the user gives input to the proof assistant by specifying a proof document. The proof document is developed and maintained through an IDE (interactive development environment) much like Eclipse. For regular users of Eclipse the user interface will seem fairly familiar. Coq combines higher-order logic and a richly-typed functional programming language. Isabelle is generic, but classical higher-order logic is most commonly

used. A unique feature of the Isabelle Prover IDE is that it offers the possibility to continuously run and check the proof document. Coq and Isabelle are arguably the most used proof assistants today.

In the proof document, we define what is to be proved. How we find the proofs is a different problem altogether. Even more so because we cannot know if a proof even exists. Isabelle features automated proof search by having access to a number of provers. In theory, one could construct a simple prover by brute forcing through all formulas using a proof system. The provers in Isabelle of course use much more sophisticated algorithms to find the proofs. In some cases, it will require a combination of different provers to provide a complete proof, or they may need to be combined with manually defined proof steps. Noteworthy provers used for automation are the SMT solvers CVC4 and Z3, and the first-order resolution provers E, SPASS and Vampire [BP16]. Vampire has long been considered the most successful first-order theorem prover, and has won the world cup in first-order theorem proving CADE ATP System Competition (CASC) twenty-seven times [KV13]. However, recent research by Blanchette et al. suggests that the CVC4 is prover significantly ahead of its competitors [BGK$^+$16].

The key concept of LCF-style (Logic of Computable Functions) is to have an abstract type for theorems (the formulas derived by the proof system) in the implementation. Formulas of this abstract type must not be able to be instantiated outside of the proof system implementation. This requires the implementation language to be strongly typed such that we can guarantee that this property holds. Isabelle is implemented purely in LCF-style, and Coq is to some extent as well.

It is worth mentioning that Coq received the ACM Software System Award in 2013. Other relevant systems that received the award are the Boyer-Moore Theorem Prover, the precursor to ACL2, in 2005, Java in 2002, Apache in 1999, TeX back in 1986, and Unix as the first ever to win the award in 1983. The awards are given to software systems based on the long-term impact of their contributions.

4.2 Isabelle/HOL

Isabelle/HOL is the most commonly used instance of Isabelle, and we will also be using it for our formalization. Isabelle/HOL features higher-order logic and provides tools ideal for formal specification [NPW02]. We will introduce the relevant concepts of Isabelle/HOL by means of examples. It should be noted that

in statements in Isabelle all free variables are implicitly universally quantified, which generally helps to make the expressions more readable. Also, when we refer to Isabelle, we always refer to the Isabelle/HOL instance of Isabelle, and we do not distinguish these two notions.

4.2.1 Theories

Proof developments in Isabelle consist of one or several source files called theory files. At the start of the file, we specify the theory name, which must always match the file name. Also, we state what other theory files should be imported. We generally always import the *Main* theory.

theory *MyTheory*
imports *Main*
begin

end

The data types, definitions, abbreviations, functions and proofs go between the *begin* and *end* commands.

4.2.2 Data Types

The *datatype* command is used to create inductive data types.

Example 4.1 The following command introduces a data type for representation of FOL terms:

datatype *tm* = *Var id* | *Fn id* (*tm list*)

◇

The *tm* type has a constructor *Var* for variables which takes as argument a string (variable symbol). The constructor *Fn* is for function, and is inductively defined, as it takes as arguments a string (function symbol) and a list of terms. The different cases are split by the | operator.

4.2.3 Plain Definitions and Abbreviations

Plain definitions are non-recursive expressions that make it possible to reason about "definitions as names" on a higher level, as they are not unfolded by default. The command *definition* is for plain definitions.

Example 4.2 The following command introduces a plain definition for checking if a list has exactly two elements:

```
definition length2 :: tm list ⇒ bool
where
  length2 l ≡ case l of [-,-] ⇒ True | - ⇒ False
```

◇

Similar to definitions, we also have abbreviations, which can be understood as shorthands for more complex expressions. Unlike definitions they are always unfolded, and can be useful in cases where we want to have a shorthand for a complex expression, but we do not need to reason about it on a higher level.

Example 4.3 The following command introduces an abbreviation for checking if a list consists of exactly two elements:

```
abbreviation (input) length2 l ≡ case l of [-,-] ⇒ True | - ⇒ False
```

◇

Finally, we will also be using inductive definitions. The command *inductive* defines an inductive predicate from the stated introduction rules. Inductive definitions generate only positive information which make them superior to recursive definitions for purposes such as proof systems.

Example 4.4 The following command defines even natural numbers inductively by the constant zero and successor function:

```
inductive even :: nat ⇒ bool
where
  Zero:
    even 0 |
  Next:
    even n ⟹ even (Suc (Suc n))
```

◇

4.2.4 Primitive Recursive Functions

There are different ways to define functions in Isabelle. We will be constraining ourselves to a particular kind of primitive recursive functions, which can be defined by the *primrec* command. This kind of primitive recursion requires us to define an equation for each possible constructor of the data type. Each equation must specify at most one reduction rule for each constructor. For instance, for lists we match head and tail on the left-hand side, and use the tail in the recursive call. Restraining ourselves to such equations ensures that the function is total, which guarantees termination.

Example 4.5 The following command defines the sum of a list of natural numbers by primitive recursion:

primrec *list-sum* :: *nat list* ⇒ *nat*
where
list-sum (*h* # *t*) = *h* + (*list-sum t*) |
list-sum [] = *0*

◇

4.2.5 Proofs

Any of the commands *theorem*, *proposition*, *lemma* and *corollary* can be used to define proofs. The only difference is the name of the command, which indicates to the reader the importance of the proof. Proofs can optionally be named, and thus referenced later. Furthermore, we have to state a HOL formula in Isabelle that is to be proved.

Example 4.6 The following lemma states that the result of *list-sum* is equal to the sum of all elements for a simple list:

lemma *list-sum* [*0*, *Suc*(*0*), *Suc*(*Suc*(*0*))] = *Suc*(*Suc*(*Suc*(*0*)))
apply *simp*
done

The proof uses apply-style, where we subsequently apply proof methods until there are no more subgoals left. In this case, the simplifier *simp* is sufficient. ◇

Isabelle comes with various proof methods. A full understanding of all the built-in proof methods is beyond what we can hope to achieve in this thesis.

Instead, we will briefly and informally describe each of the proof methods we will need for the formalization. For more in-depth explanations, please refer to *The Isabelle/Isar Reference Manual* [W+16].

- *simp* performs simplification on the first subgoal. The simplifier performs rewritings based on information from the theory and proof context, but in some cases extra rules are needed. Simplification does not necessarily solve the subgoal.

- *simp-all* is similar to *simp*, but performs simplification on all subgoals.

- *standard* performs a single refinement step, such as an introduction/elimination rule.

- *fastforce* uses a form of sequent calculus for logic reasoning. This is combined with simplification, and the proof search is conducted using a heuristic depth-first approach.

- *metis* is a purely logical prover that uses resolution to solve the subgoal.

- *iprover* is an intuitionistic prover, which means that it searches for a direct proof using rules taken from the theory or given as arguments.

Even from the descriptions, it is often not clear which proof method can succeed. We can in many cases ask for assistance by using the automated proof search feature. There are different tools for automated proof search available, and they may suggest different solutions. They will not only suggest a proof method to use, but can also use theorems from the theory. The following commands can be invoked at any given proof state, and the response will be printed in the output panel.

- *sledgehammer* uses external provers, namely resolution and SMT solvers. On success, the proof method to be used is given in the output panel.

- *try0* uses standard proof methods, for instance *simp*, and does not use any external provers. It is suitable for sufficiently simple subgoals.

- *try* uses a combination of provers (and disprovers for a classical contradiction proof or counterexample). It uses both *try0* and *sledgehammer* and is the preferred command in most cases.

Example 4.7 The following command invokes the *try* method on the lemma from Example 4.6:

lemma *list-sum* [*0*, *Suc*(*0*), *Suc*(*Suc*(*0*))] = *Suc*(*Suc*(*Suc*(*0*)))
try

Running the command gives the output *Try0 found a proof: by simp (3 ms).* ◇

4.3 Isar

Isar is a structured language for proof development in Isabelle. The main idea is that the language is readable by both humans and computers, and it is preferred over apply-style proofs in Isabelle. We will introduce the relevant aspects of the Isar language by means of examples. While it may not serve as a complete guide to learning Isar, the goal is to provide knowledge sufficient to understand the idea and structure of the proofs in our formalization.

Isar proofs are wrapped by the commands *proof* and *qed* with the content in between. By default the *proof* commands uses the *standard* method on subgoals. To provide a proof of the original statement without alterations, the command *proof* - can be used.

Example 4.8 The lemma from Example 4.6 in Isar:

lemma *list-sum* [*0*, *Suc*(*0*), *Suc*(*Suc*(*0*))] = *Suc*(*Suc*(*Suc*(*0*)))
proof *simp* **qed**

◇

Example 4.9 The lemma from Example 4.6 using the command *by*:

lemma *list-sum* [*0*, *Suc*(*0*), *Suc*(*Suc*(*0*))] = *Suc*(*Suc*(*Suc*(*0*)))
by *simp*

◇

Isar also features forward-chaining proofs where subproofs are combined to prove the theorem. Local goals are stated using the *have* command. This is useful when an intermediate result leads to the final result. The command *show* states and solves an existing subgoal. Writing *then show* carries over the result of the just previously stated local goal. The top level goal can easily be referenced by *?thesis*.

Example 4.10 The following Isar proof uses forward-chaining to prove the theorem:

```
lemma even a ⟹ even (a + a)
proof −
  assume even a
  then have even (a + a) sorry
  then show ?thesis .
qed
```

The command *sorry* can be used when there is no proof yet. ◇

4.3.1 Inductive Reasoning

We consider here a relatively simple example of an induction proof:

Example 4.11 The following Isar proof shows that the sum of two even natural numbers is also even:

```
lemma even a ⟹ even (a + a)
proof −
  assume even a
  then have even (a + a)
  proof (induct rule: even.induct)
  qed (simp add: Zero, simp add: Next)
  then show ?thesis .
qed
```

The proof is by induction on the structure of the inductive definition for *even* shown in Example 4.4. The subgoal produced by the induction are solved by simplification using the rules of *even*. ◇

It is possible to use *induct* also for case proofs, i.e. considering the constructor cases of a data type, when we do not need the induction hypothesis. The command *next* is used to separate the cases, and the command *fix* makes the fixed variables universally quantified in entire proof block:

Example 4.12 The following proof is by cases analysis of the structure of natural numbers:

```
lemma even a ⟹ even (a + a)
proof (induct a)
  assume even 0
  then show even (0 + 0) by simp
next
  fix a
  assume even (Suc a)
  then show even (Suc a + Suc a) sorry
qed
```

◇

4.3.2 Calculational Reasoning

Isabelle features two types of reasoning where we maintain a set of background facts, called the calculation, leading up to a combined result. Firstly, we can use the calculation to perform the proof in a transitive chain. Informally speaking, it is useful when we can prove the goal in a number of smaller steps, but not everything at once. For instance, assume we want to show $a = c$, but we can only show it by first proving $a = b$ followed by $b = c$. In Isar, we achieve this by using the commands *also* and *finally*.

Example 4.13 The following Isar proof uses calculational reasoning to show $a = c$ from $a = b$ and $b = c$:

```
lemma a = c
proof −
  have a = b sorry
  also have ... = c sorry
  finally show ?thesis .
qed
```

The dots ... refer to the right-hand side of the previous equation and is useful for longer expressions. The intermediate equations are proved in a transitive chain using *also*, and the chain is ended by *finally*. Since we end up with a calculation that exactly shows $a = c$, the final goal is solved using simply the dot . which is a shorthand for *by standard*. ◇

We can also choose to simply name all assumptions and subproofs and reference them as needed:

Example 4.14 The following Isar proof uses naming and referencing of subproofs to prove the conjunction from Example 4.13:

lemma $a = b \wedge c = d \wedge e = f$
proof −
  have *1*: $a = b$ **sorry**
  have *2*: $c = d$ **sorry**
  have *3*: $e = f$ **sorry**
  show *?thesis*
  using *1 2 3*
  by *simp*
qed

◇

By using *moreover* to state additional facts for the calculation, and using the facts together by use of the command *ultimately*, we avoid naming the local facts and referencing them.

Example 4.15 The following Isar proof uses calculational reasoning to prove the conjunction from Example 4.13:

lemma $a = b \wedge c = d \wedge e = f$
proof −
  have $a = b$ **sorry**
  moreover have $c = d$ **sorry**
  moreover have $e = f$ **sorry**
  ultimately show *?thesis*
  by *simp*
qed

◇

Chapter 5

Formalization: Syntax, Semantics and Proof System

This chapter formalizes the syntax and semantics of first-order logic in Isabelle. Furthermore, it formalizes the proof system HAPS and shows how to apply the rules and axioms manually in proofs in Isabelle.

The entire formalization is contained in the Isabelle theory Proven which can be found in Appendix A. The content is split into separate sections:

1) Syntax of first-order logic
2) Definition of rules and axioms
3) Code generation for rules and axioms
4) Semantics of first-order logic
5) Definition of proof system
6) Soundness of proof systems
7) Appendix: Mentioned built-in facts

5.1 Syntax

The systematic approach to building terms and formulas is straightforwardly defined in Isabelle by the *datatype* command.

We first start by defining a data type for terms:

datatype *tm* = *Var id* | *Fn id* (*tm list*)

Next, we turn to define the data type for formulas. When possible, we try to follow the style of the code in Harrison's handbook. Therefore, the formulas data type is not defined specifically for first-order logic, but with the possibility of using the same data type for other logics, such as propositional logic. This is achieved by means of an arbitrary type constant $'a$ that determines the type of atoms:

datatype $'a$ *fm* = *T* | *F* | *Atom* $'a$ |
Imp ($'a$ *fm*) ($'a$ *fm*) | *Iff* ($'a$ *fm*) ($'a$ *fm*) |
And ($'a$ *fm*) ($'a$ *fm*) | *Or* ($'a$ *fm*) ($'a$ *fm*) | *Not* ($'a$ *fm*) |
Exists id ($'a$ *fm*) | *Forall id* ($'a$ *fm*)

In first-order logic the atoms are predicates, and in propositional logic the atoms are propositional symbols. The constants *T* and *F* are for truth and falsity, respectively. The single-letter names are used to avoid confusion with the built-in boolean types of Isabelle.

The data type *fol* is defined to obtain FOL formulas with the derived type *fol fm*:

datatype *fol* = *R id* (*tm list*)

It specifies the syntax for the atomic parts of our formulas. An alternative data type for propositional symbols as atoms could easily be implemented to obtain a derived data type for formulas of propositional logic.

The constructor *R* must be used when defining FOL atoms. Consider as an example how we express an equality:

Atom (*R* (*STR* ''=''') [*x*, *y*])

The string ''=''' must be wrapped in the *STR* constructor for the code generator to export it correctly.

5.2 Semantics

The semantics of first-order logic with equality can be defined as functions in Isabelle. Given the variable denotation, function denotation and predicate denotation, we define the semantics recursively following the structure of terms and formulas. Note that we use the letters e, f and g in Isabelle for the denotations.

Following Berghofer [Ber07], we use the type variable $'a$. It reflects the arbitrary type of elements in the universe.

We first define the semantics of terms. We also define the semantics for lists of terms. Such lists occur in functions and predicates. The semantics of a list of terms is simply the list of the semantics of each term. Since the functions are mutually recursive, we must define them simultaneously. The first two arguments of both functions are the variable denotation and function denotation. The predicate denotation is not used for terms:

```
primrec — Semantics of terms
  semantics-term :: (id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm ⇒ 'a and
  semantics-list :: (id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm list ⇒ 'a list
where
  semantics-term e - (Var x) = e x |
  semantics-term e f (Fn i l) = f i (semantics-list e f l) |
  semantics-list - - [] = [] |
  semantics-list e f (t # l) = semantics-term e f t # semantics-list e f l
```

For the case of variables, the usual f has been replaced by - since it is not used. Likewise for the case of the empty list, neither e nor f is used. We see that e and f are considered maps pointing to elements of an arbitrary type $'a$.

We can now define the semantics of first-order formulas. The semantics of formulas also depends on the predicate denotation which is passed as an extra argument:

```
primrec — Semantics of formulas
  semantics :: (id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ bool)
     ⇒ fol fm ⇒ bool
where
  semantics - - - T = True |
  semantics - - - F = False |
  semantics e f g (Atom a) = (case a of R i l ⇒ if i = STR ''='' ∧ length2 l
     then (semantics-term e f (hd l) = semantics-term e f (hd (tl l)))
     else g i (semantics-list e f l)) |
  semantics e f g (Imp p q) = (semantics e f g p ⟶ semantics e f g q) |
```

semantics e f g (Iff p q) = (semantics e f g p ⟷ semantics e f g q) |
semantics e f g (And p q) = (semantics e f g p ∧ semantics e f g q) |
semantics e f g (Or p q) = (semantics e f g p ∨ semantics e f g q) |
semantics e f g (Not p) = (¬ semantics e f g p) |
semantics e f g (Exists x p) = (∃ v. semantics (e(x := v)) f g p) |
semantics e f g (Forall x p) = (∀ v. semantics (e(x := v)) f g p)

The semantics of implication, bi-implication, conjunction, disjunction and negation is defined using Isabelle's higher-order logic operators. By doing so, we are confident that the operators are implemented correctly. The semantics of the existential quantifier is true if there is a value of the quantified variable such that the formula is true. For universal quantifiers the formula must be true for all possible values of the quantified variable. Lastly, the semantics of predicates deserve a special mention, as we hard code equality into the semantics. If we encounter an equality predicate *R ''='' [s, t]*, we evaluate to true if the semantic value of *s* is equal to *t*, and false otherwise. For any other predicate, the semantic value is determined similarly to functions by lookup in the predicate denotation.

5.3 Rules and Inductive Definition

We now cover the formalization of the HAPS proof system. We will first be defining each rule and axiom independently. We then use these rules and axioms as our introduction rules in the inductive predicate that can derive formulas based on HAPS.

We introduce a new type *fol-thm* for first-order formulas derived in our proof system:

datatype *fol-thm* = *Thm* (*concl*: *fol fm*)

We instantiate formulas of this type by the *Thm* constructor, and we can extract the formula with *concl*. Following the style of Harrison's implementation, we would simply introduce the type as a *type-synonym* and avoid the *Thm* constructor. The *type-synonym* command creates an alias for a type, but Isabelle does not generate code for it. We need a data type with a constructor that is hidden outside of the implementation, such that the kernel cannot be bypassed.

5.3.1 Auxiliary Functions

Before we are ready to define our rules and axioms, we need to cover a number of auxiliary functions and definitions that we will be using. While we seek a close resemblance to Harrison's implementation in [Har09], we have reformulated certain parts of the implementation to more easily construct the proofs.

The definition *fol-equal* is used for equality of first-order formulas:

definition *fol-equal* :: *fol fm* ⇒ *fol fm* ⇒ *bool*
where
fol-equal p q ≡ *p* = *q*

For the proofs, the Isabelle built-in equality could be used instead. However, when generating code Isabelle will define its own equality functions in the exported file. Therefore, we use a wrapper to map equality in Isabelle to the built-in equality in SML.

We will later need a function that produces a chain of implications of equalities given two term lists as input. More specifically, given two lists $[u_1, u_2, \ldots, u_n]$ and $[w_1, w_2, \ldots, w_n]$, and a formula p, we want to produce the formula

$$u_1 = w_1 \longrightarrow u_2 = w_2 \longrightarrow \ldots \longrightarrow u_n = w_n \longrightarrow p$$

The equality predicate $=(x, y)$ is simply written as $x = y$ for the sake of simplicity. In Harrison's implementation, this is achieved through a more general function *itlist2*. To more easily carry through our proofs, we will instead use two functions tailored for this exact functionality:

definition *zip-eq* :: *tm list* ⇒ *tm list* ⇒ *fol fm list*
where
zip-eq l l′ ≡ *map* (λ(*t*, *t′*). *Atom* (*R* (*STR* ″=″) [*t*, *t′*])) (*zip l l′*)

primrec *imp-chain* :: *fol fm list* ⇒ *fol fm* ⇒ *fol fm*
where
imp-chain [] *q* = *q* |
imp-chain (*p* # *l*) *q* = *Imp p* (*imp-chain l q*)

The definition *zip-eq* utilizes *zip* and *map* available in Isabelle to return the just mentioned equalities in a list given two lists of terms. The function *imp-chain* then returns the chain of implications given a list of equalities and a formula.

The functions *occurs-in* and *occurs-in-list* checks if a variable symbol occurs in a term and a list of terms, respectively:

primrec
occurs-in :: *id* ⇒ *tm* ⇒ *bool* **and**
occurs-in-list :: *id* ⇒ *tm list* ⇒ *bool*
where
occurs-in i (*Var x*) = (*i* = *x*) |
occurs-in i (*Fn* - *l*) = *occurs-in-list i l* |
occurs-in-list - [] = *False* |
occurs-in-list i (*h* # *t*) = (*occurs-in i h* ∨ *occurs-in-list i t*)

The function *free-in* checks if a variable symbol is free in a given formula. If a variable symbol is free, it occurs in a predicate and is not bound by a quantifier:

primrec *free-in* :: *id* ⇒ *fol fm* ⇒ *bool*
where
free-in - *T* = *False* |
free-in - *F* = *False* |
free-in i (*Atom a*) = (*case a of R* - *l* ⇒ *occurs-in-list i l*) |
free-in i (*Imp p q*) = (*free-in i p* ∨ *free-in i q*) |
free-in i (*Iff p q*) = (*free-in i p* ∨ *free-in i q*) |
free-in i (*And p q*) = (*free-in i p* ∨ *free-in i q*) |
free-in i (*Or p q*) = (*free-in i p* ∨ *free-in i q*) |
free-in i (*Not p*) = *free-in i p* |
free-in i (*Exists x p*) = (*i* ≠ *x* ∧ *free-in i p*) |
free-in i (*Forall x p*) = (*i* ≠ *x* ∧ *free-in i p*)

The function *equal-length* checks if two lists of terms have equal length:

primrec *equal-length* :: *tm list* ⇒ *tm list* ⇒ *bool*
where
equal-length l [] = (*case l of* [] ⇒ *True* | - # - ⇒ *False*) |
equal-length l (- # *r*′) = (*case l of* [] ⇒ *False* | - # *l*′ ⇒ *equal-length l*′ *r*′)

For the soundness proof, *equal-length l r* could be replaced by *length l* = *length r*, using Isabelle's built-in length function for lists. However, this causes Isabelle to generate a data type for natural numbers. Furthermore, since the length function in SML uses integers in contrast to Isabelle which uses natural numbers, the simple solution is to avoid the length function in Isabelle.

Finally, we have also the abbreviation *fail-thm* which is used instead of exceptions in places where a rule or axiom is used incorrectly:

abbreviation (*input*) *fail-thm* ≡ *Thm T*

We need to use a workaround as Isabelle does not feature throwing of exceptions. Since *fail-thm* is used as a return value, it must have the type *fol-thm*. At first glance, it will seem disturbing that the formula ⊤ is returned. However, misuse

of axioms or rules is not particularly interesting from a theoretical perspective. Furthermore, it does not affect which formulas can be proved. The formula $\top$ is itself clearly valid and can be assumed to be provable in HAPS.

Alternatives solutions have also been considered. By defining *fail-thm* as *undefined*, the code generator can be instructed to generate exceptions for those cases. The value *undefined* has an arbitrary type in Isabelle, and can be used anywhere, but makes is harder to state the soundness theorem. Alternatively, the *option* data type is also available with constructors *Some x* or *None*. Ultimately, all alternatives were rejected to avoid introducing further levels of abstraction into the proofs and the generated code.

5.3.2 Rules and Axioms

The formulas derivable by the rules and axioms of HAPS are theorems of FOL given that soundness holds for HAPS. We define the first of our two rules, namely modus ponens:

```
definition modusponens :: fol-thm ⇒ fol-thm ⇒ fol-thm
where
  modusponens s s' ≡ case concl s of Imp p q ⇒
      let p' = concl s' in if fol-equal p p' then Thm q else fail-thm | - ⇒ fail-thm
```

Given premises $p \longrightarrow q$ and p, we derive q. If the first premise is not an implication, or if the second premise is a formula different from p, *fail-thm* is returned. Notice that the type of the premises must be *fol-thm*. This means that we can only use modus ponens for derived theorems and not arbitrary formulas.

Furthermore, we define the generalization rule:

```
definition gen :: id ⇒ fol-thm ⇒ fol-thm
where
  gen x s ≡ Thm (Forall x (concl s))
```

The rule states that we can always enclose a derived theorem in a universal quantifier.

As a result of having only two rules, we have a large number of axioms. We cover here only some of the most interesting axioms. The code for all of the axioms can be found in Appendix A.

We start by defining *axiom-impall*:

definition *axiom-impall* :: *id* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-impall x p ≡ *if* ¬ *free-in x p then Thm* (*Imp p* (*Forall x p*)) *else fail-thm*

The axiom states that if variable x is not free in p then the truth of p implies the truth of $\forall x.\, p$. That x is not free in p means that it only occurs as a bound by a quantifier, or that it does not occur in the formula at all.

Furthermore, we define *axiom-funcong*:

definition *axiom-funcong* :: *id* ⇒ *tm list* ⇒ *tm list* ⇒ *fol-thm*
where
axiom-funcong i l l′ ≡ *if equal-length l l′*
then Thm (*imp-chain* (*zip-eq l l′*) (*Atom* (*R* (*STR ″=″*) [*Fn i l*, *Fn i l′*])))
else fail-thm

The axiom states that if two lists are equal, then applying the first list as arguments to a function is equal to applying the second list. We have instantiated the axiom with two lists that have the same number of elements, as the whole formula becomes true if the lists are not equal.

The definition of *axiom-predcong* is analogous to *axiom-funcong* and the differences should be self-explanatory:

definition *axiom-predcong* :: *id* ⇒ *tm list* ⇒ *tm list* ⇒ *fol-thm*
where
axiom-predcong i l l′ ≡ *if equal-length l l′*
then Thm (*imp-chain* (*zip-eq l l′*) (*Imp* (*Atom* (*R i l*)) (*Atom* (*R i l′*))))
else fail-thm

5.3.3 Inductive Predicate

After defining the rules and axioms of HAPS, we collect them as introduction rules in an inductive predicate *OK*:

inductive *OK* :: *fol fm* ⇒ *bool* (⊢ - *0*)
where
modusponens:
⊢ *concl s* ⟹ ⊢ *concl s′* ⟹ ⊢ *concl* (*modusponens s s′*) |
gen:
⊢ *concl s* ⟹ ⊢ *concl* (*gen* - *s*) |
axiom-addimp:
⊢ *concl* (*axiom-addimp* - -) |
axiom-distribimp:
⊢ *concl* (*axiom-distribimp* - - -) |
axiom-doubleneg:
⊢ *concl* (*axiom-doubleneg* -) |
axiom-allimp:
⊢ *concl* (*axiom-allimp* - - -) |
axiom-impall:
⊢ *concl* (*axiom-impall* - -) |
axiom-existseq:
⊢ *concl* (*axiom-existseq* - -) |
axiom-eqrefl:
⊢ *concl* (*axiom-eqrefl* -) |
axiom-funcong:
⊢ *concl* (*axiom-funcong* - - -) |
axiom-predcong:
⊢ *concl* (*axiom-predcong* - - -) |
axiom-iffimp1:
⊢ *concl* (*axiom-iffimp1* - -) |
axiom-iffimp2:
⊢ *concl* (*axiom-iffimp2* - -) |
axiom-impiff:
⊢ *concl* (*axiom-impiff* - -) |
axiom-true:
⊢ *concl axiom-true* |
axiom-not:
⊢ *concl* (*axiom-not* -) |
axiom-and:
⊢ *concl* (*axiom-and* - -) |
axiom-or:
⊢ *concl* (*axiom-or* - -) |
axiom-exists:
⊢ *concl* (*axiom-exists* - -)

The symbol ⊢ can be used instead of *OK* and the outermost parentheses can be omitted, i.e. *OK (Imp p q)* can be written as ⊢ *Imp p q*. The value of a statement ⊢ *p* is true if *p* can be proved, and false otherwise. The premises of *gen* and *modusponens* are stated explicitly by use of the meta-implication ⟹. The preconditions of the axioms are not stated explicitly, as the formula ⊤ is returned in case of an exception.

The following example shows a proof with $\vdash$ by use of the rules and axioms.

Example 5.1 The following is a proof in Isabelle of Example 3.1:

```
corollary ⊢ Imp p p
proof −
  have 1: ⊢ concl (Thm (Imp (Imp p (Imp (Imp p p) p))
     (Imp (Imp p (Imp p p)) (Imp p p))))
  using axiom-distribimp
  unfolding axiom-distribimp-def
  by simp

  have 2: ⊢ concl (Thm (Imp p (Imp (Imp p p) p)))
  using axiom-addimp
  unfolding axiom-addimp-def
  by simp

  have 3: ⊢ concl (Thm (Imp (Imp p (Imp p p)) (Imp p p)))
  using 1 2 modusponens
  unfolding modusponens-def fol-equal-def
  by fastforce

  have 4: ⊢ concl (Thm (Imp p (Imp p p)))
  using axiom-addimp
  unfolding axiom-addimp-def
  by simp

  have 5: ⊢ concl (Thm (Imp p p))
  using 3 4 modusponens
  unfolding modusponens-def fol-equal-def
  by fastforce

  show ?thesis
  using 5
  by simp
qed
```

◇

Chapter 6

Formalization: Proof of Soundness

This chapter presents a formalization of a proof of soundness for HAPS in Isabelle. We present all parts of the proof including some helpful lemmas from Isabelle.

By proving the soundness of our inductive predicate $\vdash$, we also prove the soundness of HAPS. This is of course assuming that we translated the logic correctly into Isabelle. Recall soundness formulated as:

$$\vdash p \text{ implies } p \text{ true in variable, function and predicate denotations}$$

We only consider valid formulas. That is, formulas that can be derived from no assumptions. Assumptions can still be introduced by means of implications in the formula p. This is also reflected in the type of $\vdash$ as the only argument is the formula in consideration. In Isabelle, we formulate the soundness property as:

theorem *soundness*:
$\vdash p \Longrightarrow$ *semantics e f g p*

Here, the free variables e, f and g are the variable, function and predicate denotations, respectively.

6.1 Useful Built-in Theorems

Isabelle has numerous built-in theorems, readily available for us to use in our proofs which we listhere. Please note that the *simp* also has access to a number theorems as rewrite rules, even without explicitly adding then to the command.

- *fun-upd-twist* shows that the order of two map updates is interchangeable when the keys updated are different.
  $x \neq x' \Longrightarrow e(x := v,\, x' := v') = e(x' := v',\, x := v)$
- *fun-upd-upd* shows that two consecutive assignments to the same key renders the first assignment irrelevant.
  $e(x := v,\, x := v') = e(x := v')$
- *iff* shows that bi-implication can be expressed by means of a conjunction of implications (one for each direction).
  $(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow P \longleftrightarrow Q$
- *imp-conjL* shows the equivalence of two nested implications and conjunction on the left-hand side of an implication.
  $(P \wedge P' \longrightarrow Q) \longleftrightarrow (P \longrightarrow P' \longrightarrow Q)$
- *list.case(1)* shows that only the empty list case is used when the list is empty.
  $(case\ []\ of\ [] \Rightarrow P \mid h' \#\ t' \Rightarrow P'\ h'\ t') = P$
- *list.case(2)* shows that only the head/tail case is used when the list is not empty.
  $(case\ h \#\ t\ of\ [] \Rightarrow P \mid h' \#\ t' \Rightarrow P'\ h'\ t') = P'\ h\ t$
- *list.case-eq-if* shows that case split on lists can be achieved by means of if-then-else combined with head and tail functions.
  $(case\ l\ of\ [] \Rightarrow P \mid h \#\ t \Rightarrow P'\ h\ t) = (if\ l = []\ then\ P\ else\ P'\ (hd\ l)\ (tl\ l))$
- *list.collapse* shows that head and tail always exist for a non-empty list.
  $l \neq [] \Longrightarrow hd\ l \#\ tl\ l = l$
- *list.exhaust-sel* shows that if both cases of a list case split has the same result, then the case split can be omitted.
  $(l = [] \Longrightarrow P) \Longrightarrow (l = hd\ l \#\ tl\ l \Longrightarrow P) \Longrightarrow P$
- *list.inject* shows that if two lists are equal, then so are the head and tail or each list.
  $h \#\ t = h' \#\ t' \longleftrightarrow h = h' \wedge t = t'$

6.2 Preliminary Lemmas

We introduce here some preliminary lemmas that lead up to the soundness proof. For many of the lemmas, we realized that they were necessary during the proof construction process.

The following lemma shows that if a variable does not occur in a term, updating the variable denotation for that variable does not change the semantics of the term:

```
lemma map':
  ¬ occurs-in x t ⟹ semantics-term e f t = semantics-term (e(x := v)) f t
  ¬ occurs-in-list x l ⟹ semantics-list e f l = semantics-list (e(x := v)) f l
by (induct t and l rule: semantics-term.induct semantics-list.induct) simp-all
```

This also holds for lists of terms. Since the semantics of terms and lists are mutually dependent, the proof is carried out by simultaneous inductions.

The next lemma generalizes *map′* to formulas. Therefore, we now consider if a variable is free in a formula. If it is not, then updating the variable denotation for that variable does not change the semantics of the formula. The proof is by induction on formulas. The entire proof code can be seen in Appendix A.

```
lemma map:
  ¬ free-in x p ⟹ semantics e f g p ⟷ semantics (e(x := v)) f g p
proof (induct p arbitrary: e)
```

The cases for truth and falsity are trivially proved by simplification:

```
  fix e
  show ¬ free-in x T ⟹ semantics e f g T ⟷ semantics (e(x := v)) f g T
  by simp
next
  fix e
  show ¬ free-in x F ⟹ semantics e f g F ⟷ semantics (e(x := v)) f g F
  by simp
```

For predicates the proof is split into two cases, one for the equality predicate and one for other predicates. The lemma *map′* is used for arguments (terms) of the predicate:

```
fix a e
show ¬ free-in x (Atom a) ⟹
    semantics e f g (Atom a) ⟷
    semantics (e(x := v)) f g (Atom a)
proof (induct a)
  fix i l
  assume ¬ free-in x (Atom (R i l))
  then have fresh: ¬ occurs-in-list x l
  by simp
  show semantics e f g (Atom (R i l)) ⟷
      semantics (e(x := v)) f g (Atom (R i l))
  proof cases
    assume eq: i = STR ''='' ∧ length2 l
    then have semantics e f g (Atom (R i l)) ⟷
        semantics-term e f (hd l) =
        semantics-term e f (hd (tl l))
    by simp
    also have ... ⟷
        semantics-term (e(x := v)) f (hd l) =
        semantics-term (e(x := v)) f (hd (tl l))
    using map′(1) fresh occurs-in-list.simps eq list.case-eq-if list.collapse
    unfolding length2-def
    by metis
    finally show ?thesis
    using eq
    by simp
  next
    assume not-eq: ¬ (i = STR ''='' ∧ length2 l)
    then have semantics e f g (Atom (R i l)) ⟷ g i (semantics-list e f l)
    by simp iprover
    also have ... ⟷ g i (semantics-list (e(x := v)) f l)
    using map′(2) fresh
    by metis
    finally show ?thesis
    using not-eq
    by simp iprover
  qed
qed
```

The cases for the propositional logical connectives are analogous. The connectives *Or* and *Not* are shown here:

```
  fix p1 p2 e
  assume assm1: ¬ free-in x p1 ⟹
      semantics e f g p1 ⟷
      semantics (e(x := v)) f g p1 for e
  assume assm2: ¬ free-in x p2 ⟹
      semantics e f g p2 ⟷
      semantics (e(x := v)) f g p2 for e
  show ¬ free-in x (Or p1 p2) ⟹
      semantics e f g (Or p1 p2) ⟷
      semantics (e(x := v)) f g (Or p1 p2)
  using assm1 assm2
  by simp
next
  fix p e
  assume ¬ free-in x p ⟹
      semantics e f g p ⟷ semantics (e(x := v)) f g p for e
  then show ¬ free-in x (Not p) ⟹
      semantics e f g (Not p) ⟷ semantics (e(x := v)) f g (Not p)
  by simp
```

The cases for quantifiers are proved by *metis* with the lemmas *fun-upd-twist* and *fun-upd-upd* following simplification:

```
  fix x1 p e
  assume ¬ free-in x p ⟹
      semantics e f g p ⟷
      semantics (e(x := v)) f g p for e
  then show ¬ free-in x (Exists x1 p) ⟹
      semantics e f g (Exists x1 p) ⟷
      semantics (e(x := v)) f g (Exists x1 p)
  by simp (metis fun-upd-twist fun-upd-upd)
next
  fix x1 p e
  assume ¬ free-in x p ⟹
      semantics e f g p ⟷
      semantics (e(x := v)) f g p for e
  then show ¬ free-in x (Forall x1 p) ⟹
      semantics e f g (Forall x1 p) ⟷
      semantics (e(x := v)) f g (Forall x1 p)
  by simp (metis fun-upd-twist fun-upd-upd)
```

For our next lemma, we want to show that a list of length two has exactly two elements, namely the head of the list followed by the head of the tail:

```
lemma length2-equiv:
  length2 l ⟷ [hd l, hd (tl l)] = l
proof −
  have length2 l ⟹ [hd l, hd (tl l)] = l
  unfolding length2-def
  using list.case-eq-if list.exhaust-sel
  by metis
  then show ?thesis
  unfolding length2-def
  using list.case list.case-eq-if
  by metis
qed
```

Next, we show the symmetry of the definition for *equal-length*. That is, the order of the arguments is interchangeable. Induction on lists is required to prove the lemma:

```
lemma equal-length-sym:
  equal-length l l′ ⟹ equal-length l′ l
proof (induct l′ arbitrary: l)
  fix l
  assume equal-length l []
  then show equal-length [] l
  using equal-length.simps list.case-eq-if
  by metis
next
  fix l l′ a
  assume sym: equal-length l l′ ⟹ equal-length l′ l for l
  assume equal-length l (a # l′)
  then show equal-length (a # l′) l
  using equal-length.simps list.case-eq-if list.collapse list.inject sym
  by metis
qed
```

The following lemma shows that if two lists have equal length, then if one has length two then so does the other. The result seems simple, but it is not obvious for Isabelle without an explicit proof:

```
lemma equal-length2:
  equal-length l l′ ⟹ length2 l ⟷ length2 l′
proof −
  assume assm: equal-length l l′
  have equal-length l [t, t′] ⟹ length2 l for t t′
  unfolding length2-def
  using equal-length.simps list.case-eq-if
  by metis
  moreover have equal-length [t, t′] l′ ⟹ length2 l′ for t t′
```

```
  unfolding length2-def
  using equal-length.simps list.case-eq-if equal-length-sym
  by metis
  ultimately show ?thesis
  using assm length2-equiv
  by metis
qed
```

We now show a property about implication chains (nested implications $q_1 \longrightarrow q_2 \longrightarrow \ldots \longrightarrow p$). The implication chain is true unless $q_1, q_2, \ldots$ are all true while p is false. By induction, simplification solves all subgoals with the use of the lemma *imp-conjL*:

```
lemma imp-chain-equiv:
  semantics e f g (imp-chain l p) ⟷
      (∀ q ∈ set l. semantics e f g q) ⟶ semantics e f g p
using imp-conjL
by (induct l) simp-all
```

Next, we show this for an implication chain of equalities ($s_1 = t_1 \longrightarrow s_2 = t_2 \longrightarrow \ldots \longrightarrow p$). This proof is by simultaneous induction on two lists, and is finalized by the intuitionistic prover:

```
lemma imp-chain-zip-eq:
  equal-length l l' ⟹
      semantics e f g (imp-chain (zip-eq l l') p) ⟷
      semantics-list e f l = semantics-list e f l' ⟶ semantics e f g p
proof −
  assume equal-length l l'
  then have (∀ q ∈ set (zip-eq l l'). semantics e f g q) ⟷
      semantics-list e f l = semantics-list e f l'
  unfolding zip-eq-def
  using length2-def
  by (induct l l' rule: list-induct2') simp-all
  then show ?thesis
  using imp-chain-equiv
  by iprover
qed
```

The lemma *funcong* uses the lemmas about implication chains to the prove the axiom *axiom-funcong*, for the case where the two lists are of equal length:

```
lemma funcong:
  equal-length l l' ⟹
      semantics e f g (imp-chain (zip-eq l l')
          (Atom (R (STR ''='') [Fn i l, Fn i l'])))
proof −
  assume assm: equal-length l l'
  show ?thesis
  proof cases
    assume semantics-list e f l = semantics-list e f l'
    then have semantics e f g (Atom (R (STR ''='') [Fn i l, Fn i l']))
    using length2-def
    by simp
    then show ?thesis
    using imp-chain-equiv
    by iprover
  next
    assume semantics-list e f l ≠ semantics-list e f l'
    then show ?thesis
    using assm imp-chain-zip-eq
    by iprover
  qed
qed
```

The proof is by case analysis. If one of the equalities does not hold, it is trivially shown by use of *imp-chain-zip-eq*. However, if they all hold, we show that the equality of the functions must hold since their arguments have the same semantics.

The lemma *predcong* is similar to *funcong*, but for predicates. Again, we consider the two cases of predicates, namely equalities and all other predicates:

```
lemma predcong:
  equal-length l l' ⟹
      semantics e f g (imp-chain (zip-eq l l') (Imp (Atom (R i l)) (Atom (R i l'))))
proof −
  assume assm: equal-length l l'
  show ?thesis
  proof cases
```

The case for the equality predicate requires us to provide Isabelle more assistance by proving some intermediate results:

```
assume eq: i = STR ''='' ∧ length2 l ∧ length2 l'
show ?thesis
proof cases
  assume semantics-list e f l = semantics-list e f l'
  then have semantics-list e f [hd l, hd (tl l)] =
      semantics-list e f [hd l', hd (tl l')]
  using eq length2-equiv
  by simp
  then have semantics e f g (Imp (Atom (R (STR ''='') l))
      (Atom (R (STR ''='') l')))
  using eq
  by simp
  then show ?thesis
  using eq imp-chain-equiv
  by iprover
next
  assume semantics-list e f l ≠ semantics-list e f l'
  then show ?thesis
  using assm imp-chain-zip-eq
  by iprover
qed
```

The case for non-equality predicates is easily solved as the complex semantics of equality predicates can be ignored:

```
assume not-eq: ¬ (i = STR ''='' ∧ length2 l ∧ length2 l')
show ?thesis
proof cases
  assume semantics-list e f l = semantics-list e f l'
  then have semantics e f g (Imp (Atom (R i l)) (Atom (R i l')))
  using assm not-eq equal-length2
  by simp iprover
  then show ?thesis
  using imp-chain-equiv
  by iprover
next
  assume semantics-list e f l ≠ semantics-list e f l'
  then show ?thesis
  using assm imp-chain-zip-eq
  by iprover
qed
qed
```

The case where the equalities does not hold is analogous for both cases of predicates and rather trivial.

6.3 The Soundness Proof

We are now ready to prove the soundness of the inductive predicate $\vdash$. The proof is by induction on $\vdash$, which means that we prove the soundness for each rule and axiom of HAPS. The full proof can be found in Appendix A.

```
theorem soundness:
  ⊢ p ⟹ semantics e f g p
proof (induct arbitrary: e rule: OK.induct)
```

Proving the modus ponens rules requires two case proofs, and invoking the simplifier on all subgoals, while the generalization rule is proved merely by simplifications:

```
  fix e s s'
  assume semantics e f g (concl s) semantics e f g (concl s') for e
  then show semantics e f g (concl (modusponens s s'))
  unfolding modusponens-def
  proof (induct s)
    fix r
    assume semantics e f g r semantics e f g (concl s') for e
    then show semantics e f g (concl (case r of Imp p q ⇒
      let p' = concl s' in if fol-equal p p' then Thm q else fail-thm | - ⇒ fail-thm))
    unfolding fol-equal-def
    by (induct r) simp-all
  qed
next
  fix e x s
  assume semantics e f g (concl s) for e
  then show semantics e f g (concl (gen x s))
  unfolding gen-def
  by simp
```

In the proof for the modus ponens rule, both s and s' are of the type *fol-thm* which has just a single constructor *Thm*. After unfolding s the proof is conducted by structural induction on formulas.

Most of the of the axioms are trivial and can be proved sound by the simplifier. We show here only the case for *axiom-addimp*:

```
  fix e p q
  show semantics e f g (concl (axiom-addimp p q))
  unfolding axiom-addimp-def
  by simp
```

The axiom *axiom-impall* uses *map* and *axiom-existseq* uses *map′* due to their preconditions. Also here, invoking the simplifier followed by the intuitionistic prover is sufficient:

```
  fix e x p
  show semantics e f g (concl (axiom-impall x p))
  unfolding axiom-impall-def
  using map
  by simp iprover
next
  fix e x t
  show semantics e f g (concl (axiom-existseq x t))
  unfolding axiom-existseq-def
  using map′(1) length2-def
  by simp iprover
```

The cases for *axiom-funcong* and *axiom-predcong* are almost solved entirely by the lemmas *funcong* and *predcong*, respectively. This is evident as the proofs are completed by *standard* which is very basic:

```
  fix e i l l′
  show semantics e f g (concl (axiom-funcong i l l′))
  unfolding axiom-funcong-def
  using funcong
  by simp standard
next
  fix e i l l′
  show semantics e f g (concl (axiom-predcong i l l′))
  unfolding axiom-predcong-def
  using predcong
  by simp standard
```

The axiom *axiom-impiff* also deserves a special mention as it can be proved sound by the simplifier by providing only the lemma *iff*. This is due to the close relation between bi-implication, implication and equality in Isabelle:

```
  fix e p q
  show semantics e f g (concl (axiom-impiff p q))
  unfolding axiom-impiff-def
  by simp (rule iff)
```

Chapter 7

Code Generation

This chapter covers the SML code generation from the Isabelle formalization, including target language modifications and result comparisons using a test suite.

There are a number of target language modifications required to use the generated code with the existing SML version. Also, the generated file itself deserves some explanation. To gain confidence in our formalization, we compare the results between running the test suite in the OCaml version, the SML version and the SML version with the generated kernel.

7.1 Target Language Setup

In principle, code generation from Isabelle does not require any setup, as code is generated for all needed data types and functions. However, in this particular setup, we want to generated code to hook into the existing SML code base by Schlichtkrull and Villadsen [SV]. There already exist data types for terms, formulas, etc., that we want to use. We utilize the *code-printing* command to specify how certain data types, their constructors and constants should be printed in the target language. We note that it is possible to use *code-printing* to generate

code that functions differently from that of the formalization. Therefore, it is essential to show caution when using this command.

We want our data type *tm* to be printed as the existing data type in SML *term*. The constructors are similar:

```
code-printing type-constructor tm ⇀ (SML) term |
  constant Var ⇀ (SML) Var - |
  constant Fn ⇀ (SML) Fn (-, -)
```

The data type *fm* is printed as *formula*. Note that the constants for truth and falsity are different:

```
code-printing type-constructor fm ⇀ (SML) - formula |
  constant T ⇀ (SML) True |
  constant F ⇀ (SML) False |
  constant Atom ⇀ (SML) Atom - |
  constant Imp ⇀ (SML) Imp (-, -) |
  constant Iff ⇀ (SML) Iff (-, -) |
  constant And ⇀ (SML) And (-, -) |
  constant Or ⇀ (SML) Or (-, -) |
  constant Not ⇀ (SML) Not - |
  constant Exists ⇀ (SML) Exists (-, -) |
  constant Forall ⇀ (SML) Forall (-, -)
```

The data type for *fol*, which specify the atoms in formulas:

```
code-printing type-constructor fol ⇀ (SML) fol |
  constant R ⇀ (SML) R (-, -)
```

By default Isabelle generates abstract code for the equality operator, and implements it for each data type it is used on. In our case, we need to check the equality of two formulas for the modus ponens rule. We map this equality check to the built-in equality of SML:

```
code-printing — More efficient
  constant fol-equal ⇀ (SML) - = -
```

In Section 7.4 we compare the efficiency of the built-in equality in SML to the otherwise generated equality function.

7.2 The Generated SML File

We generate the SML code with the *export-code* command. We provide the names of the definitions for all rules and axioms, and combine these into a module *Proven*:

```
export-code
  modusponens gen axiom-addimp axiom-distribimp axiom-doubleneg
  axiom-allimp axiom-impall axiom-existseq axiom-eqrefl
  axiom-funcong axiom-predcong axiom-iffimp1 axiom-iffimp2 axiom-impiff
  axiom-true axiom-not axiom-and axiom-or axiom-exists concl
in SML module-name Proven file Proven.sml
```

The generated code is exported to the file *Proven.sml*.

The following signature is generated for the module which is comparable to the signature of the original version:

```
structure Proven : sig
  type nibble
  type fol_thm
  val concl : fol_thm -> fol formula
  val gen : string -> fol_thm -> fol_thm
  val axiom_or : fol formula -> fol formula -> fol_thm
  val axiom_and : fol formula -> fol formula -> fol_thm
  val axiom_not : fol formula -> fol_thm
  val axiom_true : fol_thm
  val modusponens : fol_thm -> fol_thm -> fol_thm
  val axiom_addimp : fol formula -> fol formula -> fol_thm
  val axiom_allimp : string -> fol formula -> fol formula -> fol_thm
  val axiom_eqrefl : term -> fol_thm
  val axiom_exists : string -> fol formula -> fol_thm
  val axiom_impall : string -> fol formula -> fol_thm
  val axiom_impiff : fol formula -> fol formula -> fol_thm
  val axiom_funcong : string -> term list -> term list -> fol_thm
  val axiom_iffimp1 : fol formula -> fol formula -> fol_thm
  val axiom_iffimp2 : fol formula -> fol formula -> fol_thm
  val axiom_existseq : string -> term -> fol_thm
  val axiom_predcong : string -> term list -> term list -> fol_thm
  val axiom_doubleneg : fol formula -> fol_thm
  val axiom_distribimp : fol formula -> fol formula
                         -> fol formula -> fol_thm
end = struct
```

The code generator exports a data type *nibble* which is not used. This is due to fact that we instantiate the string "=" for the equality predicate. Despite

importing *Code-Char* to properly export strings in the target language, this leftover from the code generator cannot be removed:

```
datatype nibble = Nibble0 | Nibble1 | Nibble2 | Nibble3 | Nibble4 |
  Nibble5 | Nibble6 | Nibble7 | Nibble8 | Nibble9 |
  NibbleA | NibbleB | NibbleC | NibbleD | NibbleE | NibbleF;
```

The auxiliary function *zip-eq* utilizes the built-in *map* and *zip* function in Isabelle. The code for these auxiliary functions is automatically generated:

```
fun zip (x :: xs) (y :: ys) = (x, y) :: zip xs ys
  | zip xs [] = []
  | zip [] ys = [];

fun map f [] = []
  | map f (x21 :: x22) = f x21 :: map f x22;

fun zip_eq la l = map (fn (t, ta) => Atom (R ("=", [t, ta]))) (zip la l);
```

Following LCF-style, the module has its own data type for derived formulas. We also get code for the function that extracts the formula:

```
datatype fol_thm = Thm of fol formula;

fun concl (Thm x) = x;
```

There is also generated code for the remaining auxiliary functions:

```
fun occurs_in_list uv [] = false
  | occurs_in_list i (h :: t) = occurs_in i h orelse occurs_in_list i t
and occurs_in i (Var x) = ((i : string) = x)
  | occurs_in i (Fn (uu, l)) = occurs_in_list i l;

fun free_in uu True = false
  | free_in uv False = false
  | free_in i (Atom a) = let
                           val R (_, aa) = a;
                         in
                           occurs_in_list i aa
                         end
  | free_in i (Imp (p, q)) = free_in i p orelse free_in i q
  | free_in i (Iff (p, q)) = free_in i p orelse free_in i q
  | free_in i (And (p, q)) = free_in i p orelse free_in i q
  | free_in i (Or (p, q)) = free_in i p orelse free_in i q
  | free_in i (Not p) = free_in i p
  | free_in i (Exists (x, p)) = not ((i : string) = x) andalso free_in i p
  | free_in i (Forall (x, p)) = not ((i : string) = x) andalso free_in i p;
```

```
fun equal_length l [] = (case l of [] => true | _ :: _ => false)
  | equal_length l (uu :: r) =
    (case l of [] => false | _ :: la => equal_length la r);

fun imp_chain [] q = q
  | imp_chain (p :: l) q = Imp (p, (imp_chain l q));
```

The generated code for generalization is comparable to the original version. The exceptions from the original version of the modus ponens rule has been replaced by *Thm True* for the cases where *concl sa* is not an implication:

```
fun gen x s = Thm (Forall (x, (concl s)));

fun modusponens sa s =
  (case concl sa of True => Thm True | False => Thm True
    | Atom _ => Thm True
    | Imp (p, q) => let
                      val pa = concl s;
                    in
                      (if p = pa then Thm q else Thm True)
                    end
    | Iff (_, _) => Thm True | And (_, _) => Thm True
    | Or (_, _) => Thm True | Not _ => Thm True
    | Exists (_, _) => Thm True | Forall (_, _) => Thm True);
```

We are now only missing the axioms which are all comparable, except in *axiom_funcong* and *axiom_predcong* where the function *itlist2* from the original version has been replaced by *imp_chain* and *zip_eq*:

```
fun axiom_addimp p q = Thm (Imp (p, (Imp (q, p))));

fun axiom_distribimp p q r =
  Thm (Imp ((Imp (p, (Imp (q, r)))), (Imp ((Imp (p, q)), (Imp (p, r))))));

fun axiom_doubleneg p = Thm (Imp ((Imp ((Imp (p, False)), False)), p));

fun axiom_allimp x p q =
  Thm (Imp ((Forall (x, (Imp (p, q)))),
        (Imp ((Forall (x, p)), (Forall (x, q))))));

fun axiom_impall x p =
  (if not (free_in x p) then Thm (Imp (p, (Forall (x, p)))) else Thm True);
```

```
fun axiom_existseq x t =
  (if not (occurs_in x t)
    then Thm (Exists (x, (Atom (R ("=", [Var x, t])))))
    else Thm True);

fun axiom_eqrefl t = Thm (Atom (R ("=", [t, t])));

fun axiom_funcong i la l =
  (if equal_length la l
    then Thm (imp_chain (zip_eq la l)
                (Atom (R ("=", [Fn (i, la), Fn (i, l)]))))
    else Thm True);

fun axiom_predcong i la l =
  (if equal_length la l
    then Thm (imp_chain (zip_eq la l)
                (Imp ((Atom (R (i, la))), (Atom (R (i, l))))))
    else Thm True);

fun axiom_iffimp1 p q = Thm (Imp ((Iff (p, q)), (Imp (p, q))));

fun axiom_iffimp2 p q = Thm (Imp ((Iff (p, q)), (Imp (q, p))));

fun axiom_impiff p q =
  Thm (Imp ((Imp (p, q)), (Imp ((Imp (q, p)), (Iff (p, q))))));

val axiom_true : fol_thm = Thm (Iff (True, (Imp (False, False))));

fun axiom_not p = Thm (Iff ((Not p), (Imp (p, False))));

fun axiom_or p q =
  Thm (Iff ((Or (p, q)), (Not (And ((Not p), (Not q))))));

fun axiom_and p q =
  Thm (Iff ((And (p, q)),
        (Imp ((Imp (p, (Imp (q, False)))), False))));

fun axiom_exists x p = Thm (Iff ((Exists (x, p)),
                                 (Not (Forall (x, (Not p)))))));

end; (*struct Proven*)
```

7.3 Embedding into Existing Code

There are a few number of modifications necessary to be able to run the generated kernel with the existing SML version. Firstly, we note that the generated file *Proven.sml* uses transparent ascription in the signature. This means that objects of the type *fol_thm* introduced in the module can be instantiated even outside of the module. To comply with LCF-style such objects can only be instantiated inside the module. This ensures that only formulas derived by the proof system are of this type. Therefore, we change the transparent ascription to an opaque ascription which guarantees this property. The line *structure Proven : sig* is modified to *structure Proven :> sig*.

In the original SML version, the prover is initialized by *init.sml* or *init_nj.sml* for compiling on Moscow ML and SML/NJ, respectively. The file *init_nj.sml* is just a wrapper that uses *init.sml*. Therefore, we only modify the version of *init.sml* such that it uses the generated kernel. The line *use "lcf.sml"* is replaced by:

```
use "Proven-lcf.sml";

open Proven;

fun print_thm_aux th = (
    open_box 0;
    print_string "|- "; print_space();
    open_box 0; print_formula_aux print_atom_aux (concl th); close_box();
    close_box()
);

fun print_thm th = (print_thm_aux th; print_flush ());
```

The original contents of *lcf.sml* also contains code for opening the module along with two auxiliary function definitions. As these are not part of the generated code, we include these lines directly in the modified version.

We define a new type *thm* as an alias for the type *fol_thm* from *Proven*. The original version uses the type name *thm* which we avoided in Isabelle due to it being a reserved keyword.

```
type thm = fol_thm;
```

The SML/NJ version *init_nj.sml* is modified to use the generated file instead of *init.sml*.

7.4 Tests

Now that we have successfully embedded the generated file into the existing code, we want to test if the modified version of the prover produces the same results. The original SML version comes with a test suite in the file *full_test.sml*. The file also has an OCaml version that was used to compare the results of the SML version to Harrison's OCaml version. The file collects the tests that Harrison included in the OCaml code, as well as adding some new ones, and is rather exhaustive. Therefore, if the results of the generated version are the same as the results of SML and OCaml versions, we are confident that the generated kernel can be safely substituted to obtain a verified kernel.

The full test suite can also be run in Isabelle in the built-in SML environment. Opening the file *Init.thy* in Isabelle loads the prover and runs the full test for the original version. The following line loads the kernels:

```
SML_file "lcf.sml"
```

We replace it by the generated kernel (we give the modified files the prefix *Proven*):

```
SML_file "Proven-lcf.sml"
SML_file "Proven-init.sml"
```

For technical reasons, we must load *Proven-init.sml* after loading the generated kernel.

All tests are performed on Windows 10 with an Intel i7-4790k 4.0GHz CPU and 16 GB DDR3 ram. All the executions were also timed. There are two timing for each run, one where we time entire execution and one where we time only the test suite. The code also contain examples scattered across the source files which makes it relevant to disregard the initialization of source files for comparisons.

We compared the output of running the test suite on Harrison's OCaml version, Schlichtkrull's and Villadsen's SML version, and the generated version. We found that the results of all three versions were identical.

	Original	Generated	Generated (built-in equality)
Moscow ML	13.7s (6.7s)	49.4s (24.6s)	13.7s (6.7s)
SML/NJ	6.2s (2.4s)	15.1s (6.9s)	6.1s (2.4s)
Isabelle	1.5s (0.5s)	6.4s (2.9s)	1.5s (0.5s)

Table 7.1: Efficiency of SML code.

Table 7.1 shows that the generated kernel in SML, using built-in equality, performs just as efficiently as the original version. Using the generated equality function slows the execution down substantially on all compilers. Especially on Moscow ML we experience a huge drop in efficiency.

Chapter 8

Experiments

In this chapter we experiment with the declarative proof language in the proof assistant implemented in Harrison's handbook using the generated kernel.

In Chapter 2 we discussed the set of problems for automated theorem provers (ATP) by Pelletier. In particular, we highlighted problem 43. The majority of the problems are included in the test suite in *full_test.sml*, but some of them are not solved by full automation in Harrison's proof assistant, at least in reasonable time, including problem 43. The proof assistant also comes with a declarative proof language that is somewhat similar to the Isar language in Isabelle, but obviously much simpler.

We present here three proofs of the problem following experiments with the proof language. We highlight key concepts of the language and compare the three proofs.

In the proof below we show each direction of $\forall x.\, \forall y.\, Q(x,y) \longleftrightarrow Q(x,y)$ to be a consequence of $(\forall x.\, (\forall y.\, Q(x,y) \longleftrightarrow \forall z.\, P(z,x) \longleftrightarrow P(z,y)))$:

```
prove
  (<<"(forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)) \
        ==> forall x y. Q(x,y) <=> Q(y,x)">>)
  [
    assume [("A", <<"forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)">>)],
    conclude (<<"forall x y. Q(x,y) <=> Q(y,x)">>) proof
    [
      fix "x", fix "y",
      note ("R", <<"Q(x,y) ==> Q(y,x)">>) proof
      [
        assume [("", <<"Q(x,y)">>)],
        so have (<<"forall z. P(z,x) <=> P(z,y)">>) by ["A"],
        so have (<<"forall z. P(z,y) <=> P(z,x)">>) at once,
        so conclude (<<"Q(y,x)">>) by ["A"],
        qed
      ],
      note ("L", <<"Q(y,x) ==> Q(x,y)">>) proof
      [
        assume [("", <<"Q(y,x)">>)],
        so have (<<"forall z. P(z,y) <=> P(z,x)">>) by ["A"],
        so have (<<"forall z. P(z,x) <=> P(z,y)">>) at once,
        so conclude (<<"Q(x,y)">>) by ["A"],
        qed
      ],
      conclude (<<"Q(x,y) <=> Q(y,x)">>) by ["R", "L"],
      qed
    ],
    qed
  ]
```

We assume the left-hand side of the implication with the *assume* command and give it the name "A". The command is in many ways similar to the Isabelle counterpart.

The lines below introduce new variables by reducing the quantifiers, and then directly solve the subgoal (the right-hand side of the implication) following a proof:

```
fix "x", fix "y",
conclude (<<"forall x y. Q(x,y) <=> Q(y,x)">>) proof
```

The command *have* is similar to its Isabelle counterpart and *so have* is similar to *then have* in Isabelle. The command *have (...)* is really just a sugared alias for a note without a name: *note("", ...)*. Each direction of the bi-implication is proved and used together to complete the proof.

The next proof we consider is quite explicit and avoids the use *note* by clever use of the built-in capabilities of the language:

```
prove
  (<<"(forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)) \
        ==> forall x y. Q(x,y) <=> Q(y,x)">>)
  [
    assume [("A", <<"forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)">>)],
    conclude (<<"forall x y. Q(x,y) <=> Q(y,x)">>) proof
    [
      fix "x", fix "y",
      conclude (<<"Q(x,y) <=> Q(y,x)">>) proof
      [
        have (<<"(Q(x,y) ==> Q(y,x)) /\ (Q(y,x) ==> Q(x,y))">>) proof
        [
          conclude (<<"Q(x,y) ==> Q(y,x)">>) proof
          [
            assume [("", <<"Q(x,y)">>)],
            so have (<<"forall z. P(z,x) <=> P(z,y)">>) by ["A"],
            so have (<<"forall z. P(z,y) <=> P(z,x)">>) at once,
            so conclude (<<"Q(y,x)">>) by ["A"],
            qed
          ],
          conclude (<<"Q(y,x) ==> Q(x,y)">>) proof
          [
            assume [("", <<"Q(y,x)">>)],
            so have (<<"forall z. P(z,y) <=> P(z,x)">>) by ["A"],
            so have (<<"forall z. P(z,x) <=> P(z,y)">>) at once,
            so conclude (<<"Q(x,y)">>) by ["A"],
            qed
          ],
          qed
        ],
        so our thesis at once,
        qed
      ],
      qed
    ],
    qed
  ]
```

We see that each direction of the bi-implication is shown as a conjunction:

```
have (<<"(Q(x,y) ==> Q(y,x)) /\ (Q(y,x) ==> Q(x,y))">>) proof
```

When this is the case, we can use the *conclude* command to solve each part of the conjunction independently (the proofs have been left out):

```
have (<<"(Q(x,y) ==> Q(y,x)) /\ (Q(y,x) ==> Q(x,y))">>) proof
  conclude (<<"Q(x,y) ==> Q(y,x)">>) proof ...,
  conclude (<<"Q(y,x) ==> Q(x,y)">>) proof ...,
  qed
]
```

From the proved conjunction we can show the main formula by equivalence of bi-implication and a conjunction of implications for both directions:

```
have (<<"(Q(x,y) ==> Q(y,x)) /\ (Q(y,x) ==> Q(x,y))">>) proof
  conclude (<<"Q(x,y) ==> Q(y,x)">>) proof ...,
  conclude (<<"Q(y,x) ==> Q(x,y)">>) proof ...,
  qed
],
so our thesis at once,
qed
```

The command *at once* can be used when the goal can be solved by pure first-order reasoning from the previous fact.

The last proof we present is less explicit than the previous two, but as a result it is rather short:

```
prove
  (<<"(forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)) \
        ==> forall x y. Q(x,y) <=> Q(y,x)">>)
  [
    assume [("A", <<"forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)">>)],
    fix "x", fix "y",
    note ("R", <<"Q(x,y) ==> Q(y,x)">>) proof
    [
      assume [("", <<"Q(x,y)">>)],
      so have (<<"forall z. P(z,x) <=> P(z,y)">>) by ["A"],
      so our thesis by ["A"],
      qed
    ],
    have (<<"Q(y,x) ==> Q(x,y)">>) proof
    [
      assume [("", <<"Q(y,x)">>)],
      so have (<<"forall z. P(z,y) <=> P(z,x)">>) by ["A"],
      so our thesis by ["A"],
      qed
    ],
    so our thesis by ["R"],
    qed
  ]
```

Each direction is shown without directly solving any subgoal before combining them in the end. Also the step where the symmetry of the bi-implication $P(z, x) \longleftrightarrow P(z, y)$ is shown is left out and used implicitly.

Despite limitations in the ATPs of the proof assistant, we see that more complex problems can be solved when we combine the automated procedures with a structured proof language.

Chapter 9

Conclusion

We here conclude on the results of the work of this thesis in regard to the goals we set out to achieve. The main goal of this thesis is to successfully

(1) formalize in Isabelle Harrison's axiomatic proof system,

(2) prove the soundness of the proof system, and

(3) generate executable code from the formalization and compare the results to the original code using a test suite.

The thesis formalizes the proof system by defining the syntax of first-order logic, and axioms and rules of the proof system. A small example using the proof system is given. This suggests that goal (1) is fulfilled, unless the formalization is not a correct implementation of the definitions in the proof system.

The thesis proves the proof system sound by first introducing the semantics of first-order logic, and then showing that the soundness property holds for the proof system with the defined syntax and semantics. Thus, goal (2) is also fulfilled given the success of goal (1).

The thesis shows how to generate code from the formalization, and how it can be hooked into the existing code base. The results of a test suite across different versions show no differences, which lead us to conclude that the formalization

and generated code is in fact correct. Thus, goal (3) is also fulfilled. This further implies that goals (1) and (2) also must have been fulfilled.

The distinction between syntax and semantics became clear in the period from the 1840s with the work of Boole, and up to the 1930s where this led Gödel to his Incompleteness Theorem.

> Likewise, it only became clear over an extended period that in logic it is important to distinguish between syntax (including such notions as formal language, formula, proof, and consistency) and semantics (including such notions as truth, model, and satisfiability). [Moo88, p. 96]

One of the results of our formalization is to make this distinction between the level of syntax and semantics very clear.

In Section 3.2 we described the soundness and completeness of our proof system. Harrison argues that completeness of the proof system holds, but verification of a formalized completeness proof exceeded our time frame. The approach to the soundness proof was to show each axiom sound, and to show that each rule preserved this property. For completeness, we need to show to that all valid formulas can be derived. Consequently, the nature of such a proof takes a different approach and is more difficult. Given the opportunity, we would like to prove the completeness of HAPS in Isabelle.

We mentioned that the setup of the code generator is rather lengthy, as we have to hook the generated code into an existing code base. We would like to better streamline this setup, such that no modification to the code generator is necessary.

The declarative language of the proof assistant that we experimented with in Chapter 8 could be improved in terms of better formatting of the proofs. Furthermore, we could add more features to the language. Since the kernel is implemented in LCF-style and has been verified, any new feature we add to the proof assistant will also be verified.

Appendix A

Isabelle Theory File

---

theory *Proven*
imports
Main ~~/*src*/*HOL*/*Library*/*Code-Char*
begin

A.1 Syntax of First-Order Logic

type-synonym *id* = *String.literal*

datatype *tm* = *Var id* | *Fn id* (*tm list*)

datatype *'a fm* = *T* | *F* | *Atom 'a* | *Imp* (*'a fm*) (*'a fm*) | *Iff* (*'a fm*) (*'a fm*) |
And (*'a fm*) (*'a fm*) | *Or* (*'a fm*) (*'a fm*) | *Not* (*'a fm*) |
Exists id (*'a fm*) | *Forall id* (*'a fm*)

datatype *fol* = *R id tm list*

datatype *fol-thm* = *Thm* (*concl*: *fol fm*)

A.2 Definition of Rules and Axioms

abbreviation (*input*) *fail-thm* ≡ *Thm T*

definition *fol-equal* :: *fol fm* ⇒ *fol fm* ⇒ *bool*
where
fol-equal p q ≡ *p* = *q*

definition *zip-eq* :: *tm list* ⇒ *tm list* ⇒ *fol fm list*
where
zip-eq l l′ ≡ *map* (λ(*t*, *t′*). *Atom* (*R* (*STR* ″=″) [*t*, *t′*])) (*zip l l′*)

primrec *imp-chain* :: *fol fm list* ⇒ *fol fm* ⇒ *fol fm*
where
imp-chain [] *q* = *q* |
imp-chain (*p* # *l*) *q* = *Imp p* (*imp-chain l q*)

primrec *occurs-in* :: *id* ⇒ *tm* ⇒ *bool* **and** *occurs-in-list* :: *id* ⇒ *tm list* ⇒ *bool*
where
occurs-in i (*Var x*) = (*i* = *x*) |
occurs-in i (*Fn* - *l*) = *occurs-in-list i l* |
occurs-in-list - [] = *False* |
occurs-in-list i (*h* # *t*) = (*occurs-in i h* ∨ *occurs-in-list i t*)

primrec *free-in* :: *id* ⇒ *fol fm* ⇒ *bool*
where
free-in - *T* = *False* |
free-in - *F* = *False* |
free-in i (*Atom a*) = (*case a of R* - *l* ⇒ *occurs-in-list i l*) |
free-in i (*Imp p q*) = (*free-in i p* ∨ *free-in i q*) |
free-in i (*Iff p q*) = (*free-in i p* ∨ *free-in i q*) |
free-in i (*And p q*) = (*free-in i p* ∨ *free-in i q*) |
free-in i (*Or p q*) = (*free-in i p* ∨ *free-in i q*) |
free-in i (*Not p*) = *free-in i p* |
free-in i (*Exists x p*) = (*i* ≠ *x* ∧ *free-in i p*) |
free-in i (*Forall x p*) = (*i* ≠ *x* ∧ *free-in i p*)

primrec *equal-length* :: *tm list* ⇒ *tm list* ⇒ *bool*
where
equal-length l [] = (*case l of* [] ⇒ *True* | - # - ⇒ *False*) |
equal-length l (- # *r′*) = (*case l of* [] ⇒ *False* | - # *l′* ⇒ *equal-length l′ r′*)

definition *modusponens* :: *fol-thm* ⇒ *fol-thm* ⇒ *fol-thm*
where
modusponens s s′ ≡ *case concl s of Imp p q* ⇒
let p′ = *concl s′ in if fol-equal p p′ then Thm q else fail-thm* | - ⇒ *fail-thm*

definition *gen* :: *id* ⇒ *fol-thm* ⇒ *fol-thm*
where
gen x s ≡ *Thm* (*Forall x* (*concl s*))

definition *axiom-addimp* :: *fol fm* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-addimp p q ≡ *Thm* (*Imp p* (*Imp q p*))

definition *axiom-distribimp* :: *fol fm* ⇒ *fol fm* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-distribimp p q r ≡ *Thm* (*Imp* (*Imp p* (*Imp q r*))
(*Imp* (*Imp p q*) (*Imp p r*)))

definition *axiom-doubleneg* :: *fol fm* ⇒ *fol-thm*
where
axiom-doubleneg p ≡ *Thm* (*Imp* (*Imp* (*Imp p F*) *F*) *p*)

definition *axiom-allimp* :: *id* ⇒ *fol fm* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-allimp x p q ≡ *Thm* (*Imp* (*Forall x* (*Imp p q*))
(*Imp* (*Forall x p*) (*Forall x q*)))

definition *axiom-impall* :: *id* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-impall x p ≡ *if* ¬ *free-in x p then Thm* (*Imp p* (*Forall x p*)) *else fail-thm*

definition *axiom-existseq* :: *id* ⇒ *tm* ⇒ *fol-thm*
where
axiom-existseq x t ≡ *if* ¬ *occurs-in x t*
then Thm (*Exists x* (*Atom* (*R* (*STR* ''='') [*Var x*, *t*]))) *else fail-thm*

definition *axiom-eqrefl* :: *tm* ⇒ *fol-thm*
where
axiom-eqrefl t ≡ *Thm* (*Atom* (*R* (*STR* ''='') [*t*, *t*]))

definition *axiom-funcong* :: *id* ⇒ *tm list* ⇒ *tm list* ⇒ *fol-thm*
where
axiom-funcong i l l′ ≡ *if equal-length l l′*
then Thm (*imp-chain* (*zip-eq l l′*) (*Atom* (*R* (*STR* ''='') [*Fn i l*, *Fn i l′*])))
else fail-thm

definition *axiom-predcong* :: *id* ⇒ *tm list* ⇒ *tm list* ⇒ *fol-thm*
where
axiom-predcong i l l′ ≡ *if equal-length l l′*
then Thm (*imp-chain* (*zip-eq l l′*) (*Imp* (*Atom* (*R i l*)) (*Atom* (*R i l′*))))
else fail-thm

definition *axiom-iffimp1* :: *fol fm* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-iffimp1 p q ≡ *Thm* (*Imp* (*Iff p q*) (*Imp p q*))

definition *axiom-iffimp2* :: *fol fm* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-iffimp2 p q ≡ *Thm* (*Imp* (*Iff p q*) (*Imp q p*))

definition *axiom-impiff* :: *fol fm* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-impiff p q ≡ *Thm* (*Imp* (*Imp p q*) (*Imp* (*Imp q p*) (*Iff p q*)))

definition *axiom-true* :: *fol-thm*
where
axiom-true ≡ *Thm* (*Iff T* (*Imp F F*))

definition *axiom-not* :: *fol fm* ⇒ *fol-thm*
where
axiom-not p ≡ *Thm* (*Iff* (*Not p*) (*Imp p F*))

definition *axiom-and* :: *fol fm* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-and p q ≡ *Thm* (*Iff* (*And p q*) (*Imp* (*Imp p* (*Imp q F*)) *F*))

definition *axiom-or* :: *fol fm* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-or p q ≡ *Thm* (*Iff* (*Or p q*) (*Not* (*And* (*Not p*) (*Not q*))))

definition *axiom-exists* :: *id* ⇒ *fol fm* ⇒ *fol-thm*
where
axiom-exists x p ≡ *Thm* (*Iff* (*Exists x p*) (*Not* (*Forall x* (*Not p*))))

A.3 Code Generation for Rules and Axioms

code-printing type-constructor *tm* ⇀ (*SML*) *term* |
constant *Var* ⇀ (*SML*) *Var* - |
constant *Fn* ⇀ (*SML*) *Fn* (-, -)

code-printing type-constructor *fm* ⇀ (*SML*) - *formula* |
constant *T* ⇀ (*SML*) *True* |
constant *F* ⇀ (*SML*) *False* |
constant *Atom* ⇀ (*SML*) *Atom* - |
constant *Imp* ⇀ (*SML*) *Imp* (-, -) |
constant *Iff* ⇀ (*SML*) *Iff* (-, -) |
constant *And* ⇀ (*SML*) *And* (-, -) |
constant *Or* ⇀ (*SML*) *Or* (-, -) |
constant *Not* ⇀ (*SML*) *Not* - |
constant *Exists* ⇀ (*SML*) *Exists* (-, -) |
constant *Forall* ⇀ (*SML*) *Forall* (-, -)

code-printing type-constructor *fol* ⇀ (*SML*) *fol* |

constant $R \rightharpoonup (SML)\ R$ (-, -)

code-printing — More efficient
constant *fol-equal* $\rightharpoonup$ (SML) - = -

export-code
modusponens gen axiom-addimp axiom-distribimp axiom-doubleneg axiom-allimp

axiom-impall axiom-existseq axiom-eqrefl axiom-funcong axiom-predcong
axiom-iffimp1 axiom-iffimp2 axiom-impiff axiom-true axiom-not axiom-and
axiom-or axiom-exists concl
in *SML* **module-name** *Proven* **file** *Proven.sml*

A.4 Semantics of First-Order Logic

definition *length2* :: *tm list* ⇒ *bool*
where
length2 l ≡ *case l of* [-,-] ⇒ *True* | - ⇒ *False*

primrec — Semantics of terms
semantics-term :: (*id* ⇒ *'a*) ⇒ (*id* ⇒ *'a list* ⇒ *'a*) ⇒ *tm* ⇒ *'a* **and**
semantics-list :: (*id* ⇒ *'a*) ⇒ (*id* ⇒ *'a list* ⇒ *'a*) ⇒ *tm list* ⇒ *'a list*
where
semantics-term e - (*Var x*) = *e x* |
semantics-term e f (*Fn i l*) = *f i* (*semantics-list e f l*) |
semantics-list - - [] = [] |
semantics-list e f (*t* # *l*) = *semantics-term e f t* # *semantics-list e f l*

primrec — Semantics of formulas
semantics :: (*id* ⇒ *'a*) ⇒ (*id* ⇒ *'a list* ⇒ *'a*) ⇒ (*id* ⇒ *'a list* ⇒ *bool*)
⇒ *fol fm* ⇒ *bool*
where
semantics - - - *T* = *True* |
semantics - - - *F* = *False* |
semantics e f g (*Atom a*) = (*case a of R i l* ⇒ *if i* = *STR ''='' ∧ length2 l*
then (*semantics-term e f* (*hd l*) = *semantics-term e f* (*hd* (*tl l*)))
else g i (*semantics-list e f l*)) |
semantics e f g (*Imp p q*) = (*semantics e f g p* ⟶ *semantics e f g q*) |
semantics e f g (*Iff p q*) = (*semantics e f g p* ⟷ *semantics e f g q*) |
semantics e f g (*And p q*) = (*semantics e f g p* ∧ *semantics e f g q*) |
semantics e f g (*Or p q*) = (*semantics e f g p* ∨ *semantics e f g q*) |
semantics e f g (*Not p*) = (¬ *semantics e f g p*) |
semantics e f g (*Exists x p*) = (∃ *v*. *semantics* (*e*(*x* := *v*)) *f g p*) |
semantics e f g (*Forall x p*) = (∀ *v*. *semantics* (*e*(*x* := *v*)) *f g p*)

A.5 Definition of Proof System

inductive *OK* :: *fol fm* ⇒ *bool* (⊢ - *0*)
where
modusponens:
⊢ *concl s* ⟹ ⊢ *concl s′* ⟹ ⊢ *concl* (*modusponens s s′*) |
gen:
⊢ *concl s* ⟹ ⊢ *concl* (*gen* - *s*) |
axiom-addimp:
⊢ *concl* (*axiom-addimp* - -) |
axiom-distribimp:
⊢ *concl* (*axiom-distribimp* - - -) |
axiom-doubleneg:
⊢ *concl* (*axiom-doubleneg* -) |
axiom-allimp:
⊢ *concl* (*axiom-allimp* - - -) |
axiom-impall:
⊢ *concl* (*axiom-impall* - -) |
axiom-existseq:
⊢ *concl* (*axiom-existseq* - -) |
axiom-eqrefl:
⊢ *concl* (*axiom-eqrefl* -) |
axiom-funcong:
⊢ *concl* (*axiom-funcong* - - -) |
axiom-predcong:
⊢ *concl* (*axiom-predcong* - - -) |
axiom-iffimp1:
⊢ *concl* (*axiom-iffimp1* - -) |
axiom-iffimp2:
⊢ *concl* (*axiom-iffimp2* - -) |
axiom-impiff:
⊢ *concl* (*axiom-impiff* - -) |
axiom-true:
⊢ *concl axiom-true* |
axiom-not:
⊢ *concl* (*axiom-not* -) |
axiom-and:
⊢ *concl* (*axiom-and* - -) |
axiom-or:
⊢ *concl* (*axiom-or* - -) |
axiom-exists:
⊢ *concl* (*axiom-exists* - -)

corollary ⊢ *Imp p p*
proof −
have *1*: ⊢ *concl* (*Thm* (*Imp* (*Imp p* (*Imp* (*Imp p p*) *p*))
(*Imp* (*Imp p* (*Imp p p*)) (*Imp p p*))))
using *axiom-distribimp*

unfolding *axiom-distribimp-def*
by *simp*

have *2*: $\vdash$ *concl* (*Thm* (*Imp p* (*Imp* (*Imp p p*) *p*)))
using *axiom-addimp*
unfolding *axiom-addimp-def*
by *simp*

have *3*: $\vdash$ *concl* (*Thm* (*Imp* (*Imp p* (*Imp p p*)) (*Imp p p*)))
using *1 2 modusponens*
unfolding *modusponens-def fol-equal-def*
by *fastforce*

have *4*: $\vdash$ *concl* (*Thm* (*Imp p* (*Imp p p*)))
using *axiom-addimp*
unfolding *axiom-addimp-def*
by *simp*

have *5*: $\vdash$ *concl* (*Thm* (*Imp p p*))
using *3 4 modusponens*
unfolding *modusponens-def fol-equal-def*
by *fastforce*

show *?thesis*
using *5*
by *simp*
qed

A.6 Soundness of Proof System

lemma *map'*:
$\neg$ *occurs-in x t* $\Longrightarrow$ *semantics-term e f t* = *semantics-term* (*e*(*x* := *v*)) *f t*
$\neg$ *occurs-in-list x l* $\Longrightarrow$ *semantics-list e f l* = *semantics-list* (*e*(*x* := *v*)) *f l*
by (*induct t* **and** *l rule*: *semantics-term.induct semantics-list.induct*) *simp-all*

lemma *map*:
$\neg$ *free-in x p* $\Longrightarrow$ *semantics e f g p* $\longleftrightarrow$ *semantics* (*e*(*x* := *v*)) *f g p*
proof (*induct p arbitrary*: *e*)
fix *e*
show $\neg$ *free-in x T* $\Longrightarrow$ *semantics e f g T* $\longleftrightarrow$ *semantics* (*e*(*x* := *v*)) *f g T*
by *simp*
next
fix *e*
show $\neg$ *free-in x F* $\Longrightarrow$ *semantics e f g F* $\longleftrightarrow$ *semantics* (*e*(*x* := *v*)) *f g F*
by *simp*
next
fix *a e*

```
  show ¬ free-in x (Atom a) ⟹
     semantics e f g (Atom a) ⟷
     semantics (e(x := v)) f g (Atom a)
  proof (induct a)
   fix i l
   assume ¬ free-in x (Atom (R i l))
   then have fresh: ¬ occurs-in-list x l
   by simp
   show semantics e f g (Atom (R i l)) ⟷
      semantics (e(x := v)) f g (Atom (R i l))
   proof cases
    assume eq: i = STR ''='' ∧ length2 l
    then have semantics e f g (Atom (R i l)) ⟷
       semantics-term e f (hd l) =
       semantics-term e f (hd (tl l))
    by simp
    also have ... ⟷
       semantics-term (e(x := v)) f (hd l) =
       semantics-term (e(x := v)) f (hd (tl l))
    using map'(1) fresh occurs-in-list.simps eq list.case-eq-if list.collapse
    unfolding length2-def
    by metis
    finally show ?thesis
    using eq
    by simp
   next
    assume not-eq: ¬ (i = STR ''='' ∧ length2 l)
    then have semantics e f g (Atom (R i l)) ⟷ g i (semantics-list e f l)
    by simp iprover
    also have ... ⟷ g i (semantics-list (e(x := v)) f l)
    using map'(2) fresh
    by metis
    finally show ?thesis
    using not-eq
    by simp iprover
   qed
  qed
next
  fix p1 p2 e
  assume assm1: ¬ free-in x p1 ⟹
     semantics e f g p1 ⟷ semantics (e(x := v)) f g p1 for e
  assume assm2: ¬ free-in x p2 ⟹
     semantics e f g p2 ⟷ semantics (e(x := v)) f g p2 for e
  show ¬ free-in x (Imp p1 p2) ⟹
     semantics e f g (Imp p1 p2) ⟷ semantics (e(x := v)) f g (Imp p1 p2)
  using assm1 assm2
  by simp
```

next
fix *p1 p2 e*
assume *assm1*: ¬ *free-in x p1* ⟹
semantics e f g p1 ⟷ *semantics* (*e*(*x* := *v*)) *f g p1* **for** *e*
assume *assm2*: ¬ *free-in x p2* ⟹
semantics e f g p2 ⟷ *semantics* (*e*(*x* := *v*)) *f g p2* **for** *e*
show ¬ *free-in x* (*Iff p1 p2*) ⟹
semantics e f g (*Iff p1 p2*) ⟷ *semantics* (*e*(*x* := *v*)) *f g* (*Iff p1 p2*)
using *assm1 assm2*
by *simp*
next
fix *p1 p2 e*
assume *assm1*: ¬ *free-in x p1* ⟹
semantics e f g p1 ⟷ *semantics* (*e*(*x* := *v*)) *f g p1* **for** *e*
assume *assm2*: ¬ *free-in x p2* ⟹
semantics e f g p2 ⟷ *semantics* (*e*(*x* := *v*)) *f g p2* **for** *e*
show ¬ *free-in x* (*And p1 p2*) ⟹
semantics e f g (*And p1 p2*) ⟷ *semantics* (*e*(*x* := *v*)) *f g* (*And p1 p2*)
using *assm1 assm2*
by *simp*
next
fix *p1 p2 e*
assume *assm1*: ¬ *free-in x p1* ⟹
semantics e f g p1 ⟷
semantics (*e*(*x* := *v*)) *f g p1* **for** *e*
assume *assm2*: ¬ *free-in x p2* ⟹
semantics e f g p2 ⟷
semantics (*e*(*x* := *v*)) *f g p2* **for** *e*
show ¬ *free-in x* (*Or p1 p2*) ⟹
semantics e f g (*Or p1 p2*) ⟷
semantics (*e*(*x* := *v*)) *f g* (*Or p1 p2*)
using *assm1 assm2*
by *simp*
next
fix *p e*
assume ¬ *free-in x p* ⟹
semantics e f g p ⟷ *semantics* (*e*(*x* := *v*)) *f g p* **for** *e*
then show ¬ *free-in x* (*Not p*) ⟹
semantics e f g (*Not p*) ⟷ *semantics* (*e*(*x* := *v*)) *f g* (*Not p*)
by *simp*
next
fix *x1 p e*
assume ¬ *free-in x p* ⟹
semantics e f g p ⟷
semantics (*e*(*x* := *v*)) *f g p* **for** *e*
then show ¬ *free-in x* (*Exists x1 p*) ⟹
semantics e f g (*Exists x1 p*) ⟷

```
      semantics (e(x := v)) f g (Exists x1 p)
  by simp (metis fun-upd-twist fun-upd-upd)
next
  fix x1 p e
  assume ¬ free-in x p ⟹
      semantics e f g p ⟷
      semantics (e(x := v)) f g p for e
  then show ¬ free-in x (Forall x1 p) ⟹
      semantics e f g (Forall x1 p) ⟷
      semantics (e(x := v)) f g (Forall x1 p)
  by simp (metis fun-upd-twist fun-upd-upd)
qed

lemma length2-equiv:
  length2 l ⟷ [hd l, hd (tl l)] = l
proof −
  have length2 l ⟹ [hd l, hd (tl l)] = l
  unfolding length2-def
  using list.case-eq-if list.exhaust-sel
  by metis
  then show ?thesis
  unfolding length2-def
  using list.case list.case-eq-if
  by metis
qed

lemma equal-length-sym:
  equal-length l l' ⟹ equal-length l' l
proof (induct l' arbitrary: l)
  fix l
  assume equal-length l []
  then show equal-length [] l
  using equal-length.simps list.case-eq-if
  by metis
next
  fix l l' a
  assume sym: equal-length l l' ⟹ equal-length l' l for l
  assume equal-length l (a # l')
  then show equal-length (a # l') l
  using equal-length.simps list.case-eq-if list.collapse list.inject sym
  by metis
qed

lemma equal-length2:
  equal-length l l' ⟹ length2 l ⟷ length2 l'
proof −
  assume assm: equal-length l l'
```

```
  have equal-length l [t, t'] ⟹ length2 l for t t'
  unfolding length2-def
  using equal-length.simps list.case-eq-if
  by metis
  moreover have equal-length [t, t'] l' ⟹ length2 l' for t t'
  unfolding length2-def
  using equal-length.simps list.case-eq-if equal-length-sym
  by metis
  ultimately show ?thesis
  using assm length2-equiv
  by metis
qed

lemma imp-chain-equiv:
  semantics e f g (imp-chain l p) ⟷
     (∀ q ∈ set l. semantics e f g q) ⟶ semantics e f g p
using imp-conjL
by (induct l) simp-all

lemma imp-chain-zip-eq:
  equal-length l l' ⟹
     semantics e f g (imp-chain (zip-eq l l') p) ⟷
     semantics-list e f l = semantics-list e f l' ⟶ semantics e f g p
proof −
  assume equal-length l l'
  then have (∀ q ∈ set (zip-eq l l'). semantics e f g q) ⟷
     semantics-list e f l = semantics-list e f l'
  unfolding zip-eq-def
  using length2-def
  by (induct l l' rule: list-induct2') simp-all
  then show ?thesis
  using imp-chain-equiv
  by iprover
qed

lemma funcong:
  equal-length l l' ⟹
     semantics e f g (imp-chain (zip-eq l l')
        (Atom (R (STR ''='') [Fn i l, Fn i l'])))
proof −
  assume assm: equal-length l l'
  show ?thesis
  proof cases
    assume semantics-list e f l = semantics-list e f l'
    then have semantics e f g (Atom (R (STR ''='') [Fn i l, Fn i l']))
    using length2-def
    by simp
```

```
    then show ?thesis
    using imp-chain-equiv
    by iprover
  next
    assume semantics-list e f l ≠ semantics-list e f l'
    then show ?thesis
    using assm imp-chain-zip-eq
    by iprover
  qed
qed

lemma predcong:
  equal-length l l' ⟹
      semantics e f g (imp-chain (zip-eq l l')
          (Imp (Atom (R i l)) (Atom (R i l'))))
proof −
  assume assm: equal-length l l'
  show ?thesis
  proof cases
    assume eq: i = STR ''='' ∧ length2 l ∧ length2 l'
    show ?thesis
    proof cases
      assume semantics-list e f l = semantics-list e f l'
      then have semantics-list e f [hd l, hd (tl l)] =
          semantics-list e f [hd l', hd (tl l')]
      using eq length2-equiv
      by simp
      then have semantics e f g (Imp (Atom (R (STR ''='') l))
          (Atom (R (STR ''='') l')))
      using eq
      by simp
      then show ?thesis
      using eq imp-chain-equiv
      by iprover
    next
      assume semantics-list e f l ≠ semantics-list e f l'
      then show ?thesis
      using assm imp-chain-zip-eq
      by iprover
    qed
  next
    assume not-eq: ¬ (i = STR ''='' ∧ length2 l ∧ length2 l')
    show ?thesis
    proof cases
      assume semantics-list e f l = semantics-list e f l'
      then have semantics e f g (Imp (Atom (R i l)) (Atom (R i l')))
      using assm not-eq equal-length2
```

```
      by simp iprover
      then show ?thesis
      using imp-chain-equiv
      by iprover
    next
      assume semantics-list e f l ≠ semantics-list e f l'
      then show ?thesis
      using assm imp-chain-zip-eq
      by iprover
    qed
  qed
qed

theorem soundness:
  ⊢ p ⟹ semantics e f g p
proof (induct arbitrary: e rule: OK.induct)
  fix e s s'
  assume semantics e f g (concl s) semantics e f g (concl s') for e
  then show semantics e f g (concl (modusponens s s'))
  unfolding modusponens-def
  proof (induct s)
    fix r
    assume semantics e f g r semantics e f g (concl s') for e
    then show semantics e f g (concl (case r of
        Imp p q ⇒
          let p' = concl s' in if fol-equal p p'
              then Thm q else fail-thm
        | - ⇒ fail-thm))
    unfolding fol-equal-def
    by (induct r) simp-all
  qed
next
  fix e x s
  assume semantics e f g (concl s) for e
  then show semantics e f g (concl (gen x s))
  unfolding gen-def
  by simp
next
  fix e p q
  show semantics e f g (concl (axiom-addimp p q))
  unfolding axiom-addimp-def
  by simp
next
  fix e p q r
  show semantics e f g (concl (axiom-distribimp p q r))
  unfolding axiom-distribimp-def
  by simp
```

next
fix *e g p*
show *semantics e f g* (*concl* (*axiom-doubleneg p*))
unfolding *axiom-doubleneg-def*
by *simp*
next
fix *e x p q*
show *semantics e f g* (*concl* (*axiom-allimp x p q*))
unfolding *axiom-allimp-def*
by *simp*
next
fix *e x p*
show *semantics e f g* (*concl* (*axiom-impall x p*))
unfolding *axiom-impall-def*
using *map*
by *simp iprover*
next
fix *e x t*
show *semantics e f g* (*concl* (*axiom-existseq x t*))
unfolding *axiom-existseq-def*
using *map'*(*1*) *length2-def*
by *simp iprover*
next
fix *e t*
show *semantics e f g* (*concl* (*axiom-eqrefl t*))
unfolding *axiom-eqrefl-def*
using *length2-def*
by *simp*
next
fix *e i l l'*
show *semantics e f g* (*concl* (*axiom-funcong i l l'*))
unfolding *axiom-funcong-def*
using *funcong*
by *simp standard*
next
fix *e i l l'*
show *semantics e f g* (*concl* (*axiom-predcong i l l'*))
unfolding *axiom-predcong-def*
using *predcong*
by *simp standard*
next
fix *e p q*
show *semantics e f g* (*concl* (*axiom-iffimp1 p q*))
unfolding *axiom-iffimp1-def*
by *simp*
next
fix *e p q*

show *semantics e f g* (*concl* (*axiom-iffimp2 p q*))
unfolding *axiom-iffimp2-def*
by *simp*
next
fix *e p q*
show *semantics e f g* (*concl* (*axiom-impiff p q*))
unfolding *axiom-impiff-def*
by *simp* (*rule iff*)
next
fix *e*
show *semantics e f g* (*concl* (*axiom-true*))
unfolding *axiom-true-def*
by *simp*
next
fix *e p*
show *semantics e f g* (*concl* (*axiom-not p*))
unfolding *axiom-not-def*
by *simp*
next
fix *e p q*
show *semantics e f g* (*concl* (*axiom-and p q*))
unfolding *axiom-and-def*
by *simp*
next
fix *e p q*
show *semantics e f g* (*concl* (*axiom-or p q*))
unfolding *axiom-or-def*
by *simp*
next
fix *e x p*
show *semantics e f g* (*concl* (*axiom-exists x p*))
unfolding *axiom-exists-def*
by *simp*
qed

A.7 Appendix: Mentioned Built-in Facts

proposition $x \neq x' \Longrightarrow e(x := v,\ x' := v') = e(x' := v',\ x := v)$
using *fun-upd-twist* **.**

proposition $e(x := v,\ x := v') = e(x := v')$
using *fun-upd-upd* **.**

proposition $(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow P \longleftrightarrow Q$
using *iff* **.**

proposition $(P \wedge P' \longrightarrow Q) \longleftrightarrow (P \longrightarrow P' \longrightarrow Q)$

using *imp-conjL* .

proposition (*case* [] *of* [] ⇒ P | h' # t' ⇒ P' h' t') = P
using *list.case*(*1*) .

proposition (*case* h # t *of* [] ⇒ P | h' # t' ⇒ P' h' t') = P' h t
using *list.case*(*2*) .

proposition (*case* l *of* [] ⇒ P | h # t ⇒ P' h t) =
(*if* l = [] *then* P *else* P' (*hd* l) (*tl* l))
using *list.case-eq-if* .

proposition $l \neq [] \Longrightarrow hd\ l\ \#\ tl\ l = l$
using *list.collapse* .

proposition $(l = [] \Longrightarrow P) \Longrightarrow (l = hd\ l\ \#\ tl\ l \Longrightarrow P) \Longrightarrow P$
using *list.exhaust-sel* .

proposition $h\ \#\ t = h'\ \#\ t' \longleftrightarrow h = h' \wedge t = t'$
using *list.inject* .
end

Bibliography

[BA12] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 2012.

[Ber07] Stefan Berghofer. First-Order Logic According to Fitting. *Archive of Formal Proofs*, August 2007. `http://isa-afp.org/entries/FOL-Fitting.shtml`, Formal proof development.

[BGK+16] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A Learning-Based Fact Selector for Isabelle/HOL. *Journal of Automated Reasoning*, pages 1–26, 2016.

[BP16] Jasmin Christian Blanchette and Lawrence C Paulson. Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL, 2016. `http://isabelle.in.tum.de/dist/doc/sledgehammer.pdf`.

[Har06] John Harrison. Towards Self-Verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[HK16] Lars Hupel and Viktor Kuncak. Translating Scala Programs to Isabelle/HOL. In Nicola Olivetti and Ashish Tiwari, editors, *IJCAR*, volume 9706 of *LNCS*, pages 568–577. Springer, 2016.

[KAMO16] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-Formalisation of Higher-Order Logic. *Journal of Automated Reasoning*, pages 221–259, 2016.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, et al. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd*, pages 207–220. ACM, 2009.

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[Moo88] Gregory H. Moore. The Emergence of First-Order Logic. In William Aspray and Philip Kitcher, editors, *History and Philosophy of Modern Mathematics*, pages 95–135. University of Minnesota Press, 1988.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pel86] Francis Jeffry Pelletier. Seventy-Five Problems for Testing Automatic Theorem Provers. *Journal of Automated Reasoning*, pages 191–216, 1986.

[SV] Anders Schlichtkrull and Jørgen Villadsen. SML Code for Handbook of Practical Logic and Automated Reasoning. `https://github.com/logic-tools/sml-handbook/tree/master/code/SML`.

[W+16] Makarius Wenzel et al. The Isabelle/Isar Reference Manual, 2016. `http://isabelle.in.tum.de/dist/doc/isar-ref.pdf`.