Technical Report

UCAM-CL-TR-872
ISSN 1476-2986

Number 872

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

The integration of higher order interactive proof with first order automatic theorem proving

Jia Meng

July 2015

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 2005 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Interactive and automatic theorem proving are the two most widely used computer-assisted theorem proving methods. Interactive proof tools such as HOL, Isabelle and PVS have been highly successful. They support expressive formalisms and have been used for verifying hardware, software, protocols, and so forth. Unfortunately interactive proof requires much effort from a skilled user. Many other tools are completely automatic, such as Vampire, SPASS and Otter. However, they cannot be used to verify large systems because their logic is inexpressive. This dissertation focuses on how to combine these two types of theorem proving to obtain the advantages of each of them. This research is carried out by investigating the integration of Isabelle with Vampire and SPASS.

Isabelle is an interactive theorem prover and it supports a multiplicity of logics, such as ZF and HOL. Vampire and SPASS are first order untyped resolution provers. The objective of this research is to design an effective method to support higher order interactive proof with any first order resolution prover. This integration can simplify the formal verification procedure by reducing the user interaction required during interactive proofs: many goals will be proved by automatic provers.

For such an integration to be effective, we must bridge the many differences between a typical interactive theorem prover and a resolution theorem prover. Examples of the differences are higher order versus first order; typed versus untyped.

Through experiments, we have designed and implemented a practical method to convert Isabelle's formalisms (ZF and HOL) into untyped first-order clauses. Isabelle/ZF's formulae that are not first-order need to be reformulated to first-order formulae before clause normal form transformation. For Isabelle/HOL, a sound modelling of its type system is designed first before translating its formulae into first-order clauses with its type information encoded. This method of formalization makes it possible to have Isabelle integrated with resolutions.

A large set of axioms is usually required to support interactive proofs but can easily overwhelm an automatic prover. We have experimented with various methods to solve this problem, including using different settings of an automatic prover and automatically eliminating irrelevant axioms.

The invocation of background automatic provers should be invisible to users, hence we have also designed and implemented an automatic calling procedure, which extracts all necessary information and sends it to an automatic prover at an appropriate point during an Isabelle proof.

Finally, the results and knowledge gained from this research are generally applicable and can be applied to future integration of any other interactive and automatic theorem provers.

Contents

Acknowledgements

I am very grateful to my supervisor Larry Paulson, who has given me invaluable advice, encouragement and freedom to explore. Larry has been always happy to answer my questions (even during weekends) and has guided me towards the goal of my research. I also admire him for his deep and broad knowledge, creativity, productivity and his keen insight into research.

I would also like to thank my other colleagues in Automated Reasoning Group, both present and past. I thank Mike Gordon for giving me lots of help; I thank Joe Hurd for many interesting and useful discussions; I thank Claire Quigley, with whom I have been happily working on the same project; I thank Anthony Fox, Hasan Amjad and Mick Compton for answering me many Latex questions; I also thank other members of my group such as Juliano Iyoda for making my time here very enjoyable.

Outside the Computer Laboratory, I have been having a great time with other researchers in other institutions. Professor Andrei Voronkov and Alexandre Riazanov from the Vampire team have given me support in using their system. Geoff Sutcliffe from University of Miami has added the problems I have found to his TPTP library. I would like to thank all of them.

My thanks also go to the EPSRC (grant GR/S57198/01, Automation for Interactive Proof), the Computer Laboratory and the Cambridge Overseas Trust, who have generously funded my research.

Last by not least, I would like to thank my parents. They have always supported and encouraged me to pursue my study and research.

Chapter 1

Introduction

It has been a long time since the importance of formal verification, which uses rigorous mathematics to formally prove properties of software and hardware, was first observed. Since then there has been much active research going on in this field. In particular, people have recognized the important role of computers in the process of formal verification. Consequently, the development of automated reasoning has flourished. This dissertation describes our research that aims to further improve the performance of computer-assisted theorem proving. In this chapter, we demonstrate the motivation of our research. We also give an overview of formal verification and computer-assisted theorem proving.

1.1 Formal Specification and Verification

With the rapid development of digital computers, people increasingly reply upon computerized systems. These systems can be found everywhere in our daily lives: we communicate with our friends and families sitting anywhere in the world via emails and net conferences; we carry out online banking transactions; all households are full of electronic equipments; aircrafts, factories and power stations are all using computer systems to control their electronic instruments. In a bigger scale, we have seen computer systems being used as a major component in national defence systems, on which our safety lies.

However, the reliability and trustworthiness of these computer systems cannot be taken for granted. The computer systems (both software and hardware) are enormously complicated, since they are built to meet real world requirements, which are themselves complex. For instance, an application software such as OpenOffice suite has around nine million lines of code (LOC), whereas a Linux kernel has about thirty million LOC. An Intel P4 chip is made up of fifty five million transistors.

On the other hand, we — the designers and implementors of these systems — are not perfect and we can easily introduce mistakes when building these large scale systems. It is not surprising if any of these systems contains errors regardless how much effort has been spent in the process of design and implementation.

However, the potential consequences of computer systems' errors must not be underestimated. At the very least, a malfunctioning system could incur a huge financial cost. When Intel discovered a flaw in their Floating Point Unit [57], they lost $475 million to cost of sales and to cover replacement costs, etc. In many safety critical systems, errors in computerized systems could even lead to lost of lives. A terrifying incident happened

in 1996, where a mistake in the design of a radar system used by the North American Air Defence headquarters nearly brought several countries into a war [31].

There are several approaches that aim to ensure correct designs and implementations of complex systems, especially those safety critical systems. Among them, system testing and formal proofs using mathematics are most widely used. However, exhaustive testing is infeasible and thus testing alone is not adequate for safety critical systems. In comparison, formal specification and verification is the most rigorous way to guarantee the correctness of a system. Although a formally verified system may still contain faults, for example, if the design is not appropriate, most of the errors can be eliminated as users are obliged to write down all the fine details, and nothing can be swept under the carpet.

The idea of using formal methods on program design and implementation was first proposed by computer scientists such as John McCarthy in the 1960s. Formal specification and verification requires one to first model a system (a program or a piece of hardware) mathematically and then write down the properties that the system should meet in terms of mathematical assertions. Moreover, one also needs to represent the actual implementation of the system in mathematical expressions. Subsequently, the correctness of the system is verified by proving (using mathematical deductions) the mathematical expressions, which represent the implementation, satisfy the mathematical assertions, which represent the system properties. Early work on this included the Floyd-Hoare logic invented by Robert Floyd and Tony Hoare, which was specifically designed to reason about the properties of software programs. Although formal verification was originally designed to prove the correctness of software, it has been used widely for hardware verifications and protocol verifications. Over the years, there has been much progress in the field of formal methods.

1.2 Motivation of Mechanized Theorem Proving

In the early years when formal verification was advocated, all proofs were carried out by hand. It was soon realized that proofs, especially those for large scale systems, were lengthy and tedious. Consequently, handwritten proofs were very much error prone.

Many scientists then envisaged the use of computers to assist human proofs. In their vision, it would be desirable if computers could provide two functions: constructing proofs automatically and verifying the proofs carried out by human experts. There are many problems whose proofs are tedious and complex. However, they may not require creativity or deep thinking to be solved. These problems are particularly suitable for computers to solve automatically. Moreover, for those problems that require insight of human experts and thus have to be solved by humans, their proofs will be more reliable after they are checked by computers, since any errors such as typos can be eliminated from the proof process. More importantly, compared with human beings, computer systems are more objective and thus less likely to be affected by wishful thinking.

The two objectives of using computers to assist human proofs led to the inventions of two types of provers, namely *automatic theorem provers* and *interactive theorem provers*. Since their inventions, these two types of provers have been developed continually and have been used in many large scale verification projects. In addition to formal verifications, theorem provers have been applied to prove mathematical problems. One of the recent examples is the POPLmark Challenge [4], which uses theorem provers to assist the formal

reasoning about programming languages.

1.3 Automatic Theorem Provers

Automatic provers implement various logics, among them, first-order logic is the most widely used. Furthermore, many inference rules, which are complete for first-order logic, have been invented to perform automatic proof search. Examples of the inference rules are resolution and paramodulation, and tableau calculi. However, within the realm of first-order logic, resolution is the most powerful technique and is therefore implemented by most automatic provers.

Input to automatic provers usually has to conform to some standard format, such as *clause normal form*. This means that any arbitrary formula has to be converted to this format before they can be accepted by an automatic prover.

Examples of the automatic provers are Otter [33], Vampire [52] and SPASS [65]. Most of these automatic provers have much in common. For instance, they are all designed to prove one-shot problems. They all require little human guidance in a proof search: most of the time, a user is only required to submit a goal with a collection of relevant axioms and then hit the button to start the proof process. Once a proof starts, usually nothing can be done to influence the proof search. However, depending on the techniques used, there may be certain amount of work that may be carried out before the proof process starts. For instance, most of the provers provide numerous settings that can be specified by users. The settings are usually essential for the proof search performance. Most of the automatic provers suffer from a same problem — combinatorial explosion, which has been a major limitation for them.

1.4 Interactive Theorem Provers

Compared with automatic theorem provers, a very different approach to assist human constructing proofs is to use interactive provers. With an interactive prover, a user usually carries out a proof by giving directions to a prover on how to proceed with a proof. Although interactive provers are not automatic, and are not expected to find proofs of goals automatically, they can significantly contribute to the formal verification procedure by checking the proofs carried out by human experts and by storing and managing those already proved lemmas. Some interactive provers also provide certain amount of automation, therefore they are frequently called semi-automatic provers. Examples of interactive provers are HOL [23], Isabelle [38], Nuprl [16], PVS [40], the ΩMEGA system [58], Coq [26] and KIV [49].

Interactive provers usually base their specification languages on very expressive logics, such as higher-order logic. Unlike first-order logic, which is untyped, higher-order logic introduces types such as boolean types, function types and more complicated compound types. Moreover, unlike first-order logic, higher-order logic allows quantification over boolean and function values. The rich formalism of higher-order logic makes it easier and more natural to model software and hardware and to express the desired properties.

Moreover, the inference rules implemented by interactive provers include the natural deduction calculus and the sequent calculus, which are significantly different from the

inference rules, such as resolution, implemented by automatic provers. As a result, interactive provers do not impose any constraint on the format of their input: a formula can have any arbitrary form.

1.5 Using Automatic Provers to Support Interactive Proofs

Performing rigorous mathematical proofs on the correctness of a system design and implementation can increase one's confidence in the system. Proofs may be even more important for safety-critical systems. Moreover, computer-assisted proofs give significant advantages over hand-written proofs since they can help reduce possible errors in proofs and relieve humans of tedious proof steps. When carrying out mechanized formal specification and verification, one essentially faces two options, he can use either an interactive prover or an automatic prover.

1.5.1 Automatic Provers or Interactive Provers?

Compared with interactive provers, automatic provers are completely automatic and ask for little user attention in guiding the proof search. However, formal verification, especially of those large scale systems, has several major requirements on theorem provers, which are not met by automatic provers. In comparison, interactive provers can easily satisfy those requirements.

First, formal verification usually requires the logics of theorem provers to be expressive and rich. However, many automatic provers, including resolution provers, are based on first-order logic, whose expressiveness is inadequate. A central technique used in verification is induction, either simple mathematical induction or general induction on a data structure, which is not offered by automatic provers. In contrast, the logics of interactive provers (such as higher-order logic) are much more expressive: we can easily define complex data structures and prove properties that have to be satisfied by a system using induction principles. There is always a trade-off between expressiveness of a logic and its automation power.

Second, a system verification typically involves proofs of hundreds of theorems, which need to be stored after being proved, so that they can be retrieved to support a later part of a proof. Moreover, a verification may run over several sessions: reloading libraries and adding more lemmas to existing libraries is a common task. This is in fact one of the major reasons behind the use of computers to assist human proofs: human beings are more likely to forget about what lemmas have been proved or make errors when using already proved lemmas. Unfortunately, most automatic provers are designed for one-shot problems, and thus they do not provide facilities to store previously proved theorems or organize related theorems into libraries for reuse. In comparison, most interactive provers provide facilities such that once we finish proving a theorem, it can be added automatically to a background library and can be loaded when necessary or when a new session starts.

Third, having large numbers of already proved lemmas should mean that a prover should be able to solve problems in the presence of these lemmas, where many of them may be irrelevant to a given problem. Either the prover should be able to distinguish

the relevant lemmas from irrelevant ones or a user should be able to give some hint to the prover about the relevance of lemmas. However, as automatic provers are designed to be fully automatic, they do not provide an adequate facility to allow users to guide the proof search. If we simply send all the proved lemmas to them without any hint, it is very likely that these automatic provers will not be able to filter out the irrelevant ones and thus may not be able to prove the submitted problem. On the other hand, a user can easily advise an interactive prover on how to use a proved lemma or which lemmas might be relevant to a particular goal. This information makes it much easier for an interactive prover to prove a goal.

Based on the considerations above, we may conclude that interactive provers are better suited to carry out formal verification directly. However, interactive provers are still not perfect. The ability for us to directly give instructions on how to prove a goal can be seen as both an advantage and a drawback. The advantage is that human beings have clearer insight into how to solve a problem than machines, thus their guidance is valuable to find a proof. However, the guidance also requires much human interaction, and from a skilled user. Although many interactive provers are semi-automatic, by providing certain amount of automation to their users, there are still many tedious proofs that require detailed instructions from users. Some people even suspect that formal verification is too expensive and time consuming and is thus only suitable to verify safety critical systems. This is probably one main reason why computer-assisted verification has not been adopted everywhere in the industry. Therefore, improving the automation of interactive provers seems a key to simplify the formal verification procedure and thus to encourage the use of formal methods in more verifications.

1.5.2 Combining Automatic and Interactive Provers

It is important to improve interactive provers' automation so that users' interaction can be reduced. Although automatic provers are not suitable to be used directly for formal verification, they are good candidates to assist interactive provers by automatically proving some goals.

An interactive prover that is assisted by an automatic prover should work as follows. While an interactive prover user performs normal work, the interactive prover will delegate a proof goal to an automatic prover to prove. If a proof is found by the automatic prover then no more direction will be needed from the user. However, if the goal is too complicated, then the user will still have an option to prove it himself or to simplify the goal a little and let the automatic prover try again. The existing advantages of interactive provers, such as the theorem management and the rich formalism, are still available to the user, but the amount of human interaction, especially those detailed instructions on how to prove a goal, will be significantly reduced.

Moreover, as the aim of using automatic provers is to reduce interaction, it would be ideal if users would not be asked to explicitly call an automatic prover and specify what goals should be sent to them. The invocation of automatic provers should be invisible from users' point of view, and the automatic prover should run in the background. Furthermore, users should not be expected to manually decide what lemmas are relevant for a particular proof goal: all the previously proved lemmas should be sent to the target automatic prover. Finally, a proof found from the automatic prover should not be taken for granted:

mistakes can occur even if the automatic prover is sound. For instance, errors can be made when goals and results are sent between interactive and automatic provers. Therefore, the proofs found should be checked by the interactive prover before being accepted.

1.5.3 Our Contribution

We have investigated how to effectively combine interactive and automatic theorem proving by integrating Isabelle and two resolution-based theorem provers: Vampire and SPASS. We chose Isabelle as the target interactive prover as it is widely used. We integrate Isabelle with resolution-based provers as they are by far the most powerful automatic provers. Vampire and SPASS are both leading resolution provers that have done well in recent CASCs (the CADE ATP System Competition)[1].

We now summarize our research as follows.

- We have identified the major obstacles in integrating interactive provers and automatic provers. The finding is generally applicable and is not specific to our target provers — Isabelle and Vampire or SPASS.

- We have designed a practical method to translate Isabelle/ZF and Isabelle/HOL formalisms to first-order logic. In particular, we encoded Isabelle/HOL's type system and order-sorted polymorphism in first-order logic.

- We have examined the feasibility of using resolution to support Isabelle verifications, by running a large number of experiments on Vampire and SPASS. We come to a conclusion that resolution can indeed improve Isabelle's automation. Through the experiments, we also found several settings of these provers that are suitable for verification problems.

- We have designed and implemented a procedure that calls background automatic provers at an appropriate point in an Isabelle proof so that the invocation is invisible to users. This procedure extracts all necessary information and automatically translates them to clause form and sends them to automatic provers. This transformation is carried out inside Isabelle logic, since the proofs returned from Vampire and SPASS should be reconstructed to Isabelle proofs to be verified.

- We have also carried out many experiments in order to improve the performance of our integration. We focused on how to deal with large numbers of axioms present in Isabelle proofs.

1.6 Thesis Overview

In this thesis, we start by giving some background technical knowledge of resolution-based automatic provers and Isabelle (Chapter 2). We then illustrate our investigation into using resolution to support Isabelle proofs (Chapter 3). Subsequently, we describe an implementation of a procedure that automatically calls our target automatic provers (Chapter 4). In order to improve the performance of our integration, we carried out two

[1]See http://www.cs.miami.edu/tptp/CASC/

other researches. The work and its results are shown afterwards (Chapter 5 and Chapter 6). After we describe our research work, we give a brief description of some related work that has been done and compare our approaches to theirs (Chapter 7). Finally, we present conclusions (Chapter 8).

Chapter 2

Isabelle and Resolution-Based Provers

Our research investigates integrating interactive and automatic theorem proving by linking Isabelle with Vampire and SPASS. In this chapter, we briefly describe the technical background of these two types of provers and some features specific to them. We start by giving a brief historical outline of automated reasoning.

2.1 The History of Automated Reasoning

It has been several decades since theorem provers were first invented. Rapid developments have been observed in both automatic provers and interactive provers. For a more detailed account of the development of theorem provers, please refer to the book written by Donald MacKenzie [31].

2.1.1 Automatic Theorem Provers

The invention and development of automatic provers began in the 1950s, with the objective of using computers to prove theorems automatically.

The first generation of automatic provers worked on propositional logic. Examples include the Logic Theory Machine, invented by Herbert Simon and Allen Newell and Geometry Machine, developed by Herbert Gelernter, J. R. Hansen and Donald Loveland in the 1950s.

Since these early attempts on constructing automatic provers, many techniques have been invented to deal with various logics. Examples are resolution, tableau calculi and the inverse method. A host of provers have been created that implement these techniques.

Resolution and Its Predecessors

Soon after the invention of the first generation of automatic provers that worked on propositional logic, computer scientists started working on automatic proof procedures that worked on first-order logic. First-order logic is much more expressive than propositional logic. Consequently an automatic proof procedure that can prove first-order problems is

very useful. However, there is no general procedure for first-order logic. Much attention has been attracted to its development.

One common problem suffered by all the early first-order provers was the combinatorial explosion. A proof of a first-order formula involving variables usually requires instantiations of those variables and thus generating propositional formulae. However, in the early-day provers, such instantiation was unguided, which led to huge numbers of useless propositional formulae being generated. Another cause of poor performance of these provers was the difficulty of checking the satisfiability of formulae.

Dag Prawitz designed a method that restricted random instantiations of variables via unification so that only those that could lead to proofs were carried out.

Meanwhile, Hilary Putnam and Martin Davis invented a method known as Davis-Putnum procedure that efficiently checks the satisfiability of a formula in the propositional logic by first converting the formula into *conjunctive normal form*, which represents a formula as a conjunction of clauses, where each clause is a disjunction of atomic formulae or negated atomic formulae.

Prawitz's unification and Davis-Putnum procedure addressed the two problems of the then first-order provers. Based on their findings and motivated by combining the two steps into one, Alan Robinson invented a new powerful decision procedure known as *resolution* [5]. Resolution proves by refutation: it proves a formula to be valid by proving the negation of the formula to be unsatisfiable. It effectively combines unification and satisfiability checking into one step by resolving two clauses. Robinson also proved that resolution was refutationally complete, which means a proof can be found eventually if there exists one. Since resolution contains several inference rules only and no axiom, it can easily be implemented as a computer program. Therefore resolution has become a standard technique implemented in most of the modern day automatic provers.

After resolution was invented, some of its drawbacks were discovered. For instance, unguided resolution can quickly generate huge numbers of clauses, many of them are useless for a proof search. Although resolution is complete in theory, the huge search space may result in a failed proof attempt in practice.

Many improvements have been made to the initial resolution procedure, in both theoretical and practical aspects. For instance, several variant versions of resolution, such as hyper-resolution [5], ordered resolution and semantic resolution (such as *set of support strategy* [13]) have been invented, which attempt to reduce the number of generated clauses by restricting the possible resolution that could take place. In addition, inference rules such as demodulation [68], paramodulation [37] have been introduced to deal with equality reasoning. On the practical side, techniques such as advanced indexing and efficient storage of clauses have been introduced and implemented in many provers. As a result, a range of very successful resolution-based provers have emerged. We have given a list of examples in the previous chapter. Their development is still ongoing and they have been used to solve many real world problems. A famous example was the proof of Robbins problem by William McCune of Argonne National Laboratory, using the equational logic theorem prover EQP [34]. Nowadays, within the realm of first-order logic, resolution-based provers are probably more powerful than provers based on other techniques.

Other Proof Procedures

In addition to resolution-based provers, there are many provers that are based on other proof calculi. Although resolution provers have remained dominant in the community of automated reasoning for a long time, the other provers have their own strengthes in various fields.

The tableau calculi were invented before the resolution method. The earliest tableau methods could date back to the cut-free version of Gentzen's sequent calculus. Subsequently, many improvements have been made to it. An example of tableau-based provers is ${}_3T^AP$ [8].

Unlike resolution, the tableau calculi prove theorems directly, without conversion into clauses. It tries to prove a goal to be a theorem by reducing it to one or more subgoals until all the subgoals are proved, possibly via unification.

The tableau methods have flourished in recent years; they have even been competing against many resolution provers. Although they are not as powerful as resolution provers when proving first-order logic problems, tableau provers can find proofs of non-classical logic problems. In addition, some tableau-based provers have been integrated with interactive provers. Isabelle's tableau-based classical reasoner is an example and the integration of KIV with ${}_3T^AP$ [1] is another example. There are several variants of the tableau methods, such as connection tableaux, model elimination and mating.

TPS [3] is a theorem proving system for classical type theory, also known as higher-order logic. It is based on the typed λ-calculus and it supports automatic proofs, interactive proofs and a mixture of both.

TPS performs both the natural deduction calculus and the so-called expansion proofs and it provides facilities to convert from one form to the other. Its automatic proof mode operates on expansion trees and uses mating search [2]. Mating search is similar to resolution proof in that they both prove a theorem by refutation using unification of literals. However, the difference between them is that resolution proofs require formulae to be converted to conjunctive normal form, whereas mating search only requires formulae in negation normal form.

2.1.2 Interactive Theorem Provers

There are several successful families of interactive provers, and one of them is the LCF-family.

LCF Provers

The first generation of LCF provers was a proof checker for Scott's Logic for Computable Functions developed in Stanford University by Robin Milner and his colleagues in the early 1970s. This version is also known as Stanford LCF.

Using Stanford LCF, users constructed proofs in a backward goal-oriented style interactively. Users could give detailed instruction on how to reduce a proof goal to several subgoals and how to prove the goals directly. In addition, theorems that had been proved could be stored and managed into theorem libraries. Libraries could be retrieved when a new proof session started and users could add more theorems back to the library when a session finished.

Following Stanford LCF, the second generation of LCF, called Edinburgh LCF [22] was invented by Milner and other colleagues in Edinburgh. This new version created a new high-level functional language called the "metalanguage" (ML), which allowed users to write proofs and proof procedures within it. ML is a strongly typed language. Many abstract types can be defined in it, and among them, one particular type is `thm`, which represents valid theorems. A set of inference rules represented by ML's functions are defined in the LCF kernel and only those functions are allowed to derive a new `thm` object from existing `thm` objects. In addition, ML's compiler type checks all objects have the correct types and thus a `thm` object is either an axiom of type `thm` or is derived from existing axioms via some inference rules.

In addition, a great amount of automation was provided via the use of high level tactics and tacticals. Users can construct backward proofs to a goal and once all subgoals are proved, tactics can construct the corresponding forward proofs to derive the theorem. In addition to built-in tactics, users can define their own tactics for later reuse.

After the success of Edinburgh LCF, several descendants of it have emerged, which are all written in ML and rely on ML's type checking to ensure soundness of proofs. Examples of widely used LCF style provers are HOL [23], Cambridge LCF [41], Isabelle [38] and Nuprl [16].

The HOL system was invented by Mike Gordon. Since it was originally built to verify hardware properties, and higher-order logic is particularly suitable for hardware verification, Gordon decided to implement higher-order logic. Since its invention, HOL has been used widely. For example, it has been used for hardware verifications such as verifying the correctness of the Viper microprocessor [15], the formal verification of the ARM6 micro-architecture [19] and also in network protocol specifications, such as the Netsem project [59, 60].

Isabelle was invented by Larry Paulson. In contrast to HOL, it does not have any logic hardwired into the system. Instead, a meta-logic was implemented in Isabelle that could represent a variety of object-logics, including higher-order logic (Isabelle/HOL) and Zermelo-Fraenkel set theory (Isabelle/ZF). Isabelle has been used widely to verify many protocols and to formalize mathematical theories. For instance, Isabelle/HOL has been used to verify the SET protocol suite [10] and Java bytecode [28].

Other Interactive Provers

In addition to the LCF style provers, there are many other successful interactive provers, which all aim to verify the proofs conducted by humans and at the same time providing a certain amount of automation.

PVS (Prototype Verification System) [40] was invented by the SRI verification group. PVS implements classical higher-order logic as its specification language. Similar to HOL and Isabelle, specifications and verifications developments are arranged into theories, which consist of definitions, proofs of lemmas etc.

PVS is typed and its type system supports *predicate subtyping* and dependent types, which can be used to serve as constraints on types. This fine-grained type system is more expressive and thus the verification can be formalized more naturally. Unfortunately, type checking is undecidable. It becomes the users' responsibility to prove that terms are well-typed.

PVS is based on a sequent calculus. PVS proofs are interactive, and users use the provided inference rules to instruct the prover how to prove a goal. In addition to interactive proofs, PVS provides many automatic decision procedures. It has been used to accomplish many proofs, such as the proof of Lamport's clock convergence algorithm [53].

Another interactive proof assistant is the ΩMEGA system [58]. It is mainly designed to support the working mathematicians, rather than formal verification of computer systems. Unlike many other interactive provers, which are stand alone systems, the ΩMEGA system is made up of multiple modules, distributed in the MathWeb network.

ΩMEGA supports both interactive proofs that can be carried out by its users, and also automatic proof procedures. The automation is mainly provided by the external reasoning systems that have been integrated with ΩMEGA.

2.2 Resolution-Based Automatic Provers

Resolution-based automatic provers are impressive in their power, and they can prove exceptionally complex theorems. For our research, we have experimented integrating Isabelle with Vampire and SPASS, which are leading resolution provers that have done well in recent CASCs. In addition, we have tried some examples on E [54, 55].

Most of these automatic theorem provers are untyped and work in first-order logic. They also usually require inputs to be in clause form.

We describe resolution theorem proving in general in this section. Subsequently, we introduce some specific features of the provers that we have tried.

2.2.1 First-Order Logic

First-order logic is built up from *terms* that stand for individuals and *formulae* that stand for truth values.

Definition 1. The terms of first-order logic are defined recursively as follows:

- A *variable* is a term.
- A *constant symbol* is a term. A constant symbol is also known as a *zero-place* function symbol.
- A *function application* $f(t_1, \ldots, t_n)$ is a term, if $t_1, \ldots, t_n$ are terms and f is an n-place function symbol.

Definition 2. The formulae of first-order logic are defined recursively as follows:

- If $t_1, \ldots, t_n$ are terms and P is an n-place predicate symbol, then $P(t_1, \ldots, t_n)$ is a formula, known as *atomic formula*.
- If P and Q are formulae, then $\neg P$, $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$ and $P \leftrightarrow Q$ are formulae.
- If x is a variable and P is a formula, then $\forall x\, P$ and $\exists x\, P$ are formulae, which mean P is true for all x and P is true for some x respectively.

2.2.2 Clause Normal Form

Resolution theorem provers require input formulae to be in *clause normal form* (also known as *conjunctive normal form*), which is a conjunction of clauses: a *clause* is a disjunction of literals and a *literal* is either an atomic formula or a negated atomic formula. A formula in clause form is usually written as

$$(A_1 \vee \cdots \vee A_n) \wedge \ldots \wedge (D_1 \vee \ldots \vee D_m)$$

where $A_1, \ldots, A_n$ and $D_1, \ldots, D_m$ are all literals. An *empty clause* is a clause without any literal. This represents logical falsity and is written as $\bot$.

Any first-order formula may be translated to clause form using a standard *clause normal form transformation*.

Definition 3. The standard clause normal form transformation consists of the following steps:

1. Given a first-order logic formula, eliminate all boolean equalities "$\leftrightarrow$" and implications "$\rightarrow$" by repeatedly applying the equalities

$$A \leftrightarrow B \simeq (A \rightarrow B) \wedge (B \rightarrow A)$$
$$A \rightarrow B \simeq \neg A \vee B$$

2. Convert the formula into negation normal form by pushing in negations "$\neg$" until they apply only to atomic formulae, using the following equalities

$$\neg(A \vee B) \simeq (\neg A \wedge \neg B)$$
$$\neg(A \wedge B) \simeq (\neg A \vee \neg B)$$
$$\neg(\exists x\, A) \simeq (\forall x\, \neg A)$$
$$\neg(\forall x\, A) \simeq (\exists x\, \neg A)$$
$$\neg\neg A \simeq A$$

3. Pull out all quantifiers to the front of the formula, and variable renaming may be required to avoid name clashes. The result formula is in prenex normal form.

4. Remove all existentially quantified variables using Skolemization. Start from the leftmost existential variable y, replace all occurrences of y in the formula by a Skolem term $f(x_1, \ldots, x_n)$, where f is a new function symbol and $x_1, \ldots, x_n$ are those universally quantified variables standing on the left of $\exists y$. If there is no universal variable on the left of $\exists y$, then f is simply a constant symbol. Finally, drop $\exists y$ from the formula and repeat the procedure from the next leftmost existential variable until none is left.

5. Push in disjunctions "$\vee$" until they apply only to literals, by repeatedly applying the distributive law

$$(A \wedge B) \vee C \simeq (A \vee C) \wedge (B \vee C)$$

6. Drop all universal quantifiers and all free variables are now implicitly universally quantified.

In practice, there are many optimizations to the standard clause normal form transformation. For example, it is not necessary to pull out universal quantifiers; doing so may increase their scopes, which may lead to bigger Skolem terms. Instead of pulling out quantifiers, one can try to push them in. This is a procedure called *miniscoping*. For Skolemization to work, it is sufficient to identify those universal variables that cover the scope of the occurrences of existential variables.

2.2.3 Resolution Theorem Proving

Resolution-based theorem proving [5] proves theorems by refutation. It is sound and complete. This is a powerful theorem proving method and can be mechanized relatively easily.

The resolution calculus has a collection of inference rules and no axioms. Each inference rule operates on clauses. A basic resolution system is made up of two inference rules:

Binary Resolution resolves two clauses, using the inference rule

$$\frac{A \vee B \quad \neg C \vee D}{(A \vee D)\sigma}$$

where $B\sigma = C\sigma$ and σ is the most general unifier of B and C (a unifier is a substitution of variables). Literal B is the *resolved literal* and the conclusion $(A \vee D)\sigma$ is the *resolvent*. Moreover, $\neg C \vee D$ is usually called the *main premise* whereas $A \vee B$ is called the *side premise*. A refutation is found if an empty clause is derived, which happens when an inference such as

$$\frac{A \quad \neg B}{\bot}$$

is performed, where there exists a unifier σ such that $A\sigma = B\sigma$.

Factoring unifies literals in a same clause using the inference rule

$$\frac{A_1 \vee \ldots \vee A_m \vee B_1 \vee \ldots \vee B_n}{(A_1 \vee \ldots \vee A_m \vee B_1)\sigma}$$

where $B_1\sigma = \ldots = B_n\sigma$. In first-order resolution, factoring is essential for completeness.

A resolution proof procedure starts with a set of clauses. Subsequently, the prover repeatedly applies the resolution rule to resolve each clause with each other clause and hence generates new clauses and adds them to the existing clause set. A proof search stops either when an empty clause $\bot$ is derived or when the set of clauses is *saturated*, meaning no more new clauses (or clauses that are logical consequences of existing clauses) can be generated.

A naive implementation of resolution can quickly generate large numbers of clauses, many of which are useless. This will lead to both slow progress and running out of memory. Several refinements to the standard resolution have been designed and implemented in automatic provers. Earlier systems such as Otter rely on hyper-resolution, while more recent ones such as Vampire and SPASS use *ordered resolution*, which essentially restricts possible clauses that can be generated by considering an *ordering* among literals and clauses.

Ordered resolution aims to derive resolvent clauses that are smaller than their parent clauses so that the empty clause, which is the "smallest" clause, can be generated. Therefore an ordering should be imposed on literals and clauses. When a resolution is to be performed, a prover always tries to select heavier literals to resolve. However, in first-order logic, an implemented ordering is usually not total. In practice, most resolution-based provers use a *literal selection function* or strategy that selects the most promising literals to participate in resolution. For instance, a literal selection function may decide that only maximal positive literals in a side premise can be resolved with either the maximal negative literal in the main premise or all the selected negative literals in the main premise. In addition, factoring may only be allowed on positive literals.

Resolution has been proved to be refutationally complete in theory. In practice, when there are thousands of clauses waiting to be resolved, not all clauses can be selected and perform resolution. Therefore completeness requires the clause selection to be fair. This means that any non-redundant clause must be eventually selected for resolution. Most provers implement some version of a *clause selection function*, which attempts to select a suitable clause, out of possibly thousands of clauses, to take part in a resolution step.

Simplifying the current clauses set is another technique that can improve resolution provers' performance. The idea is to delete *redundant* clauses, according to some redundancy criteria. Tautology clauses are redundant and can be discarded safely. Several other criteria have been designed. For example, if a clause C *subsumes* another clause D, meaning the list of literals in C is a subset of the list of literals in D (possibly after applying some substitution σ to C), then clause D is redundant and can be removed. Removing a subsumed clause D does not compromise completeness because it can be inferred by C. Demodulation is another commonly used technique. It rewrites a clause by substituting a term s with an equal term t that is "simpler" than s (again possibly after applying some substitution σ to t and s). It has been proved that the deletion of redundant clauses based on these criteria will not make resolution incomplete.

The *set of support* heuristic is another key feature in many automatic provers. This strategy is a type of semantic resolution, which according to Chang and Lee [13] dates from 1965, forbids resolution steps that take all their clauses from a part of the clause set known to be satisfiable. This feature is useful for problems where large numbers of axioms are included, since it prevents deductions purely involving the axioms. However, since it is incomplete in the presence of modern ordering heuristics, it is normally switched off.

Related to set of support is the distinction between axiom and (negated) conjecture clauses. The former arise from the axioms describing the problem domain, which we expect to be consistent. The latter arise from the negated conjecture, which must be inconsistent with the axiom clauses if the conjecture is provable. Some automatic provers make use of the distinction between the two types of clauses. In our integration, we translate Isabelle theorems to axiom clauses and negated goals to negated conjecture

clauses. When preparing clauses for resolution theorem provers, we must label them appropriately.

Paramodulation

Following the development of resolution, *paramodulation* [37] was invented to deal with equalities. Instead of specifying equalities as a predicate with usual axioms such as reflexivity, symmetry, transitivity and congruence rules, paramodulation treats equality as part of the logic. Equality reasoning is regarded as an inference rule

$$\frac{A \vee s \simeq t \quad B}{(A \vee B[t]_p)\sigma}$$

where if $B|_p$ is the subterm at position p in B, then $B[t]_p$ is the result of substituting that subterm for t and σ is the most general unifier of $B|_p$ and s.

Similar to the problem of the standard resolution, the standard paramodulation without any control can easily blow up the search space by generating a large number of useless clauses. A solution to this problem is *ordered paramodulation* (superposition), where a term can only be replaced by a smaller one, with respect to some term ordering. Nowadays, most of the resolution-based automatic provers have built-in paramodulation as part of the inference mechanism.

2.3 Vampire

Vampire [52] is a resolution- and paramodulation-based automatic theorem prover for first-order logic with and without equality. It consists of two major components, namely the resolution and paramodulation inference system and a preprocessor that converts any first-order formula into clause form. Although it supports input in an arbitrary first-order format, we only send clauses to it as we perform clause normal form transformation inside Isabelle.

2.3.1 Vampire's Inference Mechanism

Vampire implements several versions of resolution, such as ordered resolution and hyper-resolution. However, hyper-resolution is only applicable to logics that have no equality. Vampire also implements ordered paramodulation (superposition).

Term Ordering

The term ordering is a parameter for ordered resolution, superposition and simplification. During an ordered resolution, a literal is resolved if it has a higher weight relative to other literals in the same clause. This difference in weights gives an ordering on literals. In addition, a term ordering also determines a reduction order, which is used when clauses are simplified via rewriting.

Vampire uses the Knuth-Bendix Ordering (KBO) [51] to compute this ordering on literals. KBO is parameterized by weights and precedences of functions and predicates, which can be assigned explicitly by users. However the resulting KBO is a partial ordering on terms with variables.

Clause Selection and Literal Selection

Vampire implements both clause selection function and literal selection function. For the former, clauses are selected in an order according to their *age-weight* ratio. The clause selection function prefers older clauses to younger clauses and smaller clauses to bigger clauses.

Vampire implements several versions of literal selection function, which select literals based on their polarity and weight. For instance, Vampire provides both negative and positive selection, which are complete. It also defines several other incomplete literal selection functions. In some situations, these incomplete literal selection functions can produce a very quick proof result. Our experimental results showed that the literal selection function was one of the most important settings that influenced the proof performance.

Simplification

Like many other resolution-based theorem provers, Vampire implements a set of simplification procedures to simplify clauses and to remove redundant ones. Examples include removing tautology clauses and removing clauses that are subsumed by others. Other simplification rules of Vampire are demodulation, branch demodulation and subsumption.

Simplification takes place at various places, and is usually interleaved with resolutions. For instance, after new clauses are generated by resolution, they are first simplified or removed before being added to the existing clause set. This is called *forward simplification*. After these new clauses are simplified, they are subsequently used to simplify clauses in the existing clause set. This is called *backward simplification*.

Other Features

Vampire provides a *limited resource strategy* (LRS), which is used when users impose a time limit on a proof search. According to the time limit, this strategy discards those clauses that are too big to be processed before the time limit expires. Although LRS may render the proof incomplete, it is useful for our integration, since we always impose a time limit on an automatic prover when sending goals to it.

Another key feature of Vampire is its set of support strategy (it is called *axioms for support only* in Vampire). As it may be incomplete, Vampire's set of support is switched off by default.

2.3.2 A New Version

In the public release of Vampire, the supported clause format is TPTP[1] clause format. In the new version of Vampire (v6.03), a special syntax is used to specify which literal in a clause should be selected for resolution. This syntax is an extension of TPTP syntax. Any positive literal that should be resolved first will be tagged with `+++`, and similarly a negative literal should be tagged with `---`. This feature turns out very important for our integration as it simulates Isabelle's treatment to previously proved lemmas and supports

[1]Thousands of Problems for Theorem Provers [62]

Isabelle's notions of forward and backward chaining. Many lemmas could only be proved with this facility.

2.4 SPASS

SPASS [65] is another powerful resolution-based theorem prover for first-order logic with equality.

SPASS provides many user settings. Some of these settings are similar to Vampire's, such as the set of support strategy, the time limit on which SPASS can run, the literal selection strategy, weight-depth ratio for the clause selection strategy, simplification procedure, ordered resolution and hyper-resolution. It also allows users to set weights to functions and variables.

SPASS implements two reduction orderings. In addition to KBO, which is implemented by Vampire, SPASS implements the recursive path ordering with status (RPOS) [18]. Users can specify which ordering is appropriate for a proof.

In addition to the settings above, SPASS provides some other special features. One such example is *auto* mode. When auto mode is switched on, SPASS can automatically configure itself, in terms of the choice of inference rules, ordering on literals and literal, clause selection strategy. By default, it is switched on and it makes proof search complete.

2.5 E

Although Vampire and SPASS are the main provers we have used for experiments, we also briefly tried several experiments on E 0.82 Lung Ching [54, 55].

E is an equational theorem prover and proves by refutation. Its inference mechanism consists of superposition and rewriting only. Unlike other resolution-based provers, resolution is not built into E but is simulated by paramodulation and equational resolution.

One of the major emphasis of E is the clause selection heuristics. When there are huge numbers of clauses, it is important for a prover to select a promising clause to participate in resolution, so that the resolvent is useful to find a shorter proof, or at least, to find a proof. Therefore a good design of clause selection heuristics is essential. In addition to the age-weight ratio, which is provided by E as a standard clause selection heuristic, E also implements many other heuristics. These heuristics essentially let users decide the number of priority queues, which the clauses should be inserted into. Subsequently E selects clauses from their queues according to their ordering, and each queue is considered in a round-robin manner. The clause selection heuristics can be set by users or by the automatic mode of E. The power to select proper clause to resolve using these heuristics is one of the major strengths of E.

In addition, E defines several versions of literal selection functions. The term orderings implemented in E include both KBO and Lexicographic Path Ordering (LPO), which can be selected by users.

Finally, in order to make it easier to use E by its users, E provides an automatic mode. It not only selects a clause selection heuristic, but also selects a literal selection strategy and term ordering, by analyzing the characteristics of submitted problems.

2.6 Isabelle

Isabelle [38] is a powerful interactive theorem prover based on the typed λ-calculus. It is generic and supports a multiplicity of logics.

Isabelle is written in ML and is an LCF style theorem prover. Like other provers that are based on the LCF architecture, it allows proofs to be constructed only within a small kernel, which defines the basic inference rules. All decision procedures and other proof mechanisms must ultimately reduce their deductions to basic inference rules and axioms. Such an architecture makes proof procedures more difficult to implement, but it greatly improves their reliability.

A *proof goal* in Isabelle is a statement expressed in some logic. A *rule* is a previously proved lemma or theorem. Users need to specify what *tactics* and rules to use in order to prove a goal or decompose it into several smaller subgoals. A tactic tells Isabelle how the rules should be applied. Isabelle provides substantial automation by classical reasoning tactics and equality reasoning tactics.

2.6.1 The History of Isabelle

It has been nearly 20 years since the first public release of Isabelle. Since then, many changes have been made to Isabelle in each successive version. Isabelle has become wide spread and has been used by both researchers and industrial users for formal verifications. It is still being developed and will continue to grow. We briefly outline the development history of Isabelle and several key changes made to it.

Isabelle was made publicly available for the first time in 1986. This version, also called Isabelle-86, implemented many object-logics, such as Zermelo-Fraenkel set theory and Constructive Type Theory. The rule calculus employed was sequent calculus. Furthermore, higher-order unification was also supported. Significant amount of automation was already provided in the form of tactics and tacticals.

In the subsequent releases, a meta-logic was introduced into Isabelle and the natural deduction calculus replaced the original sequent calculus. Moreover, order-sorted polymorphism was implemented in order to support higher-order logic.

In addition to the technical changes, support for Isabelle users has increased. One of the most important changes made to Isabelle was the introduction of Isar proof language in a recent release. It allows users to write proofs in a way similar to human reasoning and thus makes proofs more readable.

2.6.2 Isabelle's Logical Framework

Isabelle's logical framework works in a natural deduction style. It supports elimination rules and introduction rules, which are used by forward and backward chaining respectively.

In a natural deduction calculus, each logical operator has both an elimination rule and an introduction rule. An elimination rule describes what we can deduce from a formula containing the operator, whereas an introduction rule tells us how we can deduce a formula containing the operator. For instance, the elimination rule for implication ($\rightarrow$) says if we know $P \rightarrow Q$ and we know P then we can deduce Q. This is usually written as

$$\frac{P \to Q \quad P}{Q} \ (\to I)$$

On the other hand, the introduction rule of implication says that if by assuming P we can prove Q then we can deduce $P \to Q$, which is written as

$$\frac{\begin{array}{c}[P] \\ \vdots \\ Q\end{array}}{P \to Q} \ (\to E)$$

where the P in the bracket indicates a discharge of assumption.

Frequently, elimination rules are used for forward chaining, where we derive new lemmas or theorems from existing theorems. Introduction rules are usually used in a backward chaining style, where a proof goal is decomposed into several smaller subgoals.

A major difference between Isabelle and other LCF style provers is that Isabelle's built-in logic, the *meta-logic*, is intended only for the formalization of other logics — the *object-logics*. The design philosophy of Isabelle [42] is that by making a built-in meta-logic that can represent any object-logic, many difficulties shared by those object-logics can be dealt with altogether. One typical example is the proof search strategy, such as depth-first search or depth-first iterative deepening search. These search strategies may be applicable to any proof goal, regardless which object-logic is being used to formalize the goal.

Isabelle's Meta-Logic

Isabelle's meta-logic is a fragment of higher-order logic. It defines the following constants.

- Implication ($\Longrightarrow$) for logical entailment and discharge of assumptions.
- Universal quantifier ($\bigwedge$) for generality of variables.
- λ-abstractions.
- Meta-equality ($\equiv$) for meta-level definitions and equivalence.

The meta-logic employs a natural deduction calculus and defines a set of primitive inference rules. They include the introduction and elimination rules for $\Longrightarrow$, $\bigwedge$ and $\equiv$ between truth values. Reflexivity, symmetry and transitivity rules are defined for $\equiv$ as well. Furthermore, α-conversion, β-conversion, extensionality and abstraction, combination rules are defined for λ-terms.

Isabelle's meta-logic defines many abstract types, which should be distinguished from the types in ML. For instance, all Isabelle well-formed terms have type `term`. Moreover, type `thm` represents object-level theorems or axioms. A `thm` object can be converted to a `term` object that represents the term structure of the theorem.

Like other LCF style theorem provers, Isabelle uses ML's secure type-checking to enforce soundness. The definition of `thm` constitutes the inference kernel. A theorem — which is a value of type `thm` — can only be constructed by applying some inference rules, which are defined in the kernel and have type such as `thm -> thm`, ultimately to axioms

(typically constants of type `thm`). Because ML's type-checker prevents arbitrary formulae from being assigned type `thm`, any expression having this type represents a correct proof.

Isabelle users seldom interact with the primitive inference rules during proofs. Instead, proofs are usually constructed using Isabelle's tactics and tacticals (§2.6.3). Isabelle's tactics are functions that represent derived meta-level inference rules; they are built on top of the primitive inference rules.

Representing Object-Logics in Isabelle

When object-logics are formalized in Isabelle, they are represented *inside* the meta-logic by extending the existing meta-logic. This is achieved by adding new types, constants and meta-level axioms as follows.

- For each operator of an object-logic, define a new operator in the meta-logic to represent it. Similarly for the types of the object-logic.
- For each inference rule of the object-logic, a meta-level axiom of type `thm` is defined to represent it.

The representation of object-logic rules in the meta-logic may require some elaboration. First, each object-logic proposition is "lifted" to a proposition in the meta-level by a special predicate *Trueprop*: *Trueprop*(P) means proposition P is true in the object-logic. *Trueprop*(P) is usually abbreviated by $[\![P]\!]$. Subsequently, object-logic entailment and discharge of assumptions are represented by meta-implications $\Longrightarrow$. Moreover, the meta-quantifier $\bigwedge$ is used to state any generality requirement on variables occurring in object-logic rules. Consider formalizing first-order logic as an example. First, we define new constants such as $\rightarrow$ for first-order implication, $\forall$ for universal quantification, and $\wedge$ for conjunction. We also define type *bool* for first-order propositions. Finally, we represent the first-order inference rules by meta-theorems. For example, the introduction rule for implication $(\rightarrow I)$ is expressed by the meta-level theorem

$$\bigwedge P\, Q.\ ([\![P]\!] \Longrightarrow [\![Q]\!]) \Longrightarrow [\![P \rightarrow Q]\!].$$

In addition, for the elimination rule of universal quantifier

$$\frac{P(x)}{\forall x\, P(x)}\ (\forall I)$$

where the proviso for $\forall I$ is that variable x must not be free in P, we use the following meta-level theorem to formalize it.

$$\bigwedge P.\ (\bigwedge x.\ [\![P(x)]\!]) \Longrightarrow [\![\forall x\, P(x)]\!]$$

Usually the outer-most meta-quantifiers $\bigwedge$ are dropped, leaving those bound variables implicitly universally quantified. These variables are called *schematic variables*. They are usually identified by having a "?" in front. Therefore the above two rules are written in Isabelle as

$$([\![?P]\!] \Longrightarrow [\![?Q]\!]) \Longrightarrow [\![?P \rightarrow ?Q]\!]$$
$$(\bigwedge x.\ [\![?P(x)]\!]) \Longrightarrow [\![\forall x\, ?P(x)]\!]$$

In general, an object-logic inference rule

$$\frac{\phi_1 \ldots \phi_n}{\theta}$$

is formalized by an Isabelle meta-theorem

$$[\![\phi_1; \ldots \phi_n]\!] \Longrightarrow \theta$$

with implicit universal quantification over its free variables. Furthermore, $[\![\phi_1; \ldots \phi_n]\!] \Longrightarrow \theta$ is an abbreviation for the nested implications $\phi_1 \Longrightarrow \ldots \Longrightarrow \phi_n \Longrightarrow \theta$.

> *For the sake of readability, from now on, we will represent all Isabelle rules in its equivalent higher-order logic form, and we will replace* $\bigwedge$ *by* $\forall$ *and replace* $\Longrightarrow$ *by* $\rightarrow$.

Finally, Isabelle represents all object-logic elimination rules in a special format:

$$\forall P\,[A \rightarrow \forall \mathbf{x}_1\,(\mathbf{B}_1 \rightarrow P) \rightarrow \ldots \rightarrow \forall \mathbf{x}_n\,(\mathbf{B}_n \rightarrow P) \rightarrow P].$$

Here, A is a formula that contains an operator to be eliminated. Each $\mathbf{x}_i$ is a list of universally quantified variables. $\mathbf{B}_1, \ldots, \mathbf{B}_n$ are lists of formulae, where each of them is regarded as conjunctions. In addition, each $\mathbf{x}_i$ and $\mathbf{B}_j$ may be empty. Furthermore, P is a predicate variable. For example, the elimination rule for set intersection is represented as

$$\forall c\,A\,B\,P\,[c \in A \cap B \rightarrow (c \in A \wedge c \in B \rightarrow P) \rightarrow P].$$

This representation is a higher-order theorem because of the universally quantified predicate P. In order to use such elimination rules in first-order automatic theorem provers, we need to preprocess them so that P can be eliminated, before translating them into first-order clauses.

2.6.3 Proofs in Isabelle

Proofs in Isabelle are performed by a set of functions defined in an LCF style kernel. Each function takes one or more theorems of type `thm` as inputs and returns another theorem (or a sequence of theorems) as a result. Both input and output theorems may be object-logic rules or a *proof state*.

One basic mechanism of Isabelle proofs is a form of *Horn clause resolution*, which is different to the resolution performed by automatic provers. A Horn resolution resolves a premise of a theorem with a conclusion of another theorem through unification. It can be described by the following inference rule

$$\frac{\phi \rightarrow \theta \quad \theta \rightarrow \psi}{\phi \rightarrow \psi}$$

In Isabelle, the Horn resolution step is a derived meta-level inference rule. It is used in both backward and forward theorem proving.

Isabelle's Proof State

As we know, LCF style proof construction works better with forward theorem proving: deriving new theorems from existing ones. However, forward proofs are difficult to reason about for human beings since it is difficult to predict what theorems should be derived for later part of a proof. In contrast, backward proofs, where a goal is decomposed to several smaller subgoals, are goal-oriented and are therefore more intuitive.

In order to allow Isabelle to work with backward proofs, the status of each proof development of a goal is represented by a *proof state*. A proof state describes the main proof goal and the current subgoals. It is formalized by a meta-level theorem (of type `thm`): the conclusion is the main goal and its premises are the current subgoals. For instance, a proof state in which the theorem to be proved is C and currently having n subgoals $(\psi_1, \ldots, \psi_n)$ is represented by the theorem

$$[\![\psi_1; \ldots \psi_n]\!] \Longrightarrow [\![C]\!].$$

More importantly, when a user starts a new proof of a goal C, the initial proof state is represented by

$$[\![C]\!] \Longrightarrow [\![C]\!]$$

where the premise is the same as the conclusion. This is a trivial theorem, and it automatically has type `thm`.

Subsequently, an Isabelle proof development is a refinement procedure on the entire proof state. The set of kernel-defined functions (typically tactics (§2.6.3)) transforms the proof state (a `thm`) to a sequence of next proof states. Meanwhile, the current subgoals are reduced to zero or more new subgoals. The theorem is finally proved when the proof state $[\![C]\!]$ is reached, at which point no more subgoal is left. Theorem $[\![C]\!]$ now represents the object-logic theorem C. Before C is declared a theorem, all free variables in it are automatically generalized to be universally quantified.

As we can see, only functions defined in the kernel are allowed to transform the proof state. These functions take the initial proof state of type `thm` to the final state, still of type `thm`, via a sequence of intermediate states, each having type `thm`. ML's secure type checking can therefore guarantee the final proof state is indeed a theorem. Thus soundness is ensured.

> *Although Isabelle represents both proof states and rules in a uniform way, in this dissertation, proof states will be written in meta-theorem form, rather than its equivalent higher-order logic form. Meta-theorem form makes it easier to pick out a particular subgoal of a proof state. The distinction may also help the readers to distinguish a proof state from an Isabelle rule.*

Tactics and Tacticals

During an Isabelle proof, the most important tools available to users are *tactics*. Users usually use these tactics to continuously refine a proof state until a final proof state is reached.

A tactic is a function from a theorem (the current proof state) to a sequence of theorems and it describes how to refine a proof state. Many tactics are parameterized: each parameterized tactic is supplied with a list of rules and the tactic then applies the rules to the current proof state. Depending on the nature of the tactic, one or more subgoals may be proved or reduced. Since a tactic may perform higher-order resolution on the proof state, there may be more than one resolvent. As a result, the return result is a sequence of all possible next states. These states are recorded and may be used later if backtracking occurs.

One of those most frequently used tactics is `resolve_tac`. Users give it a set of rules `ths`, which are normally introduction rules. Subsequently `resolve_tac ths i` resolves the conclusion of each rule in `ths` with the i-th subgoal of the proof state. Another example tactic is `rewrite_tac`. When given a list of definitions `defs` as theorems, `rewrite_tac defs` unfolds all definitions throughout all the subgoals in the proof state.

In addition to tactics, Isabelle provides *tacticals*. A tactical defines how to combine one or more tactics and apply them to a proof state. For example, using the tactical `REPEAT`, we can build a new tactic `REPEAT tac`, which repeatedly applies a tactic `tac` to a proof state and only returns when the final application of `tac` fails. The return result is thus a sequence of states that make `tac` fail. Moreover, with the tactical `THEN`, tactic $\texttt{tac}_1$ `THEN` $\texttt{tac}_2$ sequentially applies $\texttt{tac}_1$ and $\texttt{tac}_2$ to a proof state. As it shows, tacticals are convenient mechanisms to build new tactics from existing tactics.

Tactics and tacticals are available as ML functions and can be used for direct programming.

Backward and Forward Theorem Proving

Isabelle supports backward and forward proofs. Both of them are indispensable during a proof construction, but they are used rather differently.

In general, forward proof is used to derive new theorem (of type `thm`) from existing theorems using kernel-defined inference rules. There are many occasions when forward proof is applicable in Isabelle. For instance, Isabelle users may derive a new object-logic rule from another previously proved rule, by instantiating variables occurring in it. Alternatively, several rules may be joined together by resolving the conclusion of one rule with a premise of another rule (using function `RSN`). In our research, we are using forward proofs for the automatic translation of Isabelle formulae to first-order clauses. The translation function is a derived inference rule of type `thm -> thm`.

In contrast to forward proof, backward proof is more goal-oriented and is usually carried out by tactics. As we have seen these tactics require a set of rules when applied to a proof state. In theory, any rule may be given to any tactic. In practice, every rule should be used by only some appropriate tactics to produce meaningful results. Moreover, the restriction on the application of rules to tactics is often determined by the nature of the rules. Applying a rule to an improper tactic may result in a blow up on the number of subgoals or introducing too many unknown variables.

For instance, introduction rules are suitable for the tactic `resolve_tac`, which resolves the conclusion of an introduction rule with one subgoal. This is exactly what a backward chaining will do. In comparison, elimination rules are better suited for `eresolve_tac`, which unifies the premise of an elimination rule with a premise of one subgoal and thus replaces the premise of the subgoal by the conclusion of the rule. Since resolving an

elimination rule with a subgoal is effectively a forward chaining operation on the subgoal's premises, the application of an elimination rule represents forward reasoning in a backward proof.

Although forward and backward proofs are different ways of constructing proofs, they both rely on functions (inference rules or tactics) that derive new theorems (of type `thm`) from existing ones (recall a proof state is also represented by a theorem of type `thm`). These functions are all defined in the kernel so that soundness is guaranteed.

2.6.4 Isabelle's Automatic Proof Tools

Isabelle provides substantial automation to users so that many goals can be proved automatically, without users having to specify the detailed proof steps. This automation is achieved by using Isabelle's classical reasoner and equality reasoner.

The classical reasoner takes a set of rules stored in a *classical set*. The classical set contains a collection of introduction rules and elimination rules, which should be used in a backward chaining style and a forward chaining style respectively. After a lemma is proved, a user may decide it is suitable for forward chaining, and then can declare it as such, and similarly for backward chaining. Afterwards, these rules can be added to the classical set permanently or temporarily during a specific proof. Users can also remove rules from a classical set if necessary. Moreover, some of these rules are *safe* while others are *unsafe*. They affect the behaviour of backtracking in an automatic proof search. When a safe rule is applied to a goal, no information is lost. However, when an unsafe rule is applied, some information is lost: it may reduce the goal to subgoals that are not provable. As a result, if a proof search fails along a path, the classical reasoner needs to backtrack to the most recent unsafe rule applied and continue the proof search on a different path.

The equality reasoner takes a *simplifier set* as input, which has several components: a collection of rewrite rules, simplification procedures, congruence rules and the subgoaler, solver and looper tactics.

A rewrite rule (also known as simplification rule) is usually expressed as a meta-equality theorem: $LHS \equiv RHS$, which means LHS should be replaced by RHS via rewriting. A rewrite rule can also be conditional, in which case it is usually represented by $P \Longrightarrow LHS \equiv RHS$, meaning the rewriting can take place if condition P is satisfied. Similar to classical rules, after an equality is proved, a user can declare it to be a simplification rule and add it to a simplifier set. If the rule added is not an equality, then an Isabelle function translates it into an equivalent equality: it translates any positive atomic formula P into $P \equiv \top$ and translates a negative atomic formula $\neg P$ to $P \equiv \bot$, where $\top$ and $\bot$ represent logical truth and falsity respectively. For instance, a rule

$$\forall n\,[0 < n + 1]$$

which says 0 is less than $n + 1$ for any n, is translated into the simplification rule

$$\forall n\,[0 < n + 1 \equiv \top].$$

Another rule

$$\neg even\ 1$$

meaning number 1 is not even is translated into the rewrite rule

$$even\ 1 \equiv \bot.$$

As we shall see, these boolean equalities should be converted back to their original form, which does not contain $\top$ or $\bot$, before we can translate them to clause form for automatic provers to use.

In contrast to rewrite rules, *simplification procedures* are more flexible since they can create any valid rewrite rules. A simplification procedure can *dynamically* create a rewrite rule according to a pattern of an expression, where the pattern can be very general. The created rewrite rules are usually not stored and are used once only. In addition, one simplification procedure can create many rewrite rules that share some pattern. Therefore simplification procedures can save much effort in defining many rewrite rules.

Congruence rules generate contextual information during simplification. They are meta-equalities of the form

$$\ldots \Longrightarrow f(x_1 \ldots x_n) \equiv\ f(y_1 \ldots y_n)$$

where the assumptions represented by the "..." on the left hand side of implication $\Longrightarrow$ provide information about how to simplify each x argument of f to y argument and all x's and y's are universally quantified. An example of a congruence rule is `imp_cong`:

$$\forall P\ P'\ Q\ Q'\ [(P = P') \wedge (P' \Longrightarrow Q = Q') \Longrightarrow (P \rightarrow Q) = (P' \rightarrow Q')]$$

This rule says that $P \rightarrow Q$ can be simplified to $P' \rightarrow Q'$, if P can be simplified to P' and also if by assuming P', we can simplify Q to Q'.

However, these congruence rules are usually not required by first-order resolution-based automatic provers since such contextual congruence rules are implicit with paramodulation.

The *subgoaler*, *solver* and *looper* are all tactics that attempt to solve subgoals after being simplified by, say, rewrite rules or congruence rules. Users usually do not interact with them directly.

For the classical reasoner and the equality reasoner, Isabelle defines several automatic reasoning tactics that attempt to prove a goal or several subgoals using the rules stored in the classical set and the simplifier set. Examples of these tactics are:

- `simp` is the simplifier and uses the rules stored in a simplifier set. It performs conditional rewriting augmented by other code, including a decision procedure for linear arithmetic.

- `blast` [46] is a sort of generic tableaux theorem prover. It uses any supplied collection of lemmas in the classical set to perform forward or backward chaining, governed by depth-first iterative deepening.

- `auto` is a naive combination of the previous two tactics. It interleaves rewriting and chaining. However, this treatment of equality is primitive compared with that provided by a good resolution prover.

In addition to these, there are many variants of classical reasoning and equality reasoning tactics such as `clarify`, `force` and `fast`.

One advantage of Isabelle's classical reasoner is that it is not restricted to first-order logic. It can prove theorems that cannot easily be expressed in first-order logic at all, such as

$$(\bigcup_{i\in I\cup J} A_i) = (\bigcup_{i\in I} A_i) \cup (\bigcup_{i\in J} A_i).$$

The classical reasoner can also prove many theorems that are difficult for most automatic provers, such as

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

The classical reasoner's and the equality reasoner's other great advantage is that they let the user declare lemmas for them to use, and they easily cope with hundreds of such lemmas without suffering combinatorial explosion. When an interactive prover is used for formal verification, it is typical that users have to prove many intermediate lemmas for later use. The accumulation of many such declarations, which constitutes the knowledge base and pragmatic information, greatly improves the automation available to the user. Therefore, the ability to handle a large lemma library is very important in order to provide automation to users. As we will see, resolution provers usually find it difficult to handle large numbers of axioms.

Although Isabelle's classical reasoner and equality reasoner are powerful and can prove many hard problems automatically, the user has to choose which one is appropriate and invoke it manually. In contrast, resolution and paramodulation based theorem provers can effectively combine classical reasoning and equality reasoning. By integrating with these automatic provers, we can provide Isabelle with greater automation.

2.6.5 Isabelle User Interfaces

Isabelle users carry out theory and proof development in one of the two interfaces provided: the ML interface and the Isar interface. Both of the two interfaces support incremental proof development. Under either of these two interfaces, a user starts a session from some existing theory and then will start interacting with Isabelle by making definitions and carrying out proofs of lemmas. During a proof, the user issues proof commands to Isabelle to prove the goal and Isabelle reads commands line by line and responds to each command. Every time a definition is read by Isabelle or when a lemma is proved, the information is stored in the background context. Both the ML and the Isar interfaces provide the above interaction. However, the major difference between the ML and the Isar interfaces is the way proofs are conducted. Since ML is designed for programming, its proofs are linear. In contrast, Isar is designed for proving, thus its proofs are structured, although it also supports linear proofs.

ML Interface

Working with the ML [43] interface is similar to working with the ML interpreter directly, with extra commands for theory and proof development. In the ML interface, proofs are linear and they resemble the tactic scripts of the HOL system [23] and PVS [40]. A proof

script consists of a sequence of commands that directly manipulate the proof state by applying tactics.

For example, when a user states a proof goal using the `Goal` command, Isabelle echoes the current goal. Subsequently, the proof is carried out by a sequence of tactics. If the user wishes to apply a resolution tactic using the rule `th` on the first subgoal, he may type

```
by(resolve_tac [th] 1)
```

After a tactic is applied, the new subgoals are displayed as a response. However, if the application of the tactic fails then a warning shows up and the proof state is unchanged. During a proof a user needs to specify which tactic should be used and which subgoal the tactic should be applied to. Alternatively, the user can decide to use an automatic reasoning tactic that attempts to refine all of the subgoals. The proof goes on until no more subgoal exists, at which point the proof succeeds.

In addition to commands like `by`, there are many other proof commands. For instance, command `undo` undoes the previous tactic.

Finally, as with tactical scripts, linear proofs are goal oriented but may be hard for human to read.

Isar Interface

The Isar (Intelligible semi-automated reasoning) [67] interface is built on top of Isabelle system by adding a high level proof language as a layer. This high level proof language is called the *Isar formal proof language*, which consists of declarative proof texts. The proof language is interpreted and executed by the *Isar/VM Interpreter*.

In the Isar interface, users can write proofs in a structured way, which resemble the proofs carried out by human beings. Proof texts may consist of typical keywords such as "`from ...`", "`have ...`" and "`thus ...`". Therefore the proofs are more human-readable. Many Isabelle users are now working under the Isar interface.

The Isar proof language is made up of a set of commands. To ensure the proof texts are well structured, a transition semantics of the commands is defined, which then induces a typing of commands. Subsequently, the typing imposes the correctness of the proof text's structure.

The semantics of the Isar proof language commands is defined by a transition of *configurations*. Isar defines three basic configurations, namely **library**, **theory** and **proof**, which respectively represent a collection of theories, an individual theory context and a configuration where a proof is being performed. Any command that triggers a transition between these configurations is given a type accordingly. For example, command `theorem` starts in configuration **theory** and ends in **proof**. Therefore its type is **theory** $\rightarrow$ **proof**.

Most of the Isar language commands are proof commands that take place in the **proof** configuration. In addition, a proof state is always in one of the three *modes*

- **prove** mode allows direct modifications to the current goals, which are usually carried out by the `apply` or `by` command. Proofs are typically linear in this mode.

- **state** mode allows users to state any new claims or assumptions. The current goal cannot be changed in this mode.

- **chain** mode allows users to state any new claims while carrying over the previously proved facts for later proof.

Each proof command can only be executed in a certain mode, which is checked by the Isar/VM Interpreter before execution. For instance, command `show`, which states a proposition to prove, can only be executed in the **state** mode. An execution of a proof command may cause the proof state to enter a different mode. The transition of a proof state gives a finer-grained typing to proof commands. In the example of `show` command, its execution leads a proof state to the **prove** mode; therefore it is given a type **state** $\rightarrow$ **prove**.

In addition to the three proof modes, each proof state also consists of a *proof context* and a *proof goal*. A proof context describes the proof environment of a goal. It contains the current declarations such as assumptions, facts and finished claims. A proof goal contains information about subgoals waiting to be proved. While proof commands are executed, proof context and proof goal are modified.

In addition to writing structured proofs using the Isar proof language, users can also write proofs in a linear style. Isar supports a simulation of traditional linear proofs, which operate in the **prove** mode only. Usually after a linear proof command is executed, the proof state remains in the **prove** mode.

Isabelle/Isar is integrated with, and must work with, Proof General, which provides a graphical user interface. A user types commands and proofs in an editor window to carry out theory and proof developments. When Isabelle reads a command, it displays its response in another window. Proof General also supports ML mode, but Isar mode is more popular.

2.6.6 Object-Logics Supported in Isabelle

Isabelle supports many object-logics. Among them, the most widely used is higher-order logic (Isabelle/HOL [38]), which is also the basis of the HOL system and PVS. It is currently the best developed object-logic and provides an extensive library and packages. It also provides substantial support for formal verification by including large theory developments, such as security protocols and communication protocols. Isabelle/HOL has also been used for mathematical proofs. We will describe Isabelle/HOL in more detail in section §2.7.

Isabelle/ZF [47] (Zermelo-Fraenkel set theory) is untyped and is based on first-order logic. We will explain its syntax in section §2.8.

Some other object-logics supported include FOL (first-order logic), LCF (a version of Scott's Logic for Computable Functions. It is built on top of FOL.), CTT (a version of Martin-Löf's Constructive Type Theory with extensional equality).

2.7 Isabelle/HOL

Isabelle/HOL [38] supports higher-order logic. It is typed and provides *axiomatic type classes*. However, unlike PVS, it does not provide predicate subtyping. Formalizing Isabelle/HOL in first-order logic is a key component of our integration between Isabelle and Vampire/SPASS.

Higher-order logic is also called simple type theory. It extends first-order logic with λ-abstractions and variables over functions and predicates. A type system is integral to higher-order logic: every term should be well-typed. Moreover, in higher-order logic there is no distinction between terms and formulae: a formula is simply a term with type *bool*.

2.7.1 Higher-Order Logic Terms

There are four kinds of terms in higher-order logic, namely *Variables*, *Constants*, λ-*abstractions* and *Function Applications*.

- *Variables* in Isabelle may be fixed or schematic. A schematic variable can be instantiated to any other term with the same type.

- *Constants* include any previously defined values, functions or predicates. For instance, the truth values $\top$ and $\bot$ are constants. The successor function and operations such as $+$, $-$, $\times$ and $/$ for natural numbers are constants. Other commonly used constants include the logical operators such as $\wedge$ (and), $\vee$ (or), $\neg$ (negation), $\equiv$ (equivalence), $\rightarrow$ (implication), $=$ (equal) and Hilbert's ϵ operator.

- λ-*abstractions*, also called λ-terms, represent functions. They are written as $\lambda x.\, t(x)$, where x is the binding variable and term $t(x)$ is the body. It represents a function, which when given x, returns $t(x)$. A λ-term can be nested as $\lambda x.\, \lambda y.\, t$, which is usually abbreviated as $\lambda x\, y.\, t$. A convention is followed, where the "." of the λ-term covers the scope that goes to as far right as possible. Sometimes the "." is omitted as the scope of a quantifier can be indicated by brackets.

- *Function Applications* have the form of $t_1\, t_2$, where both t_1 and t_2 are terms and t_1 is the function and t_2 is the argument. A sequence of function applications, such as $t_1\, t_2 \ldots t_n$ is allowed, where function application associates to the left.

Universal quantifier $\forall$ and existential quantifier $\exists$ are constants and are regarded as *binders*. Formulae $\forall x.\, P(x)$ and $\exists x.\, P(x)$ are syntactic sugar of $\forall(\lambda x.\, P(x))$ and $\exists(\lambda x.\, P(x))$ respectively.

2.7.2 Higher-Order Logic's Type System

Higher-order logic is very powerful, thus a type system is required to avoid any unsoundness, such as the Russell's paradox. A type system has a collection of types, where each type represents a set of values. For instance type *nat* represents all natural numbers, whereas type *bool* denotes a set with two values $\{\top, \bot\}$.

A type can be an *atomic type* (such as *nat* and *bool*) or a *compound type*, which is built up from atomic types using *type constructors*. A compound type is usually written as $(\tau_1, \ldots, \tau_n)op$, where *op* is a type constructor. It constructs the compound type from the types $\tau_1, \ldots, \tau_n$, which may be either atomic or compound types. An example of a compound type is a function type $\tau_1 \rightarrow \tau_2$, where the function type constructor $\rightarrow$ is written infix.

To allow greater flexibility and function reuse, *polymorphism* has been introduced into higher-order logic, by Isabelle and the HOL system. A type is *polymorphic* if it has a

type variable, which can be instantiated to a specific type. With the introduction of polymorphism, a higher-order term can now have a polymorphic type. A simple example is the identity function $\lambda x.\, x$, which when given an argument, returns that argument as the result. Its type is polymorphic $\alpha \to \alpha$ and can be instantiated to instances such as $bool \to bool$ or $nat \to nat$.

Similar to the concept of fixed and schematic variables, Isabelle distinguishes fixed and schematic type variables. Only schematic type variables may be instantiated to some other types.

2.7.3 Well-Typed Higher-Order Logic Terms

For soundness reason, higher-order logic requires each of its terms to be *well-typed* based on a given type system. A term is well-typed if we can assign a type to it and each of its subterms in a consistent way (meaning no conflict) as follows

- A variable is assigned a type τ, which can be polymorphic.
- A constant is assigned a type τ when it is defined. This type can be polymorphic. Occurrences of the constant can have different instances of τ.
- $\lambda x.\, t$ is well-typed if there are some types τ_1 and τ_2, such that $\lambda x.\, t$ is assigned $\tau_1 \to \tau_2$, and x, t are assigned τ_1 and τ_2 respectively.
- A function application $t_1\, t_2$ is well-typed if there are some types τ_1 and τ_2, such that $t_1\, t_2$ is assigned τ_2, and t_1 and t_2 are assigned $\tau_1 \to \tau_2$ and τ_1 respectively.

A higher-order logic term t with type τ is written as $t : \tau$. However, such type information is usually omitted from an expression as a type inference algorithm can be used to infer the type of a given higher-order logic term.

2.7.4 Axiomatic Type Classes in Isabelle/HOL

Isabelle/HOL's type system supports *axiomatic type classes* [66]. Axiomatic type classes generalize polymorphism. Before looking at how it has been formally defined in Isabelle, we look at the benefits of using it.

First, axiomatic type classes allow meaningful overloading of polymorphic operators and theorems. For example, the $\leqslant$ relation is polymorphic and can be applied to types such as natural numbers or real numbers. Moreover, a requirement on the application of $\leqslant$ is that these types must satisfy a property: they must be partially ordered. To implement this restriction on the overloading of $\leqslant$, we can define a type class called *partial order* with the partial ordering property as an associated axiom. Any type that satisfies this axiom is an *instance* of class *partial order* and hence belongs to it. Subsequently, whenever we need to overload the $\leqslant$ operator on some type, we only need to check if the type is an instance of type class *partial order*. This also simplifies the procedure of adding a newly defined type, since it is sufficient to prove the type satisfies the axioms of the class in order to show it is indeed an instance of the class.

Second, we can use an already defined type class to derive a new class by adding additional axioms that have to be satisfied. Taking the *partial order* type class as an

example, if we add an axiom about linear ordering, then we will have a new type class *linear order*. The newly derived type class is a *subclass* of the original type class (the *superclass*). Therefore any axiom that holds in the superclass also holds in the subclass. Moreover, any instance of a subclass automatically belongs to the superclass. This allows convenient construction of type class hierarchies.

Third, we can use axiomatic type classes to prove theorems. Instead of proving a theorem for a particular type, we can now prove a theorem holds for a type class. Once this is proved, it holds for all instances of that class automatically, including any types defined in future Isabelle sessions.

Having seen the benefits of axiomatic type classes, we now look at how Isabelle/HOL defines them [66].

Definition 4. Isabelle's axiomatic type classes are defined as follows:

- A *type class* is a set of types for which certain operations are defined. An *axiomatic* type class has a set of axioms that must be satisfied by its instances. If a type τ belongs to a class C then it is written as $\tau :: C$.
- A type class C is a *subclass* of another type class D, if all axioms of D can be proved in C. If a type τ is an instance of C then it is an instance of D as well. Furthermore, a type class may have more than one direct superclass.
- A *sort* is an intersection of type classes. If C is a subclass of both D_1 and D_2 then C is subset of the intersection of D_1 and D_2, and thus has sort D_1 and D_2.
- Each type constructor has one or more *arities*, which describe the type class constraints on the type constructor's arguments and its result. For a compound type $(\tau_1, \ldots, \tau_n)op$, an arity of type constructor op is written as $op :: (C_1, \ldots, C_n)C$, where $C_1, \ldots, C_n$ and C are type classes: $C_1, \ldots, C_n$ are type classes of op's arguments and C is the type class of op's result.

In addition, type classes are open-ended, which means any new type that satisfies the axioms can be admitted to the class.

In order to make the idea more concrete, let us look at an example. Isabelle/HOL defines the type class of linear orders (`linorder`) to be a subclass of the type class of partial orders (`order`). This is written in Isabelle as

```
axclass linorder < order
  linorder_linear:  "x≤ y ∨ y≤ x"
```

where `linorder_linear` is the axiom — in addition to axioms of `order` — that has to be satisfied by instances of `linorder`. Now, to assert that type `real` is an instance of class `linorder`, we must show that the corresponding instance of the axiom `linorder_linear` holds for that type. This is written in Isabelle as

```
instance real :: linorder
  proof ... qed
```

where `proof ... qed` is the proof of the axiom `linorder_linear`, which is not shown here.

Axiomatic type classes not only ensure meaningful overloading, but also save efforts in theorem proving. We can prove a theorem such as $(-a) \times (-b) = a \times b$ in type class `ring` and declare it as a simplification rule, where it will govern numeric types such as `int`, `rat`, `real` and `complex`, which are all instances of `ring`. Type checking remains decidable, for it is the user's responsibility to notice that a type belongs to a certain class and to declare it as such, providing the necessary proofs. Of course, full type checking must be performed, including checking of sorts, since a theorem about linear orderings cannot be assumed to hold for arbitrary orderings.

Isabelle provides compound types through the use of type constructors. As we have seen, the arities of type constructors describe the type class information of the arguments and the result of this type constructor. Take the Isabelle's list type constructor `list` as an example. A list can be linearly ordered (by the usual lexicographic ordering) if we have a linear ordering of the list element types. This type constructor declaration is written in Isabelle as an arity

```
instance list :: (linorder) linorder
  proof ... qed
```

In addition, each type constructor may have multiple arities. For instance, `list` has another arity

```
instance list :: (order) order
  proof ... qed
```

Finally, polymorphic functions and predicates can be instantiated as usual, with extra type class constraints: a type variable must be instantiated to a type belonging to the same type class. In our type class example, the relation $\leqslant$ is polymorphic. Its type is $\alpha \to \alpha \to bool$, where α is a type variable of class `ord`. The effect is to allow $\leqslant$ to be applied only if its arguments belong to type class `ord`. This polymorphic type can be specialized when $\leqslant$ is applied to different arguments. When applied to sets, its type will be $\alpha\ set \to \alpha\ set \to bool$, whereas for natural numbers its type will be $nat \to nat \to bool$.

2.8 Isabelle/ZF

Isabelle/ZF [47] implements Zermelo-Fraenkel set theory. It is based on Isabelle's classical first-order logic Isabelle/FOL and is untyped.

2.8.1 Zermelo-Fraenkel Set Theory

In set theory every mathematical concept is defined by sets, which are the only fundamental objects. A fundamental relation in set theory is the membership relation, which determines a set by its elements. For example, when examining whether two sets are equal, we check if they have exactly the same elements. Based on this membership relation, other operations can be defined, such as set unions, intersections and unordered pairs.

There are several axiomatic approaches to set theory. In an axiomatic approach, axioms are required to assert that the above set operations are possible. One of the most influential axiomatic approaches to set theory is the Zermelo-Fraenkel (ZF) set theory,

which is expressed in first-order logic. The existence of the set operations are asserted by several axioms and axiom schemes, namely *axiom of extensionality*, *axiom of the unordered pair*, *axiom of union*, *axiom of power set*, *axiom of empty set*, *axiom of infinity*, *axiom of foundation*, *axiom scheme of separation* and *axiom scheme of replacement*.

Because of the axiom schemes, ZF set theory defines an infinite number of axioms. Based on the axioms and set construction operations, we can define integers, real numbers, functions and other more complex mathematical concepts.

2.8.2 ZF Set Theory in Isabelle

ZF set theory is formalized in Isabelle as Isabelle/ZF. It is well developed with a significant number of theories for proof developments. Moreover Isabelle/ZF also provides packages for handling inductive and co-inductive definitions.

Every ZF object is defined in terms of sets and set construction operations. This is not practical in real proofs. Therefore Isabelle/ZF defines a collection of constants for primitive sets. For instance, Isabelle defines constant `Collect` that constructs a set based on the axiom scheme of separation. An infinite set `Inf` is defined in correspondence to the axiom of infinity. Other primitive constants defined include `Pow` for power set, `Union` for union of sets, `Upair` for unordered pairs and `succ` for successor of sets.

Isabelle/ZF also defines terms that are outside the scope of first-order logic. Examples are general unions $\bigcup_{x\in A} B[x]$, general intersections $\bigcap_{x\in A} B[x]$, general sums $\sum_{x\in A} B[x]$, general products $\prod_{x\in A} B[x]$ and λ-terms.

Since it is usually hard to prove any theorem using definitions only, Isabelle/ZF provides many derived rules, used in a natural deduction style. Furthermore, these rules can be used by the classical reasoner for an automatic proof search. As a result, Isabelle users usually carry out proofs using the derived rules rather than the constants' definitions.

Finally, unlike Isabelle/HOL, Isabelle/ZF does not employ polymorphism. It has a simple type system, where ZF terms are typed i (for individuals) and formulae are typed o. For example, the constant `succ` has type $i \to i$ while the membership constant $\in$ has type $i \to i \to o$. Furthermore, a function in Isabelle/ZF is represented as a set of pairs and thus its type is i rather than $i \to i$.

2.9 An Example Proof in Isabelle

Before we leave this section, we look at an Isabelle proof example. Our goal is to prove that for any two lists xs and ys, the sum of their lengths is equal to the length of the concatenation of those lists. The goal is written in Isabelle/HOL as

```
length (xs @ ys) = length xs + length ys
```

where `length` returns the length of an input list and `@` is the list concatenation operator.

We show the proofs in both the ML interface and the Isar interface. As with many recursively defined datatypes, we first apply induction on list `xs`, which reduces the goal to two subgoals. Afterwards, we call Isabelle's classical reasoner and simplifier to prove each subgoal. As we will see, both interfaces support linear proofs. In the ML interface, linear proofs are carried out by tactics directly, whereas in the Isar interface linear proofs are carried out by the simulations of tactics. For instance, the tactic that uses induction

is called `induct_tac` in the ML interface but its simulation in the Isar interface is called `induct`.

2.9.1 ML Interface

In the ML interface, a goal is declared using the `Goal` command. A sequence of tactics is carried out using the `by` command. The proof is interactive and the response from Isabelle is also shown. Please note: Isabelle only displays the current subgoals, rather than the entire proof state.

```
> Goal "length (xs @ ys) = length xs + length ys";          ⟵state a goal
Level 0 (1 subgoal)
length (xs @ ys) = length xs + length ys
 1.  length (xs @ ys) = length xs + length ys
val it = [] :  Thm.thm list                                 ⟵original problem
> by(induct_tac "xs" 1);
Level 1 (2 subgoals)
length (xs @ ys) = length xs + length ys
 1.  length ([] @ ys) = length [] + length ys
 2.  !!a list.
  length (list @ ys) = length list + length ys
  ⟹length ((a # list) @ ys) = length (a # list) + length ys
val it = () :  unit                                ⟵base case and induction step
> by(Auto_tac);
Level 2
length (xs @ ys) = length xs + length ys
No subgoals!                                                ⟵proved by Auto_tac
```

2.9.2 Isar Interface

In the Isar interface, this goal is declared using the `lemma` command, which initializes a proof state in the **prove** mode. Command `proof(induct xs)` refines the proof goal using induction and takes Isabelle proof into the **state** mode. A case analysis on each subgoal is performed afterwards: first with the base case when xs is an empty list (`Nil`), then with the inductive step.

```
lemma length_append [simp]:  "length (xs @ ys) = length xs + length ys"
proof(induct xs)                                      ⟵apply induction on xs
 case Nil thus ?case by auto                        ⟵prove base case by auto
next
 case(Cons a list) thus ?case by auto        ⟵prove inductive step by auto
qed
```

These are the proof texts that we type in the Isar editor window. The Isar response window is not shown here.

Chapter 3

Using Resolution to Prove Isabelle Problems

Our integration aims to improve the automation of Isabelle by using resolution-based automatic theorem provers to automatically find proofs of Isabelle's goals. One important task involved is to design a method that can efficiently bridge the many differences between these two types of provers, which is a prerequisite of the integration. In this chapter, we carry out an investigation on the major obstacles that need to be tackled and then describe how we have solved the problems and our approach of linking Isabelle with our target automatic provers. In particular, we put emphasis on the translation of Isabelle formalism to first-order clause form, which is the major component of the integration.

We have carried out a series of experiments in order to examine whether our integration approach is practical: it should not only work in theory but also enable an automatic prover to find a proof within a reasonable amount of time. We also used the experiments to determine the feasibility of using resolution to support Isabelle proofs. We list the experimental results and offer some discussion at the end of this chapter.

3.1 Initial Investigation

Integrating any two existing systems with significant differences is hard: we cannot expect any of the systems to undergo major change to suit the other one. Instead we need to design an intermediate program that runs between them and acts like a translator to bridge any differences involved, so that neither the developers of the provers, nor the users will be affected.

Isabelle is very different from any of the automatic provers that we intend to use in our integration. The first task that we need to carry out before we can design and implement any intermediate system is to investigate the obstacles brought by those differences. Subsequently we need to tackle each problem in turn and design a suitable method to act between Isabelle and automatic theorem provers. We also need to evaluate the effectiveness of our method.

3.1.1 Identifying the Major Obstacles

Our research mainly concerns integrating Isabelle with Vampire and SPASS, because they are leading resolution provers that have done well in recent CASCs (the CADE ATP System Competition)[1]. We have given some background information about these two provers in a previous chapter (Chapter 2). In order to best present the major problems that our integration faces, we list some of the differences between Isabelle and the resolution provers. We also describe the associated problems that we need to tackle. Furthermore, the research on linking these two automatic provers with Isabelle is only a starting point to achieving the ultimate goal, which is to use any resolution-based automatic prover to support Isabelle proofs. Therefore, many of the differences and problems described below are not specific to Vampire and SPASS, but are generally applicable to many other automatic provers.

Isabelle Logic and First-Order Logic

Isabelle's meta-logic is a fragment of higher-order logic and it supports a variety of object-logics. In contrast, most resolution automatic provers, such as Vampire and SPASS, only work for first-order logic. Therefore, one task of our integration is to translate Isabelle formalisms to first-order logic.

There are two possible approaches to this translation. The first option is to separately translate each object-logic supported by Isabelle to first-order logic. The other alternative is to treat each object-logic as embedded inside the meta-logic and instead of translating each object-logic, we translate the meta-logic to first-order logic. The latter approach seems more systematic and scalable than the first one as all object-logics can be dealt with in a uniform way. This may all work very well in theory, but when it comes to reality, we also need to consider an efficiency issue.

The second approach does not exploit any object-logic-specific information: it regards all object-logic operators as purely constants defined in the meta-logic. Therefore the translation of an object-logic rule or goal is literal. As we know, Isabelle's meta-logic is higher-order. It is very likely that an object-logic represented by this meta-logic may contain terms that are outside the scope of first-order logic. As a result, if we translate such non-first-order terms, we will have to translate higher-order expressions to first-order form. Nonetheless, if we have a closer look at some of the object-logic expressions, we may find many of them can be reformulated to an equivalent first-order formula. This optimized translation requires some object-logic-specific knowledge, which is lost if we treat all object-logic as part of the meta-logic. Although we can translate the entire higher-order logic to first-order logic with a bit more effort, the resulting first-order terms may be too complicated for an automatic prover to handle efficiently. This may result in either a slow and lengthy proof or a failed proof attempt. In any case, the second approach is likely to hinder automatic provers' performance.

Based on the consideration above, we have decided to formalize each Isabelle object-logic in first-order logic separately. We also take into account of the specific knowledge available to the object-logics. For many object-level formulae that are not originally in first-order form, we convert them to equivalent first-order logic formulae.

[1]See http://www.cs.miami.edu/tptp/CASC/

Typed and Untyped Systems

Isabelle is a typed interactive prover and provides a sophisticated type system. In comparison, most first-order automatic provers are untyped. A question that we need to consider is whether we should keep Isabelle's type information when translating Isabelle's typed formulae to first-order input of automatic provers.

A quick answer to this question may be that it is not necessary. When we send an Isabelle goal and some theorems to an automatic prover, types do not have to participate in the proof search. Adding types in first-order formulae will inevitably result in bigger terms. This may slow down the proof search as bigger terms take longer time to process. Although proving higher-order problems without considering type constraints may yield an incorrect proof, which is the reason why a type system is required for higher-order logic, the proof found by an automatic prover is always checked by Isabelle before being accepted. These arguments all seem to suggest we can completely ignore types in our translation and treat Isabelle formulae as untyped. However, a careful analysis reveals an opposite argument.

Many goals and theorems in Isabelle contain polymorphic predicates and functions. For instance, the $\leqslant$ relation is a polymorphic predicate that can be overloaded on integers or sets, etc. When an automatic prover tries to prove a goal containing $i \leqslant j$ for two variables i and j, the type of $\leqslant$ will help the prover to decide which theorem is applicable. Without this type information, it is likely that the automatic prover will apply an incorrect theorem. This will significantly slow down the proof search. Furthermore, although Isabelle can always check a proof found by an automatic prover and reject it if necessary, such several-rounded communications between them may also affect the performance of our integration. This strongly suggests that we should convey the type information to automatic provers.

Finally, since our integration design should be flexible enough so that we can add any automatic provers to assist Isabelle proofs, we cannot assume any special support from them for Isabelle's type system. Consequently, we have to formalize Isabelle's type system and typed formulae purely inside first-order logic.

Natural Deduction Calculus and Resolution Theorem Proving

Isabelle's logical framework works in the natural deduction style. It provides both equality and classical reasoning. Theorems are distinguished by whether they should be used for forward chaining (for elimination rules) or backward chaining (for introduction rules). This information about how theorems should be used is essential when Isabelle's classical reasoner performs automatic proof search: using a theorem incorrectly will result in a failed proof attempt.

On the other hand, resolution provers work by refutation of a problem expressed in clause normal form, although arbitrary formulae are now accepted by most of the provers as well. The inference system is based on resolution and paramodulation. Performance of a proof search is largely affected by the clause selection function and the literal selection function. However, there is no direct support of Isabelle's notion of backward chaining or forward chaining.

These differences in their proof calculi lead to the following questions:

- Can resolution and paramodulation effectively combine the functions provided by

Isabelle's classical and equality reasoners? In theory, the answer is yes. However, we still need to examine whether it is feasible.

- Do we need to convey the information about the forward and backward use of Isabelle theorems to an automatic prover? Without such information, the search space of an automatic prover may easily explode if wrong literals are selected for resolution. Subsequently, we need to consider how we can encode this information in the input sent to the prover.

- How can we translate a resolution proof found by an automatic prover to an Isabelle proof so that it can be verified? This requires us to perform clause normal form transformation inside Isabelle logic using inference rules, which is certainly harder than sending Isabelle formulae directly to the automatic prover.

Number of Lemmas

Isabelle's classical reasoner can utilize a large set of lemmas without suffering a combinatorial explosion. Although certain proofs do require the user to name crucial lemmas, hundreds of other lemmas are available to the classical reasoner at all times. These are lemmas that were previously designated as being useful for classical reasoning, and they constitute a knowledge base for the user's application domain. Typically omitted from this knowledge base are transitivity laws and similar lemmas that would blow up the search space. As our aim of integrating Isabelle with automatic provers is to reduce interaction, we should preserve this advantage: the user should only have to identify a few crucial lemmas, while the resolution search automatically finds other needed facts from the knowledge base.

Resolution- and paramodulation- based automatic provers are very powerful and have been able to solve very complex problems. However, they mainly work in a problem domain where not many axioms are involved. Therefore it is important to assess their performance of proof search in the presence of large numbers of axioms.

Interactive and Automatic Modes

Isabelle is an interactive prover: most of the time, a user is responsible to guide the prover on proof search. A resolution prover is completely automatic: it is difficult to influence the direction of its proof search once it starts running. Its behaviour is only affected by its numerous settings, which can be specified manually.

The settings of automatic provers can dramatically affect their performance, in the form of the proof speed and the proof script's length. Therefore, correct settings or combinations of settings are keys to the success of our integration. Moreover, we need to study those settings in advance and hardwire them in our integrated system rather than asking Isabelle users to specify the values of those settings. From users' point of view, the prover process should be invisible and we should not ask users to decide what settings are suitable for their problem. Isabelle users do not possess the technical knowledge of resolution provers.

A quick response from an automatic prover is important. Recall that an automatic prover should be called to prove a goal in the background while a user is trying to find

a proof himself. A quick proof found by the automatic prover can help the user tremendously. However, a slow proof search on the goal is not useful: if an automatic prover spends an excessive amount of time on a goal, then by the time the result is found and returned to the user, the user may have already accomplished the proof himself. The requirement that the background automatic prover should find a proof quickly is, in fact, consistent with the situation in which it is used: it is used to support interactive prover with near-real-time response requirement.

However, as we know, most resolution provers are designed to run for minutes or hours on batch jobs. Therefore a question we need to ask is how fast can these provers find proofs. We need to use experiments to measure their performance.

3.1.2 Research Outline

After the initial investigation on our research, we decided to answer the questions above and solve the problems by experimenting on formalizing Isabelle/ZF and Isabelle/HOL in first-order logic. We also performed a series of experiments on both Vampire and SPASS to assess our method of formalization and to evaluate how feasible it is to use resolution-based provers to assist Isabelle proofs. Since we received greater support from the Vampire team at the time of experiments, we carried out most of the experiments on Vampire.

Isabelle/ZF is untyped and is an extension of first-order logic. It is simpler than many other object-logics and is a good starting point of our formalization. The primary objective of experiments on ZF was to examine the performance of Vampire and in particular, to find those Vampire settings that were suitable for us to use.

Isabelle/HOL implements higher-order logic. While formalizing it in first-order logic, our major emphasis was on HOL's type system. Experiments on HOL were mainly used to determine whether our type encoding was practical and whether it was useful to keep type information in first-order clauses generated from Isabelle's formulae.

We describe our formalization and experiments in the following sections.

3.2 Formalizing Isabelle/ZF in First-Order Logic

Isabelle/ZF [47] implements Zermelo-Fraenkel set theory and is based on first-order logic. Its type system does not employ polymorphism and is very simple: ZF terms are typed i (for individuals) and formulae are typed o. Therefore we can simply treat it as an untyped logic. As a result, the main concern of our research on formalizing Isabelle/ZF in first-order logic is to find an effective way to translate its formulae and terms to first-order clauses.

For those ZF formulae (theorems and goals) that are already in first-order logic form, we translate them directly to clauses via clause normal form transformation. However, set theory contains some other terms and some special ways of expressing formulae, which are outside the scope of first-order logic. As remarked above, we should use logic-specific knowledge to reformulate those formulae and terms before translating them to clause form.

3.2.1 Elimination Rules

The first problem that we encountered during our translation research was brought about by the way Isabelle/ZF (and also Isabelle in general) formulates its elimination rules: they are represented in higher-order form.

Recall Isabelle represents all elimination rules with the following format (§2.6.2)

$$\forall P\,[A \to \forall \mathbf{x}_1\,(\mathbf{B}_1 \to P) \to \ldots \to \forall \mathbf{x}_n\,(\mathbf{B}_n \to P) \to P].$$

Clearly, this is not a first-order formula at all due to the predicate variable P. We cannot simply translate a formula like this to clauses. It might work if we design a method that translates higher-order logic to first-order logic and use it for our translation. However, many elimination rules do not involve any other higher-order function or predicate, except the predicate variable P. In fact, these elimination rules encode first-order logic inference rules. We have found a simple reformulation of an elimination rule, which should be applied before clause normal transformation.

Definition 5. For an Isabelle elimination rule

$$\forall P\,[A \to \forall \mathbf{x}_1\,(\mathbf{B}_1 \to P) \to \ldots \to \forall \mathbf{x}_n\,(\mathbf{B}_n \to P) \to P],$$

we transform it into the following equivalent first-order formula

$$A \to (\exists \mathbf{x}_1\,\mathbf{B}_1 \vee \ldots \vee \exists \mathbf{x}_n\,\mathbf{B}_n).$$

In the case when $n = 0$, the elimination rule is simply $\neg A$.

As we can see, this reformulation of elimination rules has successfully removed the predicate variable P. For example, the set intersection elimination rule (`IntE`) is represented in Isabelle as

$$\forall c\,A\,B\,P\,[c \in A \cap B \to (c \in A \wedge c \in B \to P) \to P].$$

This says if $c \in A \cap B$ then c is in both A and B. Using our method, we transform it to an equivalent first-order formula ($\cap E$)

$$\forall c\,A\,B\,[c \in A \cap B \to (c \in A \wedge c \in B)].$$

We now give a brief justification of our translation, using the example of `IntE`. The first formula (`IntE`) is the elimination rule written in the natural deduction form and we need to unfold it in order to eliminate the predicate variable P. Although the predicate P disappears, we are not losing information during a proof. Suppose we are going to prove a goal

$$(i \in S_1 \cap S_2) \to R_1 \to R_2 \ldots \to R_n \to Q \tag{3.1}$$

where $R_1 \ldots R_n$ and Q are formulae. In addition, $(i \in S_1 \cap S_2)$ and each R_i are assumptions of the goal and Q is the conclusion.

In Isabelle, this goal is proved by applying the set intersection elimination rule `IntE`. First, $c \in A \cap B$ is instantiated to $i \in S_1 \cap S_2$, and P to Q (c, A, B, P are universally quantified variables). Then the goal (3.1) is replaced by a new goal

$$R_1 \to R_2 \ldots \to R_n \to (i \in S_1 \wedge i \in S_2) \to Q. \tag{3.2}$$

Now, if we use resolution to prove the new goal (3.2), then we will have to negate it and convert it to clauses — suppose this set of clauses is called $\mathbf{C}$. On the other hand, if we use resolution to prove the original goal (3.1) directly, with the use of transformed elimination rule $\cap E$, then we need to convert the negated goal to clauses as $\mathbf{C}_1$. We also need to convert $\cap E$ to clauses as $\mathbf{C}_2$. It can be shown easily that by resolving clauses from $\mathbf{C}_1$ and clauses from $\mathbf{C}_2$, we will get the same set of clauses as $\mathbf{C}$. As far as resolution is concerned, there is no difference in terms of the clauses generated (here clauses $\mathbf{C}$ in both cases), regardless whether we use an original Isabelle elimination rule or its equivalent first-order formula that we have described above. Therefore, our transformation on Isabelle elimination rules is correct.

We have used this method to reformulate all Isabelle elimination rules. For instance, we translate the elimination rule `domainE`

$$\forall a\, r\, P\, [a \in \mathit{domain}(r) \rightarrow \forall y\, (\langle a, y\rangle \in r \rightarrow P) \rightarrow P]$$

to

$$\forall a\, r\, [a \in \mathit{domain}(r) \rightarrow \exists y\, (\langle a, y\rangle \in r)].$$

3.2.2 Transforming Other Isabelle/ZF Terms

Isabelle/ZF contains many other operations expressed in the special set theory syntax, which are not present in first-order logic. Examples include general union $\bigcup_{x\in A} B(x)$ and general intersection $\bigcap_{x\in A} B(x)$. Isabelle's meta-logic represents these terms using higher-order terms such as the λ-abstractions. A straightforward approach to formalize these non-first-order ZF terms would be to translate higher-order constructs (§3.1.1). However, these terms serve as typical examples where the semantics of a logic can help to optimize logic formalization.

In Zermelo-Fraenkel set theory, any formula $\phi(Z)$, which contains a free occurrence of a term Z, is equivalent to

$$\exists v\, [\phi(v) \wedge \forall u\, (u \in v \leftrightarrow u \in Z)]$$

where v is a fresh variable and $\phi(v)$ is the result of replacing Z by v. The interpretation of this equivalence is that $\phi(Z)$ is true if and only if there is some term v, which satisfies ϕ and v contains exactly the same elements as Z does. Hence, by the axiom of extensionality, Z and v are equal.

This transformation allows any occurrence of the term Z to be forced into a context of the form $u \in Z$. Consequently, if we let $\bigcap_{x\in A} B(x)$ be Z, then by substitution, a formula $\phi(\bigcap_{x\in A} B(x))$ is equivalent to

$$\exists v\, [\phi(v) \wedge \forall u\, (u \in v \leftrightarrow u \in \bigcap_{x\in A} B(x))].$$

Such transformation is helpful as we can further translate $u \in \bigcap_{x\in A} B(x)$ into

$$\forall x\, [x \in A \rightarrow u \in B(x)] \wedge \exists a\, [a \in A]$$

which completes the translation of any formula $\phi(\bigcap_{x\in A} B(x))$ into first-order logic.

We translate a term with general union $\bigcup_{x\in A} B(x)$ into first-order logic in a same way, except that we replace $u \in \bigcup_{x\in A} B(x)$ using the following equivalence

$$u \in \bigcup_{x\in A} B(x) \simeq \exists x\,[x \in A \wedge u \in B(x)]$$

Finally, Isabelle/ZF contains bounded quantification on variables. For instance, a formula

$$\exists x \in A\,[P(x)]$$

says there is some x in the set A, such that x satisfies P. We reformulate this bounded quantification to normal first-order quantification using the equivalence

$$\exists x \in A\,[P(x)] \simeq \exists x\,[x \in A \wedge P(x)].$$

In addition, a formula

$$\forall x \in A\,[P(x)]$$

means for any x that is in set A, $P(x)$ is true. We reformulate this formula using the equivalence

$$\forall x \in A\,[P(x)] \simeq \forall x\,[x \in A \rightarrow P(x)].$$

These transformation of terms and formulae require some logic-specific knowledge (in this case, their interpretations). We find such knowledge can help improve the formalization of the logic in first-order form.

3.2.3 Efficiency Issues of Translation

In addition to correctly translating Isabelle/ZF formulae into first-order form, we also need to consider whether the translation is practical, which is measured by the performance of an automatic prover when it tries to prove goals using our translation.

There may be more than one way to translate a ZF formula into first-order form. Even for those ZF formulae that are already in first-order logic form, we may have a choice in whether to convert it to clause form directly or to transform the formula into another equivalent first-order formula first and then perform the clause normal transformation. These representations are all correct, but one may take much more time than another for an automatic prover to prove. We should aim to find a most suitable translation so that first-order clauses generated from that translation can be proved by automatic provers within a reasonable amount of time. We have carried out many experiments in order to measure performances among several representations of ZF formulae. Several interesting problems have emerged from the experiments.

One example is the subset relation $R \subseteq S$ between two sets. Since it is already in first-order logic, we can translate it to first-order clauses directly. Alternatively, we can translate it to its equivalent membership relation $\forall x\,(x \in R \rightarrow x \in S)$. From the experiments (§3.3) that I have carried out on Vampire, it is clear that Vampire can find a proof much more quickly if the subset relation is replaced by its equivalent membership relation. This is probably because during most of the complex proofs in set theory, subset relations have to be reduced to equivalent membership relations anyway.

Set equality $A = B$ is another example. During many proofs, set equality predicates should be reduced to two subset predicates by resolution: $A \subseteq B$ and $B \subseteq A$. However,

Vampire usually gives the positive equality literal a low priority relative to other literals in the same clause. Therefore the positive equality literal is likely to be selected and resolved last during resolution. As a result, the positive equality in the clause

$$\neg(A \subseteq B) \vee \neg(B \subseteq A) \vee (A = B)$$

is not selected to participate in a resolution step. This makes the reduction of set equality to subset relations via resolution impossible. Our experiments show that better performance can be achieved when we replace set equality predicates by the subset predicates, which will be further reduced to formulae involving membership predicates as shown above.

3.3 Experiments on Formalizing Isabelle/ZF

After having formalized Isbelle/ZF in first-order logic, we decided to run some experiments with the following objectives:

- Determine whether it is feasible to use resolution and paramodulation methods to assist Isabelle proofs by improving its automation.
- Evaluate the effectiveness of our translation from Isabelle/ZF to first-order logic. Although the translation is correct, we need to ensure it is practical.
- Study the use of automatic provers and identify suitable settings for our problems.
- Examine the performance of automatic provers in the presence of large numbers of axioms.

Since automatic provers will be used to prove Isabelle goals, the experimental results will be more convincing if we run those provers on proof goals taken from Isabelle theory files, rather than problems in some other domain (such as those stored in the TPTP library). Therefore, for our experiments, we took proof goals from the following Isabelle theories:

- `equalities.thy`: proofs of many simple set equalities.
- `Comb.thy`: a development of combinatory logic similar to the Isabelle/HOL version described by Paulson [44].
- `PropLog.thy`: a development of propositional logic.

In order to test whether resolution provers are suitable candidates for our integration, with improved automation as the ultimate goal, we need to look at the performance of automatic provers from two aspects.

First, we need to examine whether automatic provers can prove goals that were originally proved by Isabelle's automatic tactics or a combination of tactics. Resolution and paramodulation-based provers should in theory combine Isabelle's equational and classical reasoning, which need to be performed separately in Isabelle. For this purpose, we carried

out experiments, which consisted of taking `blast`, `fast`, `clarify`, `auto` and `simp` invocations from existing proofs taken from the three theory files and attempting to reproduce proofs using several automatic provers, including Vampire and SPASS (§3.3.1).

Moreover, it is not good enough if an automatic prover can only prove those goals that were proved by Isabelle's built-in tools, since our aim of integration is to use automatic provers to prove many more goals that are too complicated to be proved by any Isabelle tactic. Therefore, running experiments to test the potential improvement in automation is even more important. As a result, we carried out another set of experiments, using some goals that were not proved by `blast` or `auto`, etc. (§3.3.2).

We included large numbers of axiom clauses in some parts of the experiments to investigate how well automatic provers can handle them. In addition, we carried out experiments on two particular issues concerning subset relations and the performance of proving equalities by Vampire.

As we ran most of our experiments on Vampire, we describe experimental results received from Vampire in the rest of this section. Moreover, as our integration will only let automatic provers run for a limited duration of time, we set the time limit for each proof attempt to 60 seconds, which is roughly the amount of time a user may spend on considering how to construct the rest of a proof manually.

While we ran the experiments, for each proof, a set of classical rules (backward and forward chaining rules) and equality rewriting rules (simplification rules) in the current Isabelle context were translated to first-order axiom clauses. The goals were negated, converted to conjecture clauses and sent to Vampire. Moreover, each Isabelle goal usually presented more than one proof goal to Vampire. In some of the examples, we also performed *formula renaming* [39] before the clause normal form transformation in order to minimize the number of clauses. We tried several Vampire settings on these experiments and the findings we received are valuable for future use.

3.3.1 Experiments on Combining Isabelle's Tactics

We carried out two runs of experiments in order to examine whether Vampire can prove goals that were proved by Isabelle's built-in tactics: one with moderate number of axiom clauses, the other with many more axioms.

In the first set of experiments, we have attempted to prove around 250 lemmas using Vampire. These lemmas were taken from Isabelle theories `equalities.thy` and `Comb.thy`. Around 70 axiom clauses were included in the axiom set. Most of the goals were proved with this axiom set. This is a promising finding as it indicates Vampire (and probably many other good resolution-based provers) can combine Isabelle's equality and classical reasoning.

Moreover, by running large numbers of sample problems, we have also identified those settings of Vampire, which are suitable for our problems. The experiments show that the default setting of Vampire is usually good. This means that we can send Vampire Isabelle's goals and let it run without further instruction, and there is a good chance that a proof will be found.

Moreover, the literal selection mode is the most important factor in determining the speed of proofs. Four selection modes — *selection4*, *selection5*, *selection6* and *selection7* — are better than the others.

Setting File	*No. of Goals Proved from* `Comb.thy`	*No. of Goals Proved from* `equalities.thy`	*Total No. of Goals Proved*
defaultSetting	21	28	49
setting1	22	28	50
setting2	21	27	48
setting3	21	28	49
setting4	18	27	45
combined settings	24	28	52

Table 3.1: Number of Goals Proved with the Large Axiom Set

- *selection4* selects maximal literals. One negative literal with maximal weight is also selected to improve selection.
- *selection5* selects maximal literals. One negative literal is also selected that can minimize the expected number of possible inferences.
- *selection6* performs positive hyper-resolution selection, where negative literals are always selected before positive literals. Among the negative literals, the maximally non-ground ones are selected. From these literals, the ones with greater size are selected. Moreover, in a positive clause, maximal literals are selected.
- *selection7* selects maximal literals. One negative literal with maximal number of different variables, smaller depths of variable occurrences and bigger size, is also selected.

Vampire also supports the set of support strategy (SOS). Most of the goals require us to use this heuristic in order to find proofs: if we turned it off, then either the proofs were found very slowly or Vampire failed to find proofs within 60 seconds. Based on the experiments, we have found five combinations of settings which perform better. They were written to five separate setting files so that we can conveniently use them for later part of experiments. These settings can also be used to run five Vampire processes in parallel when Vampire helps Isabelle users to automatically find proofs after our integration is complete.

As the ultimate aim of our integration is to give Vampire or other automatic provers all of the default classical and simplification rules as axioms, tests with a larger axiom set were necessary. During this second run of tests, around 129 to 160 axiom clauses were used. Vampire tried to prove 37 lemmas (63 separate goals), which were drawn from the previous 250 lemmas. Each lemma was attempted five times using the five combinations of settings: `defaultSetting` uses default settings; `setting1` to `setting3` use Vampire's literal selection mode *selection5* to *selection7* respectively; `setting4` turns on dynamic splitting. The lemmas from `equalities.thy` were mainly proved by the `blast` tactic (with other tactics such as `clarify` and `simp` as well), while lemmas from `Comb.thy` are more complicated and many of them also required the `auto` tactic. Among the 63 goals, fifty-two of them were proved by the combination of all five settings within the time limit. The results are shown in Table 3.1.

The findings from the two sets of experiments suggest that for most of the problems, resolution and paramodulation can indeed replace Isabelle automatic tactics such as `blast`, `auto`. Moreover, for those goals where Vampire failed to prove, it is mainly due to the presence of large numbers of axioms, since they were provable in the first run of the test where we gave only 70 axioms to Vampire.

The experimental results also indicate a potential that more goals can be proved by running several Vampire processes with different settings in parallel, although the results so far are not dramatic. In addition, there are eight relatively complex goals where the amount of time taken by each setting to find proofs varies significantly. It shows that parallelism could prove goals more quickly. Performance variance in different settings is more significant when proving more complicated lemmas (here lemmas drawn from `Comb.thy`).

3.3.2 Experiments on Greater Automation

A more important aim of this integration is to prove goals that cannot be proved by Isabelle's built-in tools and hence improve automation. The second set of experiments examined whether the integration can prove goals that were not proved by `blast`, `auto`, or `simp`. Isabelle proofs of these goals consist of short sequences of proof commands specified by users. If these goals can be proved automatically with our integration, then Isabelle users will not have to specify the proof steps. This set of experiments took 15 lemmas from Isabelle/ZF theory files `Comb.thy` and `PropLog.thy`. The combination of the five Vampire settings was used during the tests.

An issue that we need to consider is at which stage of a proof, we should send the current goal or subgoals to Vampire for an automatic proof. Induction is sometimes necessary to prove a goal and we are not aiming to automate this induction step. Therefore for those lemmas that were proved by induction in Isabelle, we sent to Vampire those subgoals we were left with after induction was performed.

The results are shown in Table 3.2. Some lemmas present more than one subgoal to Vampire. Eight lemmas have their *lemma ID*s marked with asterisks, which means that we have attempted to eliminate all Isabelle proof steps for these lemmas.

Ten lemmas out of fifteen were proved by Vampire. What is more encouraging is that when Vampire proved lemma 10, it completely eliminated six Isabelle proof steps. In addition, among the eight lemmas that we tried to let Vampire completely replace Isabelle's proof steps, six succeeded. These experimental results indicate that we can improve Isabelle's automation by letting external resolution prover, such as Vampire, to prove many goals too complicated for any Isabelle's automatic tactics.

For the five lemmas that could not be proved by Vampire automatically, one or more subgoals' proofs were not found. Lemma 2, 11 and 14 all present more than one subgoal to Vampire and each has one of the subgoals proved. These subgoals were generated after induction was applied to the original single goal. The subgoals that were failed to be proved represent cases in the inductive steps. Perhaps they are too difficult for Vampire to prove automatically.

Lemma ID	*No. of Isabelle Proof Steps Eliminated*	*No. of Subgoals Sent to Vampire*	*No. of Subgoals Proved by Vampire*
1	4	2	2
2	2	2	1
3	3	2	2
4	2	1	1
5*	2	1	1
6*	4	1	1
7*	2	1	1
8*	3	1	1
9*	4	1	1
10*	6	1	1
11	3	3	1
12*	3	1	0
13*	2	1	0
14	10	2	1
15	4	2	2

Table 3.2: Number of More Complex Goals Proved by Vampire

3.3.3 Performance on Large Axiom Sets

An issue that arose from our Isabelle/ZF experiments is that many problems could only be proved for a minimal set of axioms and not with the full set of default axioms. Recall that one of our objectives is to preserve Isabelle's policy of not usually requiring the user to identify which previous lemmas should be used.

We took fifteen problems that seemed difficult in the presence of the full axiom set. We offered them to Geoff Sutcliffe for inclusion in the TPTP Library [62] (COL088-1 to COL100-2 and SET787-1, SET787-2). He kindly ran experiments using three provers (E, SPASS and Vampire) together with a tool he was developing for the very purpose of eliminating redundant axioms. Gernot Stenz ran the same problems on E-SETHEO, because that system is not available for downloading. Finally, I attempted the problems using both Vampire and SPASS. Thus, we made $15 \times 6 = 90$ trials altogether. These trials were not uniform, as they involved different hardware and different resource limits, but they are still illustrative of our difficulty.

Of the fifteen problems, only five could be proved. Only seven of the ninety proof attempts succeeded. We give a more detailed result later (§5.1).

Unfortunately, the hardest problems arose from proofs using the technique of *rule inversion*, which is important for reasoning about operational semantics. Rule inversion is a form of case analysis that involves identifying which of the many rules of an operational semantics definition may have caused a given event. Isabelle's `blast` method handles such proofs easily, but converting the case analysis rule to clause form yields an explosion: 135 clauses in one simple case. We have been able to reduce this number by various means, but the proofs remain difficult.

	Pos. Subset	*Neg. Subset*	*Both*
No. of Goals Involving	14	31	13
No. of Goals Proved if only Keeping	11	18	6
No. of Goals Proved if Removing Both Pos. and Neg. Subset	13	30	13

Table 3.3: Number of Goals Proved with and without Subset Relations

3.3.4 Some Other Findings

While we performed the previous runs of experiments, we discovered some limitations of Vampire. We found many problems involving subset relations or equalities were hard for Vampire to prove. In order to investigate the reason behind the behaviour and to find a feasible solution, we performed more specific experiments on the subset relations and equalities.

Subset and Membership Relations

For many problems concerning subset relations, Vampire either spent an excessive amount of time in the proof search or failed altogether. One solution we envisaged was to replace the subset relation by its equivalent membership relation. In order to investigate whether this solution could help, we carried out proofs of 32 goals. These goals involve either positive subset predicates or negative subset predicates, or both. Without replacing any subset predicate, only 17 goals were proved. However, after we removed all subset predicates, 30 goals were proved. A more detailed comparison is shown in Table 3.3.

Some explanation of Table 3.3 may be useful. The intersection of row *No. of Goals Involving* with column *Pos. Subset* indicates the total number of goals where positive subset predicates exist. These goals may involve negative subset predicates as well. The intersection of row *No. of Goals Proved if Removing Both Pos. and Neg. Subset* with *Pos. Subset* indicates that 13 goals were proved (out of 14 goals that involve positive predicates) once all subset predicates (both positive and negative) were removed. Similarly for other columns and rows.

The experimental results showed that many goals were not proved in the presence of subset relations. However, once we replaced subset relations by membership relations, almost all goals were proved. This seems to suggest that the proofs of these goals did require Vampire to reduce the subset relation to the membership relation via resolution and Vampire did not successfully resolve the desired literals.

Moreover, if we look at the results more closely, we can see the effect of a positive subset is not quite the same as that of a negative subset: Vampire's performance improvement (in terms of numbers of goals proved) was seen greater when we replaced negative subset than we replaced positive subset. This should be explained by the fact that the literal selection function of Vampire does not treat the same literal with different polarities in the same way.

The findings above seem to indicate we should always replace the subset relation by the membership relation. One possible explanation is that in set theory (and thus also in

ZF set theory), the fundamental relationship is the membership relation, which is used to define all other functions and therefore the membership relation participates in most of the proofs. This finding may be useful for Isabelle/ZF. However, such replacement may be unnecessary when we formalize another object-logic in first-order logic.

Tests on Equality Literals

We tried to prove twelve goals involving equality literals. Eleven of these goals have negative equalities and one has a unit clause with a positive equality literal. Vampire quickly found a proof for the goal containing the positive equality. In comparison, only one goal involving a negative equality was proved. Once we removed all negative equalities using subset and then membership literals, ten goals were proved.

The results above showed that the equalities in our experiments had to be resolved and reduced to subset relations for a proof to be found. When Vampire found a proof containing positive equality, this is explained by the fact that in a unit clause, a positive equality is definitely selected and resolved with some negative equality literal (a negative equality literal receives a higher weight than other literals occurring in the same clause). However, in the case when negative equality had to be resolved, Vampire's literal selection function did not select a positive equality in another clause.

The finding also suggests that we should replace equalities by subset relations whenever possible. However, if we replace positive equalities, then we would prevent the use of paramodulation. There may be many problems whose proofs merely require substituting equals by equals. Such problems may not be proved (at least not as quickly) if we replace positive equalities by subset relations. On the other hand, we can replace negative equalities by the subset relations since negative equalities do not take part in paramodulation. Such replacement provides an option of an alternative formalization.

3.4 Formalizing Isabelle/HOL in First-Order Logic

Isabelle/HOL [38] implements classical higher-order logic. In contrast to untyped Isabelle/ZF, Isabelle/HOL has a complex type system (§2.7.4), which is usually not supported by standard first-order automatic provers. From our analysis presented earlier in this chapter, we have seen the potential benefits of sending type information to an automatic prover for a proof search. As a result, we now need to design a sound and efficient encoding of Isabelle/HOL's types.

We must encode types in first-order logic. We take two steps for this encoding. First, we represent Isabelle's type system in first-order logic. Afterwards, we embed the types of polymorphic operators in first-order formulae.

Representing types in first-order logic is a major task of formalizing Isabelle/HOL. However, before that, we need to briefly look at the boolean equalities that are present in Isabelle/HOL.

3.4.1 Boolean Equalities

Isabelle has equalities between boolean terms, which are not in first-order logic. For these boolean equalities, we translate them to two-way implication. For instance, set union

satisfies the equality (`Un_iff`)

$$\forall c\, A\, B\, [(c \in A \cup B) = (c \in A \vee c \in B)].$$

We translate this equality to the boolean equivalence

$$\forall c\, A\, B\, [c \in A \cup B \leftrightarrow (c \in A \vee c \in B)],$$

which can then be replaced by a conjunction of two implications

$$\forall c\, A\, B\, [(c \in A \cup B \rightarrow c \in A \vee c \in B) \wedge (c \in A \vee c \in B \rightarrow c \in A \cup B)].$$

Boolean equalities usually occur in Isabelle's simplification rules. Therefore any simplification rule that is a boolean equality should be transformed to the format above.

3.4.2 Formalizing Isabelle/HOL's Type System

Isabelle/HOL's type system consists of types, axiomatic type classes, sorts, subclass relations and type constructors. Formalizing the type system requires us to represent each of these components in first-order form. We should start by encoding types and type classes, since their representations directly determine the others'.

The relationship between types and type classes is a set membership relation. A naive representation of this may define a membership predicate mem, which is similar to the true set membership predicate $\in$ used in Isabelle logics. For instance, if a type τ is an instance of type class C then we may represent this as $mem(\tau, C)$. However, this is not the best way to represent types.

A better solution is to describe sets using the most basic forms of first-order logic. In this approach,

- the set membership is described by a predicate: if x belongs to set A then $A(x)$ will be true.
- the set intersection is described by predicate conjunction: if x belongs to sets A and B then both $A(x)$ and $B(x)$ are true, hence $A(x) \wedge B(x)$ is true.
- the subset relation is described by predicate implication, by first reducing the subset relation to the membership relation.

A restriction of this formalization of set is that one cannot quantify over sets, but Isabelle does not allow quantification over its type classes.

As a result, we represent type classes by predicates, and types by terms. Subclass relations resemble subset relations and are represented by predicate implications. A sort, which is an intersection of type classes, is treated as an intersection of sets, hence can be expressed by a predicate conjunction. Finally, type constructors are simply functions from types to types; we use first-order functions to represent them.

We can now formally define the first-order representation of Isabelle/HOL's type system.

Definition 6. We formalize Isabelle/HOL's type system as follows:

- We represent each type class by a unary predicate, and represent a type by a term. If a type τ is an instance of a class C, then $C(\tau)$ will be true.

- We represent subclass relation by universal implication. Therefore, if C is a subclass of D, then
$$\forall \tau\, [C(\tau) \rightarrow D(\tau)]$$
will be true.

- We handle sorts by predicate conjunctions. Therefore, if $\tau :: C_1, \ldots, C_n$ (type τ belongs to classes $C_1, \ldots, C_n$) then
$$C_1(\tau) \wedge \ldots \wedge C_n(\tau)$$
will be true.

- We represent type constructors by first-order functions. Furthermore, each type constructor can have multiple arities. Therefore, for each type constructor op, we translate each of its arities with the form
$$op :: (C_1, \ldots, C_n)C$$
into a first-order Horn clause
$$\forall \tau_1\, \ldots \tau_n\, [C_1(\tau_1) \wedge \ldots \wedge C_n(\tau_n) \rightarrow C(op(\tau_1, \ldots, \tau_n))].$$
For example, the function type constructor fun has an arity $fun :: (type, type)type$, which means if both arguments of fun are instances of class $type$, then the result of this function type also belongs to class $type$. We formalize this in first-order logic as
$$\forall \tau_1\, \tau_2\, [type(\tau_1) \wedge type(\tau_2) \rightarrow type(fun(\tau_1, \tau_2))].$$

3.4.3 Embedding Type Information in First-Order Clauses

Isabelle's predicates and functions are typed and many of them are polymorphic. This information must be conveyed to automatic provers. After we encode the type system in first-order logic using the method described in the previous section, we include polymorphic types in clauses.

Please note: we consider polymorphic types for each clause, rather than for each formula in an arbitrary first-order format: formulating polymorphic types on a per clause basis is more efficient. This is made possible because we perform clause normal form transformation to a polymorphic formula first, with the result as typed first-order clauses.

Definition 7. We include types of polymorphic operators and type constraints from type classes in the following way:

- For predicates (other than equality) and functions, we include their types as additional arguments: for an n-place ($n > 0$) function or predicate op, we translate $op(t_1, \ldots, t_n)$ to
$$op(\tau, t_1, \ldots, t_n)$$

where τ is the type of *op*. If *op* is polymorphic then τ is the instance of the polymorphic type when *op* is applied to terms $t_1, \ldots, t_n$. Constants do not need to carry type information: their types should be inferred automatically. Removing unnecessary type information can help to reduce the size of the terms and hence may reduce the processing time of automatic provers.

- Equality is discussed below. However, equalities between boolean values — which are legal in higher-order logic — are simply replaced by two implications.

- Any type class constraints on type variables occurring in a clause are included either as additional literals or additional clauses, depending on whether the clause concerned is an axiom clause or a negated conjecture clause. Type variables are schematic in axiom clauses but are fixed in negated conjecture clauses.

  – If the clause is an axiom clause, then we include type class constraints on type variables as preconditions, in the form of additional negative literals: one for each type variable. For instance, if the clause is $L_1 \vee \ldots \vee L_n$ — where $L_1, \ldots, L_n$ are literals in the clause — and it contains the type variable τ, which is an instance of class C, then we include this constraint in the clause as $\neg C(\tau)$. Therefore, the first-order axiom clause representing the formalized axiom clause above is

  $$\neg C(\tau) \vee L_1 \vee \ldots \vee L_n,$$

  and τ is a universally quantified variable.

  – If the clause is a negated conjecture clause, then we represent the type class constraints as additional unit clauses: one for each type variable. In the example above, if $L_1 \vee \ldots \vee L_n$ is a negated conjecture clause, then we represent the type constraint as an additional unit clause $C(\tau)$. Therefore, we use two clauses

  $$C(\tau)$$

  and

  $$L_1 \vee \ldots \vee L_n$$

  to represent the formalized negated clause above and τ is a fixed variable.

In order to see the effect of this type embedding, let us consider the polymorphic relation $\leqslant$. When it is applied to two linearly ordered arguments its type becomes $\alpha \to \alpha \to bool$, where α is a type variable of class *linorder*. Therefore Isabelle axiom `linorder_linear`

$$\forall x\, y\, [x \leqslant y \vee y \leqslant x]$$

which says any two terms x and y are linearly ordered, provided they both have some type τ that belongs to class *linorder*, will be translated to

$$\forall \tau\, x\, y\, [\neg linorder(\tau) \vee (le(F(\tau, F(\tau, bool)), x, y) \vee le(F(\tau, F(\tau, bool)), y, x))]$$

where *le* is the predicate $\leqslant$ in prefix form and F is the function type constructor. In this case, the type class constraint is that τ — the type of x and y — must belong to class *linorder*, which is represented by a negative literal.

The type class precondition restricts the instantiation of the type variable and hence the application of the axiom: the axiom can be applied by instantiating the type variable τ to any instance of class *linorder*. For example, since the natural number type *nat* is an instance of *linorder*, the formula $linorder(nat)$ is true. Therefore we can instantiate the τ to *nat*. However, we cannot derive $linorder(\alpha\ set)$ for any α, thus we cannot instantiate τ to $\alpha\ set$.

Types for Equalities

Isabelle/HOL's polymorphic equality requires special treatment because equality is built into most automatic provers — it can only take two arguments as in $A = B$. Therefore we cannot insert types of equalities as additional arguments.

We have attempted several approaches to solve the problem. First, we defined a new equality predicate that took three arguments: two equal terms and the type of the equality. Moreover, we also included necessary equality axioms, such as reflexivity, symmetry and transitivity. We then ran a series of experiments in order to examine whether this approach could work in reality. Unfortunately our experimental results showed that Vampire spent an excessive amount of time in trying to find proofs involving this new equality predicate. Even for some trivial examples, Vampire spent tens of seconds in the proof search. The reason for this behaviour is that resolution- and paramodulation-based provers treat their built-in equality as part of the logic. The proof search involving equalities are carried out by inference rules such as superposition rather than simple equality reasoning. Therefore using built-in equalities, automatic provers will be able to give much better performance than could be achieved with any user defined equality literal. As a result, we will have to use the built-in equality.

In order to use the built-in equality of an automatic prover, we should insert type information at a different place. Instead of bounding the type of equality with equality predicate, we embed the type information in its arguments: the formula $A = B$ with type information included becomes

$$equal(typeinfo(A, \tau), typeinfo(B, \tau))$$

where *equal* is the built-in equality of the prover and τ is the type of A and B. In addition, we include the axiom

$$equal(typeinfo(A, \tau), typeinfo(B, \tau)) \rightarrow equal(A, B)$$

to allow an automatic prover's equality reasoning (such as paramodulation) to work. Its effect is to strip the types away, so that an occurrence of A may be replaced by B, or vice versa. Furthermore, equalities in previously proved lemmas and conjectures are translated into *equal* in this format with all type information included.

This approach of including types in equalities is useful when the application of inference rules depends on the type of equality's arguments. For instance, when we prove set equality $A = B$, we may need to prove $A \subseteq B$ and $B \subseteq A$ using the inference rule $A \subseteq B \wedge B \subseteq A \rightarrow A = B$; when we prove integer equality $i = j$, we may need to prove $i \leqslant j$ and $j \leqslant i$ using the rule $i \leqslant j \wedge j \leqslant i \rightarrow i = j$. The inclusion of types prevents an untyped automatic prover from attempting to prove absurdities like $A \leqslant B$ or $i \subseteq j$.

However, further experiments showed that this approach sometimes harms an automatic prover's performance, when equalities are only meant to substitute equal by equal. In this case, types embedded in equalities have to be removed using the axiom above before an automatic prover's equality reasoning can take any effect. Therefore we decide to regard equality as untyped. When we translate an equality between A and B, we translate it to $equal(A, B)$, where A and B are themselves typed. We may only switch to typed equalities in the form that we have described above, if untyped equalities make a proof fail in the circumstances such as a wrong inference rule has been used to find an absurd proof.

In conclusion, Isabelle's type information exists in various places and needs to be extracted and converted to first-order format. The type class information for type variables in Isabelle goals is translated into additional clauses. Subclass relationships and type constructors' arities are global facts, and hence are converted to axiom clauses. The type class information for type variables from Isabelle theorems are translated into extra literals of the axiom clauses.

3.5 Experiments on Formalizing Isabelle/HOL

We used the same general approach as we did in our earlier experiments on untyped ZF formulae. Isabelle/HOL lemmas were chosen, each of them usually presenting more than one goal to Vampire. The combination of the five setting files was used. The time limit for each proof attempt was 60 seconds. We used formula renaming [39] before the clause normal form transformation in order to minimize the number of clauses.

The experiments on formalizing Isabelle/HOL in first-order logic were intended to demonstrate whether the type encoding is practical for resolution (§3.5.1). Furthermore, we also aimed to use this set of experiments to examine whether the performance of an automatic prover can indeed benefit from the inclusion of types; the type information should help an automatic prover to decide which inference rule is applicable and hence reduce the search space (§3.5.2). For instance, if we want to prove the subset relation between two sets: $X \leqslant Y$, its typed formula will be

$$le(F(set(\tau), F(set(\tau), bool)), X, Y)$$

Clearly inference rules such as

$$le(F(nat, F(nat, bool)), A, B)$$

which concerns the $\leqslant$ relation (le) on natural numbers, is not applicable. We expect an automatic prover to ignore this rule since the types of le do not match.

3.5.1 Examination on Type Formulation

This set of experiments took 56 lemmas (108 goals) from the Isabelle/HOL theory files and tried to reproduce the proofs. We used the following theories:

- `Multiset.thy`: a development of multisets.
- `Comb.thy`: combinatory logic formalized in higher-order logic.

Theory	*No. of Lemmas*	*No. of Goals*	*No. of Goals Proved*
`Multiset`	3	3	3
`Comb`	18	29	24
`List_Prefix`	7	8	8
`Message`	28	68	62

Table 3.4: Number of Goals Proved for Typed Lemmas

- `List_Prefix.thy`: a prefixing relation on lists.
- `Message.thy`: a theory of messages for security protocol verification [45].

During the experiments, around 70 to 130 axiom clauses were used. Ninety-seven goals were proved using this typed formalism, as shown in Table 3.4.

Eleven lemmas from `Message.thy` cannot be proved by Isabelle's classical reasoners directly. Either they consist of more than one proof command or they explicitly indicate how existing theorems should be used. Vampire proved seven of these lemmas and three more once some irrelevant axioms were removed. Only one lemma could not be proved at all, and we came close: only one of its seven subgoals could not be proved. Although this is a small sample, it suggests that Vampire indeed surpasses Isabelle's built-in tools in many situations.

Moreover, we carried out further experiments on those failed proof attempts. Among the six failed proof attempts on goals from `Message.thy`, three were made provable by removing some irrelevant axiom clauses. This again indicates that automatic provers are not good at handling large numbers of axioms.

There were also some goals from `Multiset.thy` and `Message.thy` that were proved by applying results from several axiomatic type classes. This finding suggests that our formalisation of types and sorts is practical, while preventing the application of lemmas when the type in question does not belong to the necessary type class.

3.5.2 Using Type Information to Reduce Search Space

We performed more specific experiments in order to examine whether the use of type information on overloaded operators can indeed reduce the search space. For this investigation, we need to prove a goal that contains some polymorphic operator. While we give an automatic prover those relevant theorems containing the polymorphic operator, we also send the prover many irrelevant theorems that share the polymorphic operator. Although this could potentially complicate the search space of the prover, the type information should help it to distinguish the useful theorems from irrelevant ones.

For this purpose, we carried out some experiments that involved proving lemmas about the subset relations taken from the Isabelle/HOL's theory file `Set.thy`. In addition to the relevant axioms about subset properties, many irrelevant axioms about natural numbers were also included in the axiom set as they share the overloaded operator $\leqslant$. We first ran the experiments by not including the axioms of natural numbers and then ran the experiments again while adding those natural number axioms. Vampire spent the same amount of time in proofs regardless whether the natural number axioms were added or not. Clearly, the presence of irrelevant axioms did not make proofs harder for Vampire

when types were included. This demonstrates the benefits of including type information of overloaded operators, since without these information, Vampire may pick up irrelevant natural number axioms in the proof search, which would slow down the proof procedure.

During the experiments with Isabelle/HOL's `Comb.thy`, we also tried to translate HOL conjectures into first-order clauses without the inclusion of type information in order to compare the performance between the typed proofs and untyped proofs. These untyped experiments took the same set of goals (29 goals) from `Comb.thy`. Among these 29 goals, there were three Isabelle goals where Vampire found proofs more quickly when given untyped input and four goals where Vampire proved faster when given typed inputs. In particular, there was a case where Vampire proved a lot faster when given typed input (0.4 seconds compared with 36 seconds for untyped input). However, for those goals where untyped input required less time to be proved, the difference in time taken for typed and untyped input was not very significant. When typed input gave better performance, it could be explained by the restriction of the search space. For the cases where untyped input outperformed typed input, it could be caused by the large literals in typed input due to the inclusion of types. Large literals may slow down proof search to some extent.

We have noticed that more proofs were found for Isabelle/HOL's lemmas than for Isabelle/ZF's lemmas. We believe that type information deserves the credit for this improvement: it reduces the search space.

3.5.3 Some Other Findings

During the experiments, we also investigated the formalization of polymorphic equalities. Among the goals from `Message.thy`, eight goals required the use of equality clauses (on sets), in either lemmas or conjectures. Five of them were proved by switching SOS off and were not proved otherwise. One of them was proved by turning SOS off and removing some irrelevant axioms. The other two could only be proved if the equality literals were replaced by two directional subset relations.

Recall that SOS is incomplete in modern ordered resolution. The experimental results showed that SOS' incompleteness became evident only when proving equalities. It could not prove some trivial examples, and reported unprovable immediately after getting started. However, SOS is better at ignoring irrelevant axioms and should be turned on for our integration. We can always turn if off when it reports unprovable, which usually happens within one or two seconds after a proof is started.

Although we set 60 seconds for each proof attempt, we also allowed a longer time for Vampire to re-try those failed proof attempts. We were hoping to see whether increasing the time for Vampire could make more goals proved. We found that if a goal could not be proved within a minute or two then it would not be proved at all, regardless how long we let the prover run. This finding consolidates our decision to have 60 seconds as the time limit, and if a proof cannot be found by a Vampire process within 60 seconds, we should run another process with a different setting.

3.6 Obtaining Forward and Backward Chaining

Isabelle theorems usually have information indicating whether they should be used for forward chaining (as elimination rules) or backward chaining (as introduction rules). For

instance, the introduction rule of set intersection (`IntI`)

$$\forall c\, A\, B\, [c \in A \to c \in B \to c \in A \cap B]$$

should be used in a backward chaining style: in order to prove $c \in A \cap B$, it is enough to prove $c \in A$ and $c \in B$. When proving any goal of the form $x \in Y \cap Z$, Isabelle resolves $c \in A \cap B$ with the goal during resolution, leaving the new subgoals $x \in Y$ and $x \in Z$. In contrast, the elimination rule of set intersection (`IntE`)

$$\forall c\, A\, B\, P\, [c \in A \cap B \to (c \in A \wedge c \in B \to P) \to P]$$

should be used in a forward chaining style: given $c \in A \cap B$, we can derive $c \in A$ and $c \in B$. Therefore $c \in A \cap B$ can be resolved with an assumption of the form $x \in Y \cap Z$, deriving new assumption $x \in Y \wedge x \in Z$. As we can see, these information essentially says which literal in a theorem should be eliminated first. This information corresponds to which literal should be resolved first in resolution provers.

Such information about how theorems should be used is lost after they are translated to clauses. When `IntI` is converted to clause form, it generates a clause

$$c \notin A \vee \ c \notin B \vee c \in A \cap B.$$

Given this clause, an automatic prover may select any literal to resolve during resolution. However, an automatic prover should only select $c \in A \cap B$ to resolve as this exactly corresponds to the way `IntI` is used in Isabelle. As it shows, it is desirable if we can tell automatic provers which literal should be resolved: resolving on a wrong literal will be wasteful and could significantly hamper the performance of automatic provers.

In the modern ordered resolution, many automatic provers base their literal selection on literals' weights: they usually select a literal with a higher weight relative to other literals in the same clause. This difference in weights gives an ordering on literals. Therefore getting a proper ordering on literals is a possible solution to our problem. We have used Vampire for our experiments.

The public release of Vampire does not allow explicit weight assignment to literals, but uses Knuth-Bendix Ordering (KBO) [51] to compute an ordering on literals. Recall that KBO is parameterized by weights and precedences of functions and predicates, which can be assigned explicitly by users. Therefore we have attempted to assign weights and precedences to functions and predicates. However, there are two problems with this approach.

- First, the information on Isabelle theorems only says which operator should be resolved first *in that theorem*. A same operator may have this priority in one theorem but not in another. As a result, when translating these theorems to first-order clauses, the ordering on literals should be local to their enclosing clauses. However, the weight and precedence assignments have global effect on all clauses.

- Second, the resulting KBO is a partial ordering on terms with variables, thus we may not have a desired ordering on literals regardless how we assign weights and precedences.

These limitations of KBO mean this approach does not solve our problem exactly.

The new version of Vampire (v6.03) has a special syntax which is an extension of the TPTP syntax (§2.3.2). Recall that any positive literal that should be resolved first will be tagged with `+++`, and similarly a negative literal should be tagged with `---`. Moreover, the tagging only has effect within every clause. Using this new syntax, we can label $c \in A \cap B$ in the clause generated from `IntI` with `+++`. This special syntax simulates Isabelle's notions of forward and backward chaining. Experiments showed that many lemmas could only be proved with this facility.

3.7 Concluding Remarks

In this chapter, we have discussed how to use resolution-based provers to assist Isabelle.

In the initial investigation into our research, we identified some major problems we need to solve in order for the integration to work. We tried to solve these problems by formalizing Isabelle/ZF and Isabelle/HOL in first-order logic. We have also carried out many experiments in order to examine the quality of our translation method. Based on the experimental results, we have been able to answer many questions and solve many problems. The findings are now summarized below.

- Each Isabelle object-logic contains some logic-specific information (such as the semantics of a logic), which may be useful when we translate it into first-order form. Therefore we should translate each object-logic separately.

- Resolution- and paramodulation-based provers are indeed suitable candidates for our integration by providing extra automation. Their power is more evident if they provide mechanism to simulate Isabelle treatment to theorems, which is specific to the natural deduction calculus only.

- The type information of Isabelle/HOL is useful. It not only ensures soundness but also reduces the search space of automatic provers. This suggests that when we formalize other typed object-logics in first-order clauses, we should also preserve their types.

- The settings of an automatic prover determine its proof search strategy and are essential to our problems. Among them, some are more significant. We have found some settings good for our problems, which can be used in the future integration.

- However, both Isabelle/ZF and Isabelle/HOL experiments indicate the major problem of resolution-based provers, which is their inability to cope with large numbers of axioms.

Although we did most of the experiments on Vampire, the results should be generally applicable to many other resolution-based provers. The findings should give some insight into the possible behaviour of other automatic provers, such as SPASS and Otter.

Our research has mainly been focused on formalizing the first-order logic part of Isabelle/ZF and Isabelle/HOL, except for some non-first-order ZF terms. We decided to use the formalization of ZF as the pilot study. Although having been well developed, it is not the most widely used object-logic among Isabelle users. Therefore it is sufficient to try

out some examples with the primary objective on examining the efficiency of resolution provers.

When we formalized Isabelle/HOL, we have largely left out higher-order logic constructs, such as the λ-terms. There was a major reason for this: many Isabelle proof goals are, in fact, expressed in first-order logic. Higher-order logic constructs are more frequently used in initial formalization of theories rather than being used in actual proofs. In contrast, its type system is more important in our formalization. Therefore, formalizing first-order aspect of Isabelle/HOL is already good enough to prove many goals automatically.

Chapter 4

Calling Automatic Theorem Provers

In the previous chapter, we have described the obstacles in integrating Isabelle with any resolution-based theorem prover. We have put emphasis on translating Isabelle formalisms into first-order logic. In particular, we have designed a method to translate Isabelle/ZF and Isabelle/HOL including Isabelle/HOL's type system into first-order logic. Our experimental results indicate the method is indeed practical.

In this chapter, we explain the general techniques that we have used to automatically translate Isabelle formulae into clause normal form inside Isabelle logic. We also describe the application of these techniques to the translation of formulae of Isabelle/HOL.

The automatic translation must be integrated with Isabelle, so that goals and all essential Isabelle data are extracted and converted to clauses automatically during a proof, without users' interaction or awareness. We describe a design and an implementation in this chapter.

4.1 Design Considerations

The objective of integrating Isabelle with automatic provers is to free users from doing tedious proof steps. In addition, the process of delegating jobs to automatic provers should be automatic as well — the invocation of the target automatic provers should not require users to do any extra work. Therefore, we need a delegation program, which links Isabelle and one or more automatic provers and carries out the automatic communication. After Isabelle is integrated with automatic provers via the delegation program, a scenario of a proof session can be described as follows.

- An Isabelle user works normally: making definitions and carrying out proofs of theorems.
- While the user is engaged in a proof and is refining goals to subgoals, if there is any open subgoal, then the delegation program — and not the user — will collect the subgoals and all the necessary information and send them to a background automatic prover. The automatic prover will then start searching for proofs of the subgoals.
- While the user is still trying to solve the goal, the automatic prover may find a proof. The delegation program will translate this proof to an Isabelle proof and will

notify the user of the success. The user will no longer be required to find the rest of the proof steps himself. He can cut and paste the constructed Isabelle proof steps, and then re-run these Isabelle proof steps in order to complete the proof of the goal.

This delegation program directs the two-way communications between Isabelle and the background automatic provers. In my research, I have investigated the communication from Isabelle to automatic provers and have implemented a program that performs the communication in this direction. This program should follow the progress of a user's proof development and whenever there is an opportunity for an automatic prover to help prove a goal, it should send the goal with other relevant data to the automatic prover.

In order to achieve the effect described above, the first task for our research is to identify an appropriate point during an Isabelle proof when a proof goal should be sent to a background automatic prover. In addition, our program must be able to locate and extract all the necessary information, without a user's instruction, convert this information into first-order clause form, and finally send the clauses to designated places ready for the automatic prover to read. Furthermore, although my research does not concern reconstructing the automatic provers' proofs into Isabelle proofs, which is the communication from the automatic provers to Isabelle, the implementation of my program must ensure that proof reconstruction is possible. We now start to design a program, which can solve all these questions.

4.1.1 When to Call An Automatic Prover

It might seem obvious that our program should call an automatic prover when an Isabelle user is carrying out a proof and when a new goal is declared. However, a closer analysis reveals that not all such points during a proof are suitable for our program to call an automatic prover. When a new goal is declared, it may be in one of the two conditions: either it can be changed immediately or it is not allowed to be changed immediately. If a goal can be changed immediately, a user can apply a tactic to decompose it to several simpler subgoals or prove the goal altogether. If a goal is not allowed to be changed immediately, then the user typically continue with the proof by stating local assumptions, etc. Among these two types of goals, only those goals that are not subject to immediate changes are suitable to be sent to an automatic prover by our program.

There are two major reasons for our choice. First, if the new goal is subject to immediate changes, then the proof response from an automatic prover may not keep up with the speed at which the proof goal is being modified. In this case, a goal may be changed by a sequence of tactics that are applied by the user, which means that by the time an automatic prover finds a proof, the original goal sent to the prover is already obsolete. Second and more importantly, if a goal may be changed by a tactic, then the result of applying the tactic is another new goal, which is again subject to immediate changes by tactics. This can easily lead to a sequence of tactics on a proof goal, which in turn will result in a sequence of intermediate goals. If all of these intermediate goals are sent to an automatic prover, the prover will be overwhelmed by the vast volume of proof requests. Consequently, our program should only send those goals that are not allowed to be modified immediately.

Recall that in the Isar interface, a proof is always in one of the three proof modes (§2.6.5): **prove**, **state** and **chain**. In **prove** mode, goals are subject to immediate

changes, whereas once a proof enters **state** mode, goals remain unchanged. Therefore, our program should send all open subgoals to an automatic prover once the proof enters **state** mode. In fact, this is a good position to delegate jobs to a background prover for another reason. Typically when a proof enters **state** mode, a user is expected to consider what assumptions etc. should be stated, which is the preparation work before the goal is actually proved. The actual proof of a goal is usually carried out by the application of appropriate tactics or structuring methods. Compared with the actual proof steps, the preparation work requires less human effort. For instance, the proper assumptions that should be made are usually indicated by the structure of the proof goal. On the other hand, the actual proof steps would require the user to decide what rules should be applied and how to apply these rules — this is exactly the kind of work that the user should be freed from doing, by having automatic provers to do it for the user. We are hoping that the automatic prover can find a proof while the user is only carrying out the preparation work, thus no huge amount of effort has been spent on the actual proof steps. This is a strong justification for calling the automatic provers when a proof enters **state** mode. Finally, although a proof may enter **chain** mode at some point during a proof, and goals are not subject to immediate changes in this mode, our program does not need to send any goal to an automatic prover at this point. This is because when a proof enters **chain** mode, no new goal is declared. **chain** mode merely allows previously claimed local assumptions to be carried over and be aggregated with any new claim to be made.

Based on these considerations, we need to define a function that watches for the changes of proof modes. Whenever a proof is about to enter **state** mode, this function should start the process of sending goals and other necessary information to the background automatic provers. After this is done, the proof enters **state** mode. In addition, this invocation of automatic provers should be invisible to Isabelle users: our function should not present any effect observable by the rest of Isabelle program.

In contrast to the Isar interface, the ML interface does not provide the three proof modes. In the ML interface, goals are proved by tactics directly, and are thus always subject to immediate changes. As a result, we cannot implement a program that calls the automatic provers at appropriate points automatically. We have to define a tactic, which when applied by a user, delegates proof jobs to the background automatic provers. Therefore it is a user's responsibility to decide when to call an automatic prover.

Since most of the Isabelle users are now working in the Isar interface, we concentrate on describing the program we have implemented for the Isar interface in this chapter. We also briefly describe the tactic version that we have experimented and implemented for use by the ML interface at the end of this chapter (§4.8).

4.1.2 What to Send to An Automatic Prover

After the automatic prover calling procedure starts, it is important to identify and locate all essential information that must be sent to the prover. The automatic prover must receive the following items, which are involved in an Isabelle proof.

- Assumptions local to the current proof.
- The subgoals to be proved.
- Existing Isabelle lemmas or theorems.

- Isabelle's type information.

As we have seen in the previous chapter (§2.6.5), the local assumptions are part of the current proof environment, and are stored with the proof context. Therefore, it is sufficient to inspect a proof context to extract local assumptions. Moreover, since local assumptions may be modified when a proof goal is being refined, we must send all the current local assumptions to the automatic provers when we send unsolved subgoals to the provers.

On the other hand, the subgoals represent dynamic information and are stored separately from the proof context: they are part of the proof state. Consequently, we need to inspect a proof state to retrieve all the open subgoals. Moreover, it is possible that several subgoals are waiting to be proved. This may happen when a proof is carried out by induction or case analysis. We must send all of these subgoals to an automatic prover.

In Isabelle, existing theorems are used by automatic reasoning tools during the proof search. These theorems represent a knowledge base and must be sent to our automatic provers. Our integration aims to reduce user interaction by not asking users to name the theorems that should be used for a proof. Consequently, we need to send all existing theorems available under the current proof context to an automatic prover. These theorems include the classical reasoning rules, which are stored in a *classical set* that is used by the classical reasoner. They also include simplification rules, which are stored in a *simplifier set* that is used by the equality reasoner. Although there are other components in a simplifier set, such as the congruence rules, which are essential for the proof search carried out by Isabelle's equality reasoner, we do not need to deliver them to our target automatic provers. This is because equivalent mechanisms are built into resolution- and paramodulation-based provers.

Isabelle theorems are stored at different places, depending on whether they are *global* theorems or *local* theorems. Global theorems are already proved top level goals. After being proved, they can be declared as either elimination rules or introduction rules and then be included in the Isabelle's classical set, or, they can be declared as simplification rules and then added into the simplifier set. Global theorems are not associated with any individual proof goal. Instead, they are stored within the background theory context. Consequently, we can extract the global theorems from a theory context and translate them to clauses once and for all. Subsequently, we can store the generated clauses at a proper place (such as on the machine where an automatic prover runs) before the prover is called. When a proof goal is sent to the automatic prover, the clauses generated from the global theorems can be used by the automatic prover directly.

Similar to global theorems, local theorems can be declared as simplification rules or classical reasoning rules. However, in Isabelle, local theorems are temporary theorems proved during a structured proof. Their only role is to assist the proof of the current main goal. After the main goal is proved, these local theorems become inaccessible and are discarded. As a result, Isabelle stores local theorems separately from global theorems. However, as we can see, local theorems are closely associated with a proof context. Since a proof context stores local assumptions, it makes sense to store local theorems in a proof context as well, so that we can retrieve both local assumptions and local theorems by inspecting a proof context alone. As a result, we should modify Isabelle's existing proof context data structure and add in local theorems when they are declared. Similar to local assumptions, the collection of local theorems may be different at different points

of a proof. Therefore, when we send Isabelle subgoals to an automatic prover, we must extract all the currently available local theorems from the proof context, convert them to clauses and then send the clauses to the automatic prover.

In addition to local assumptions, open subgoals and Isabelle theorems, Isabelle's type information should be conveyed to automatic provers if the provers must perform typed proofs. Some of the type information is embedded in the polymorphic formulae, such as the types of the polymorphic operators and the type classes of the type variables occurring in the formulae. For this sort of type information, we translate them into appropriate first-order form at the same time when we convert their enclosing Isabelle formulae into clause form. In addition, this type information should be sent to an automatic prover along with the enclosing Isabelle clauses: if the types occur in local assumptions, local theorems or subgoals then they are sent to an automatic prover every time the prover is called; if the types occur in global theorems then they can be written to files in advance, so that they can be read directly by the automatic prover when the automatic prover starts a proof search.

Moreover, Isabelle's type system provides type class relations and arities of type constructors. They are regarded as facts and are stored with the background theory context, rather than with goals or theorems. This type information should be extracted from the current theory context and then be converted to first-order clauses once and for all, which can happen before an automatic prover is called. Similar to clauses generated from global theorems, these clauses can be used directly by an automatic prover when the prover starts a proof search.

In summary, we need to send all existing theorems, local assumptions and all the subgoals, plus some other facts concerning Isabelle types, to our target automatic provers. In addition, some of these data — global theorems, type class relations and type constructors' arities — do not change while an Isabelle goal is refined. In order to improve efficiency, this information can be converted to first-order clause form and stored at locations convenient to our automatic provers, so that the automatic provers can use them as a knowledge base. On the other hand, we must find the current values of all other data — which may be changed constantly when an Isabelle goal is refined — and convert them to clauses and then send them to the background automatic provers when a proof job is to be delegated to the provers.

4.1.3 How to Convert to Clause Normal Form

All Isabelle goals and theorems that should be sent to an automatic prover must be translated into first-order clause normal form. The easiest approach to this is direct programming, where we can explicitly manipulate the term structures of Isabelle formulae. However, the operations on term structures cannot be carried out inside Isabelle logic: to ensure soundness, ML's type checker does not allow such operations to be functions of type `thm -> thm`. Consequently, if we directly manipulate the term structures of Isabelle formulae, the clauses will not have type `thm`. Proof reconstruction is impossible unless each clause has type `thm`.

As a result, we have to translate Isabelle formulae inside Isabelle logic, by applying Isabelle inference rules. We build up these inference rules from the primitive inference rules, which are kernel defined functions. Consequently, the inference rules we use for

translating Isabelle formulae have type `thm -> thm`. Therefore, when we transform Isabelle theorems (of type `thm`) into their equivalent clause form using these inference rules, the resulting clauses will still have type `thm`, which are valid Isabelle theorems.

Our translation function mainly consists of the use of three techniques, namely forward proof, simplification by rewriting and application of tactics. The first method has been used in `meson_tac`, implemented by Larry Paulson.

Recall, Isabelle's Horn resolution is the inference rule

$$\frac{\phi \to \theta \quad \theta \to \psi}{\phi \to \psi}$$

Therefore, if we wish to convert a formula A to B, then we can first prove an auxiliary lemma $P \to Q$, where P and Q match the patterns of A and B respectively, but generalize over some of the variables in A and B. Subsequently, we transform A to B by performing the resolution

$$\frac{A \quad P \to Q}{Q\sigma}$$

where A is resolved with P by applying the unifier σ, which also makes $Q\sigma = B$. For instance, to transform the formula $\neg(A \wedge B)$ into $\neg A \vee \neg B$, we carry out the resolution

$$\frac{\neg(A \wedge B) \quad \forall P\, Q\, [\neg(P \wedge Q) \to (\neg P \vee \neg Q)]}{\neg A \vee \neg B}$$

Using this form of Horn resolution, we can effectively change the structure of a top level formula (in the example above, we have pushed in the top level negation inwards). However, sometimes a transformation of a top level formula may require us to transform its sub-formulae to a desired form first and then the updated top level formula can be transformed, i.e. the transformation may have to work in a bottom-up approach. In order to solve this problem, we have used a similar forward resolution technique. This technique can be applied to a proof state. It essentially allows an assumption of a subgoal of a proof state to undergo some sort of transformation, and then resolves the resulting assumption with the conclusion of the subgoal concerned. The resolution between the transformed assumption and the conclusion will update the entire proof state by instantiating some of the free variables. Although forward resolution is usually applied to proof states, we can use it on transforming Isabelle theorems, since proof states and Isabelle theorems are represented in the same way in Isabelle's meta-logic (§2.6.3).

For instance, if we wish to transform any formula $A \otimes B$, where $\otimes$ represents any relational operator, such as disjunction $\vee$, into some format, such as clause normal form, by first transforming A and B into the desired form A' and B' respectively, and then transform $A' \otimes B'$, then we can first apply the inference rule

$$\forall P_1\, P_2\, Q_1\, Q_2\, [P_1 \otimes Q_1 \to (P_1 \to P_2) \to (Q_1 \to Q_2) \to (P_2 \otimes Q_2)]$$

to $A \otimes B$ and get

$$\forall P_2\, Q_2\, [(A \to P_2) \to (B \to Q_2) \to (P_2 \otimes Q_2)].$$

This effectively breaks $A \otimes B$ into two parts. The resulting formula can be seen as a proof state, where $A \to P_2$ and $B \to Q_2$ are subgoals. Subsequently, we apply the forward

resolution technique, so that we first transform the two subgoals' assumptions A and B to A' and B', and then resolve A' and B' with the two subgoals' conclusions P_2 and Q_2. The final result is the new proof state $A' \otimes B'$ — this proof state is the form of the formula we are looking for.

We have mainly used the resolution technique above for translating Isabelle theorems to clauses. Since both the Isabelle theorems and the auxiliary lemmas that we use to transform the theorems are theorems with type `thm`, and the Horn resolution and the forward resolution are kernel defined inference rules, the resulting formulae are valid Isabelle theorems, of type `thm`.

Another method that we have used for transforming Isabelle formulae is rewriting. If we wish to convert a formula with pattern P to another formula with pattern Q, then we first prove $P \equiv Q$ to be a theorem. Subsequently, we add this theorem as a rewrite rule to a simplifier set. Finally, we translate the formula into the desired form by calling the simplifier to simplify the formula. This procedure ensures an instance of the left hand side of a rewrite rule is replaced by the corresponding instance of the right hand side of the rule. Since the rewrite rules have type `thm` and rewriting is also a kernel defined function, the resulting formula is an Isabelle theorem of type `thm`. Although we have used both resolution and rewriting to translate Isabelle theorems to clauses, resolution is more efficient than rewriting. Therefore, most of the Isabelle theorems are transformed by resolution inference.

The third method we have used for transforming Isabelle formulae is by applying tactics. This is used when we need to negate a goal and convert it to clause normal form. We cannot use the two methods above to transform a goal directly. Recall that Isabelle stores all current subgoals inside a proof state and Isabelle represents a proof state as a theorem of type `thm`. However, since the unproved subgoals are not yet theorems, they do not have type `thm`. If we wish to transform a negated goal to clauses inside Isabelle logic, then we cannot simply take a subgoal out of a proof state and perform clause normal form transformation. As a result, our goal transformation process has to take the entire proof state into account. When a task concerns the entire proof state, Isabelle's tactics are the best candidates for this job. Tactics are functions that are best suited to refine a proof state and the effect can be limited to a particular subgoal, by giving the number of the subgoal to the tactic. In order to negate open subgoals and convert them to clauses, we have defined tactics both by combining existing tactics through tacticals and also by coding new tactics. In addition, some of the transformation functions that we have used for translating Isabelle theorems have been used in our tactics. We give more explanation on the use of tactics to negate goals and convert the results to clauses later (§4.6).

The techniques described above can be used to translate formulae expressed in many Isabelle logics. We have implemented a translation function using these techniques to convert Isabelle/HOL's formulae to clause form. We will describe the clause normal form transformation function in more detail in the rest of this chapter.

The transformation mechanism above converts Isabelle formulae into clauses. However, the types of polymorphic operators and type class information on type variables are still embedded in the clauses, in the same way as in the original Isabelle formulae. These clauses are still not suitable to be used by automatic provers directly. We need to perform a further transformation on these Isabelle clauses so that the type information is extracted and expressed in first-order clauses, and we must ensure both the Isabelle

clauses and the type clauses are expressed in formats suitable for our target automatic provers.

Our research mainly concerns integrating Isabelle with Vampire or SPASS, but it is desirable if we can easily add more automatic theorem provers and integrate them with Isabelle. These automatic provers may have complementary strengths. For instance, we may add many different resolution provers. Alternatively, we may use both resolution provers and SAT solvers. They may help prove or disprove Isabelle goals, in different domains. They may also run in parallel, trying to prove a same problem and thus can reduce the amount of time spent on the proof search.

These automatic theorem provers may require their inputs to be in different formats, but they all accept first-order clauses, though written in different ways, which is sufficient for our integration. Consequently, we need to define an internal clause representation so that when an Isabelle goal or theorem is converted to clause form, the clauses generated will be converted to this internal format and then stored (along with its type information). When an automatic prover is called, the clauses in this internal format will be translated to some format specific to that automatic prover and the stored type information will be translated to appropriate format as well. Crucial to the efficient performance, this internal clause data structure must contain all essential information for all automatic theorem provers and must be able to be converted to any automatic prover's specific format quickly.

So far, we have analyzed how to convert Isabelle theorems and goals to clause normal form. We also need to convert other Isabelle's type information, such as type constructors' arities and type class relations, to first-order clauses. This translation can be carried out by straightforward programming. Furthermore, for the same reason above, we should define internal formats for these clauses as well.

4.2 Overview of Automatic Calling Procedure

Based on the design issues and analysis described above, I carried out the implementation of the program that automatically extracts all subgoals and other proof data at an appropriate point during a proof, converts these data to clause normal form and writes the clauses in a suitable format for an automatic prover to read. This automatic procedure works as follows.

- During a proof session, when Isabelle receives a command that causes a proof to enter **state** mode, the function `enter_forward` is called.
- `enter_forward` passes the current proof context and the proof state to the function `isar_atp`, which extracts all local assumptions and local theorems and converts them to clauses in a format suitable to a target automatic prover. `isar_atp` also negates all of the open subgoals and converts them to clauses. All the clauses above are then written to designated files, waiting to be read by the automatic prover.
- Finally, `enter_forward` lets Isabelle enter **state** mode. Calling the external prover is invisible to the user.

Function `isar_atp` retrieves theorems and goals and translates them into clause normal form. It is made up of three sub-components.

- `isar_local_thms` extracts all local theorems available to the current proof context, converts them to clause normal form as axiom clauses and writes the clauses to a designated file `axiom_file`.

- `isar_atp_h` finds all local assumptions, converts them to clause normal form as negated conjecture clauses and writes the clauses to file `hyps_file`.

- `isar_atp_goal` negates all open subgoals and converts each of the negated subgoals to negated conjecture clauses. It writes clauses generated from different subgoals to different problem files, where the names are generated dynamically.

Moreover, the destination files `axiom_file`, `hyps_file` and files for subgoals are generated only for temporary use — they can be discarded once an automatic prover is finished with them. Therefore, we use dynamic generation of files' names, where the actual location of the files depends on the Isabelle session in which the action above takes place. After the procedure above, the files containing clauses will be read by an automatic prover for the proof search.

We consider the three functions — `isar_local_thms`, `isar_atp_h` and `isar_atp_goal` — in more detail in the following sections.

Global theorems and other facts, such as types, can be stored in their clause form permanently with our target automatic provers. Therefore, their translation to clause form does not have to be part of the automatic calling procedure we have described above. We explain their translation later (§4.7).

Furthermore, as we have remarked above, we need internal clause formats to store all clauses generated from Isabelle formulae and type information. We define the data structures of the internal clause formats first.

4.3 Generic Clause Data Types

Recall that when we formalize Isabelle formalisms in first-order logic, clauses are generated from four sources and represent four kinds of Isabelle information. These are theorems, goals, type constructors' arities and subclass relationships between type classes. We define three ML abstract data types to be the internal formats of their clauses.

4.3.1 Type `clause`

After Isabelle formulae (both goals and theorems) are converted to clause form, each clause still has type information embedded. Predicates and functions have their types contained in the clause, and each type variable has a type class requirement on it. Therefore, we need to define a data structure that not only contains a list of literals, but also the type information.

For this, we define the abstract data type `clause` to represent first-order clauses derived from Isabelle theorems and goals. A `clause` object contains the following fields.

- A unique identifier. If the clause is derived from an Isabelle theorem, then the theorem's name is recorded as well. This information is useful for later proof reconstruction.

- An indication of whether the clause should be labelled as an axiom or a negated conjecture clause. Many automatic provers make use of the distinction between them. For instance, the set of support strategy forbids resolutions that involve purely axiom clauses. Moreover, SPASS's `PrefCon` option allows one to set the ratio to compute the weight for conjecture clauses so that those conjecture clauses are preferred over others.

- A list of literals in this clause. Since our encoding formalizes Isabelle types, we also define abstract data types for typed literals and functions.

- Additional type information. This includes type classes of type variables (both fixed and schematic) that occur in the clause.

After an Isabelle theorem or a negated goal is converted to clause normal form inside Isabelle logic, a list of clauses (of type `thm`) is generated. These clauses are converted to the clauses of `clause` type by two functions.

- `make_axiom_clause` converts each clause derived from an Isabelle theorem to an axiom clause of type `clause`.

- `make_conjecture_clause` converts each clause generated from a negated goal or a (non-negated) local assumption to a negated conjecture clause of type `clause`.

Please note, since we define type `clause` to encode first-order clauses, any attempt to convert a higher-order Isabelle clause to this internal format will cause an exception to be raised by `make_axiom_clause` and `make_conjecture_clause`. As a result, these higher-order clauses are omitted when our delegation program sends Isabelle theorems to automatic provers.

4.3.2 Type `arityClause`

Unlike type variables' class requirements, which are contained within each Isabelle clause, type constructors' arities come from a different source: they are usually stored with each theory file as facts. Recall that each arity of a type constructor *op*

$$op :: (C_1, \ldots, C_n)C$$

is formalized as the first-order Horn clause

$$\forall \tau_1 \ldots \tau_n \, [C_1(\tau_1) \wedge \ldots \wedge C_n(\tau_n) \rightarrow C(op(\tau_1, \ldots, \tau_n))].$$

This clause format is significantly simpler than an ordinary Isabelle clause. Therefore, we define a new abstract data type `arityClause` to be the internal format for clauses generated from arities. Each arity is translated into a clause of type `arityClause`. Fields included in an `arityClause` object are

- A unique identifier.

- A positive literal, which represents the type class of the type constructor's result. In the example above, this field is C.

- A list of negative literals, which represent the type classes of the type constructor's arguments. In the example above, this field is $C_1, \ldots, C_n$.

Arities describe type class information about type constructors and thus they should be used as facts by automatic provers. As a result, all clauses of type `arityClause` are labelled as axiom clauses.

Finally, the function `make_axiom_arity_clause` converts each arity of a type constructor into a clause of type `arityClause`.

4.3.3 Type `classrelClause`

In addition to the type constructors' arities, we need to convert subclass relationships to clauses. Each type class relation

$$C < D,$$

which means type class C is a subclass of type class D, becomes the Horn clause

$$\forall \tau \, [C(\tau) \rightarrow D(\tau)].$$

We define a data type `classrelClause` to represent this Horn clause. The fields contained in `classrelClause` are

- A unique identifier.
- A positive literal, representing the superclass.
- A negative literal, representing the subclass, if there is one. Otherwise the field is empty.

Like arities, class relations are facts: we convert each type class relation to an axiom clause. Function `make_axiom_classrelClause` converts each subclass relationship into a clause with type `classrelClause`.

4.3.4 Conversion to TPTP Format

The clause data types defined above can be easily translated to any textual format specific to any automatic theorem prover. In addition, these clauses contain sufficient information for all the automatic provers.

We have implemented the conversion to the widely-used TPTP format. This conversion is carried out by three functions:

- `tptp_clause` translates a clause with type `clause` into one or more TPTP clauses as strings.
- `tptp_arity_clause` translates a clause of type `arityClause` into one TPTP clause as a string.
- `tptp_classrelClause` translates a clause of type `classrelClause` into one TPTP clause as a string.

Please note, when the function `tptp_clause` converts a `clause` object into TPTP format, more than one TPTP clause may be produced. This happens if we are translating a negated conjecture clause, and the negated conjecture clause contains some type variables. Recall that we formalize type class constraints on type variables that occur in negated conjecture clauses as additional unit clauses (§3.4.3). For instance, if a negated conjecture clause is

$$a \in A$$

which contains type information such that a has type τ, which is a type variable, and τ belongs to type class C, then when `tptp_clause` translates this clause to TPTP format, two TPTP negated conjecture clauses will be generated: the additional one is $C(\tau)$.

All clauses of type `clause` contain full Isabelle type information. However, when we translate `clause` objects to TPTP format, we can choose to include or exclude the type information in the output TPTP strings. By default, the output TPTP clauses include type information, except that the equalities are untyped, as described in the previous section (§3.4). Users can modify this behaviour (for instance, on including type information for equalities) by setting boolean flags.

4.4 Isabelle Local Theorems

As remarked above, local theorems should be associated with a proof context. A proof context is implemented as a data structure in Isabelle, which is initialized at the start of a proof and is emptied as soon as the main goal is proved. In order to make it easier for our program to extract local theorems, we add a `delta` field in the proof context data structure. The `delta` field is initialized to empty when a proof starts. Each time a local theorem is proved and declared, this theorem is added to the `delta` field. Subsequently, we implement a function `isar_local_thms`, which inspects the current value of the `delta` field in the proof context and finds all local theorems, when an automatic prover is about to be called.

After `isar_local_thms` extracts all the local theorems from the proof context, it converts the local theorems to clauses using Isabelle's inference rules, by calling other functions that we describe next.

4.4.1 Translating Existing Theorems to Clause Form

Isabelle theorems must be converted to clause normal form. In addition, this transformation must be performed inside Isabelle logic, which means that our clause normal form transformation function must have type `thm -> thm`.

We have defined function `cnf_axiom` to convert each Isabelle theorem to clauses. This function calls primitive inference rules, such as rewriting and resolution with other proved theorems. Therefore, `cnf_axiom` is a function of type `thm -> thm`, which in LCF style provers is the type of derived inference rules.

Function `cnf_axiom` mainly consists of the following functions, each of them has type `thm -> thm`.

- `skolem_axiom` converts an Isabelle theorem into negation normal form, and then performs Skolemization.

- `isa_cls` performs the same steps, then converts the resulting formula into clause normal form, which is represented by a list of clauses. Each clause has type `thm`, and is therefore an Isabelle theorem. In addition, all free variables are schematic and are implicitly universally quantified.

Skolemization takes several steps, with existing lemmas treated differently from the negated conjecture. The first step is to move every existentially quantified variable to the front of the theorem, and is performed by simplification using the rewrite rule[1]

$$\forall P\,[(\forall x\,\exists y\,P(x,y)) \equiv (\exists f\,\forall x\,P(x,f(x)))].$$

This equivalence expresses the axiom of choice, which appears to be necessary when performing Skolemization by inference. A single application of this equivalence to an Isabelle theorem yields a function of one variable. Repeated application — to move an existential variable past several universal variables — results in a function of all of those variables. Rewriting with this equivalence, along with others to extract existential quantifiers from conjunctions, disjunctions, etc., yields a formula in which all existential quantifiers are lined up at the front. Please note: during our implementation, all outermost universal variables have to be turned into free variables that are implicitly universally quantified. Consequently, they do not have $\forall$ in front.

These existential quantifiers must now be removed altogether. The procedure depends upon whether the clauses have been produced from the negated conjecture or from existing lemmas. Skolemization of the negated conjecture is easy: it is treated just like any subgoal that has existentially quantified assumptions. Skolemization of lemmas requires a further use of the axiom of choice, in the form of Hilbert's ϵ-operator. The term $\epsilon x\,P(x)$ denotes some value x such that $P(x)$ is true, if such exists; otherwise, it denotes any value of the appropriate type. If we have transformed a lemma into the form $\exists x\,P(x)$, then we may conclude $P(\epsilon x\,P(x))$. This inference is trivial in Isabelle, using the basic properties of Hilbert's ϵ-operator, in the form of the theorem `someI_ex`

$$\forall P\,[\exists x\,P(x) \to P(\epsilon x\,P(x))].$$

Resolving any formula, which contains an existentially quantified variable at the front, with `someI_ex` once will remove the outermost existential variable. As a result, after we perform Skolemization in the previous stage, we repeatedly resolve the resulting formula with `someI_ex` to remove all existential quantifiers.

For instance, the Isabelle lemma `subsetI` expresses the natural deduction rule for introducing the subset relation: to show $A \subseteq B$, it suffices to show that for arbitrary x, if $x \in A$ then $x \in B$. This lemma is equivalent to the first-order formula

$$\forall x\,(x \in A \to x \in B) \to A \subseteq B,$$

where A and B are implicitly universally quantified. After it is transformed to negation normal form and Skolemized, it becomes

$$\exists x\,[(x \in A \wedge x \notin B) \vee A \subseteq B].$$

[1] See `meson.ML` written by Larry Paulson

To replace the existential variable $\exists x$ by a ϵ-term, we resolve the formula above with `someI_ex`. The result is the large formula

$$\underbrace{(\epsilon x.\, x \in A \wedge x \notin B \vee A \subseteq B)}_{\epsilon-term} \in A \wedge \underbrace{(\epsilon x.\, x \in A \wedge x \notin B \vee A \subseteq B)}_{\epsilon-term} \notin B \vee A \subseteq B$$

The two ϵ-terms are identical, representing the eliminated $\exists x$.

Obviously, the ϵ-terms must be replaced by proper Skolem terms before the clauses are delivered to an automatic prover. For each ϵ-term, we generate a unique Skolem term for it. The universally quantified variables that cover the scope of a Skolem term are exactly those free variables that appear inside the corresponding ϵ-term. Therefore, it is sufficient to inspect each ϵ-term separately in generating a Skolem term. This step, which is performed outside Isabelle, yields a list of clauses of type `clause`, which are ready to be converted to TPTP format.

Please note that our clause normal form transformation function can convert not only all first-order formulae but also many higher-order formulae. If an input theorem is a higher-order formula, then the result of transforming it will be a list of higher-order clauses, which may contain terms such as function or predicate variables. However, when a generated higher-order clause is converted to internal `clause` format, an exception will be raised since type `clause` is designed for first-order clauses only.

Quantifier Miniscoping

The Skolemization procedure that we have described above pulls out all existentially quantified variables and replaces these variables by Skolem terms through the use of the axiom of choice. However, when an existentially quantified variable is pulled across a universally quantified variable, the axiom of choice does not check whether the universal variable really occurs in the sub-formula of $P(x, y)$ (in the axiom of choice formula) where y occurs. Therefore, the resulting formula may be unnecessarily large. For instance, if we Skolemize the formula

$$\forall x\, \exists y\, [P(x) \wedge Q(y)]$$

using the axiom of choice, we will get

$$\exists f\, \forall x\, [P(x) \wedge Q(f(x))]$$

although we should be able to get a simpler formula

$$\exists a\, \forall x\, [P(x) \wedge Q(a)]$$

because the scope of x does not cover the scope of y.

The problem above can be solved by the technique called *quantifier miniscoping*, which tries to reduce the scope of both universal and existential quantifiers by pushing in the quantifiers as much as possible.

We have briefly experimented with performing quantifier miniscoping before Skolemization.

In the first approach, we push in quantifiers as far inwards as possible, by resolving our input formula with the following inference rules

$$\forall P\, Q\, [\forall x\, (P(x) \wedge Q) \rightarrow (\forall x\, P(x)) \wedge Q]$$

$$\forall P\, Q\, [\forall x\, (P \wedge Q(x)) \rightarrow P \wedge (\forall x\, Q(x))]$$
$$\forall P\, Q\, [\forall x\, (P(x) \vee Q) \rightarrow (\forall x\, P(x)) \vee Q]$$
$$\forall P\, Q\, [\forall x\, (P \vee Q(x)) \rightarrow P \vee (\forall x\, Q(x))]$$
$$\forall P\, Q\, [\exists x\, (P(x) \wedge Q) \rightarrow (\exists x\, P(x)) \wedge Q]$$
$$\forall P\, Q\, [\exists x\, (P \wedge Q(x)) \rightarrow P \wedge (\exists x\, Q(x))]$$
$$\forall P\, Q\, [\exists x\, (P(x) \vee Q) \rightarrow (\exists x\, P(x)) \vee Q]$$
$$\forall P\, Q\, [\exists x\, (P \vee Q(x)) \rightarrow P \vee (\exists x\, Q(x))]$$

where $P(x)$ is any formula with some occurrence of x and Q is any formula that does not contain x. This approach recursively pushes in all quantifiers inwards.

In addition, we deliberately leave out the two extra inference rules

$$\forall P\, Q\, [\forall x\, (P(x) \wedge Q(x)) \rightarrow (\forall x\, P(x) \wedge \forall x\, Q(x))]$$

$$\forall P\, Q\, [\exists x\, (P(x) \vee Q(x)) \rightarrow (\exists x\, P(x) \vee \exists x\, Q(x))]$$

where both P and Q have occurrences of x. If we push in the quantifiers in these two situations, we may potentially duplicate the quantified variables, and thus may generate too many different Skolem terms, although some of these Skolem terms should actually be the same.

We tried this approach on several Isabelle problems and found out that some quantifiers were not pushed in as a result: if any inner quantifier cannot be fully pushed inwards then none of the outer quantifiers has a chance to be moved inwards. Therefore, we decided to implement a second version of miniscoping, where we also used the previously left out rules plus the rules we have used in the first approach.

However, even this approach leaves out some opportunities of moving quantifiers inwards. For instance, it is not sound to replace

$$\forall x\, [P(x) \vee Q(x)]$$

by

$$\forall x\, P(x) \vee \forall x\, Q(x)$$

in general. Similarly we cannot replace

$$\exists x\, [P(x) \wedge Q(x)]$$

by

$$\exists x\, P(x) \wedge \exists x\, Q(x)$$

However, we may have a formula

$$\forall x\, [P(x) \vee (Q(x) \vee R)]$$

where R does not contain x, but still $\forall x$ cannot be eliminated from the scope of R using either the first or the second method. The reason is that $Q(x) \vee R$ as a whole contains an occurrence of x, which makes it not applicable to apply the inference rules defined above.

We have investigated a solution to this problem. We decided to normalize a series of disjunctions directly under a universal quantifier, such that all disjuncts that did not contain the universally quantified variable could line up at the end. We can then perform quantifier miniscoping to this normalized formula. In the example above, after we normalize the formula to

$$\forall x\,[(P(x) \vee Q(x)) \vee R]$$

we can push in $\forall x$ inwards and get

$$\forall x\,[P(x) \vee Q(x)] \vee R.$$

In a similarly way, we can normalize a series of conjunctions directly under an existential quantifier.

We have briefly carried out some implementations in order to normalize formulae. We perform this normalization inside Isabelle logic by forward resolution and rewriting. We have normalized most of the formulae except some deeply nested conjunctions and disjunctions. For instance, the formula

$$\exists x\,[(A(x) \wedge B \wedge C(x)) \wedge D \wedge E(x)]$$

can be normalized, then have $\exists x$ pushed in and become

$$\exists x\,[A(x) \wedge C(x) \wedge E(x)] \wedge D \wedge B$$

Moreover, the formula

$$\forall x\,[(A(x) \vee B(x)) \vee C \vee D(x)]$$

is normalized and then transformed to

$$\forall x\,[(A(x) \vee B(x)) \vee D(x)] \vee C$$

by quantifier miniscoping.

Although quantifier miniscoping can improve the performance of Skolemization and hence later clause normal form transformation, we decided we should move on to the rest of the integration tasks rather than concentrating on further improving the quality of quantifier miniscoping. The miniscoping we have done so far should be enough for the quality of our Isabelle formulae transformation.

After quantifier miniscoping is performed, we conduct Skolemization by pulling out existential quantifiers and applying the axiom of choice, as we have described in the first part of this section.

4.4.2 Preprocessing of Elimination Rules

Recall (§3.2.1) that we need to transform an elimination rule

$$\forall P\,[A \to \forall \mathbf{x}_1\,(\mathbf{B}_1 \to P) \to \ldots \to \forall \mathbf{x}_n\,(\mathbf{B}_n \to P) \to P]$$

into an equivalent form

$$A \to (\exists \mathbf{x}_1\,\mathbf{B}_1 \vee \ldots \vee \exists \mathbf{x}_n\,\mathbf{B}_n)$$

in order to remove the predicate variable P.

This transformation must take place before the elimination rules can be converted to clause normal form by the functions that we have described in the previous section.

I have attempted to use both rewrite rules and forward proof to transform the elimination rules into their equivalent format. It turns out to be complicated to define a few inference rules that can deal with the transformation of all elimination rules automatically.

As a result, we adopt an alternative method that still ensures correctness, which means the result of this transformation on an elimination rule is still a valid Isabelle theorem, of type `thm`. We have defined a function `transform_elim` of type `thm -> thm` to transform any elimination rule into its equivalent form. It performs two major steps.

1. From a given elimination rule, it constructs an Isabelle term (of type `term`) that represents the structure of the first-order formula equivalent to the rule. This is straightforward programming.

2. It then invokes an Isabelle function `prove_goalw_cterm`. This function takes a term and a tactic and then proves the given term to be an Isabelle theorem using the supplied tactic. Here, `prove_goalw_cterm` receives the constructed term from the previous step and a proof, which begins by applying the elimination rule under consideration. The remaining proof steps involve elementary first-order reasoning.

As an example, consider the elimination rule `UnionE`. Intuitively, it says that if $A \in \bigcup C$, then there is some x such that $A \in x$ and $x \in C$. It can be expressed in higher-order logic as

$$\forall P\, A\, C\, [A \in \bigcup C \rightarrow \forall x\, (A \in x \wedge x \in C \rightarrow P) \rightarrow P].$$

Apply function `transform_elim` to it and the result of our transformation is a first-order theorem:

$$\forall A\, C\, [A \in \bigcup C \rightarrow \exists x\, (A \in x \wedge x \in C)]$$

This theorem is finally converted to clause normal form by `isa_cls` as two clauses

$$A \notin \bigcup C \vee (\epsilon x.\, A \notin \bigcup C \vee x \in C \wedge A \in x) \in C$$

and

$$A \notin \bigcup C \vee A \in (\epsilon x.\, A \notin \bigcup C \vee x \in C \wedge A \in x)$$

where A and C are implicitly universally quantified. Since the procedure above is carried out inside Isabelle logic, each clause has the type `thm`, which means it is an Isabelle theorem.

When we convert an Isabelle rule to first-order clauses, we must check whether it is an elimination rule first. This is achieved easily by inspecting the term structure of the rule. If it is an elimination rule, then it must go through the preprocessing phase before it can be converted to clauses by clause normal form transformation function `isa_cls`.

4.4.3 Isabelle Simplification Rules

The functions defined above can be used directly to convert almost all Isabelle theorems to clause normal form. However, Isabelle's simplification rules require some special considerations.

There are three types of simplification rules: boolean equalities, non-boolean equalities and simplification rules generated from non-equalities.

- For boolean equalities of the form $P \equiv Q$, we simply transform it to the equivalent $(P \to Q) \wedge (Q \to P)$ and then convert the resulting formula into clause normal form using the transformation procedure defined in the previous section.

- Non-boolean equalities are regarded as simple equalities and are converted to clauses accordingly.

- Simplification rules generated from non-equality theorems usually have a form $P \equiv \bot$ or $P \equiv \top$. We must convert them to their original non-equality theorem format and thus remove the $\top$ and $\bot$ in them before transforming them to clauses. In the first approach, I implemented a transformation inside Isabelle, by first proving auxiliary lemmas such as
$$\forall P\,[P \equiv \bot \to \neg P]$$
and
$$\forall P\,[P \equiv \top \to P].$$
Afterwards, we can resolve the simplification rules with these lemmas, so that $P \equiv \bot$ and $P \equiv \top$ are replaced by $\neg P$ and P respectively.

  At the time of writing this dissertation, Larry Paulson has implemented another method, which simplifies these simplification rules so that $\bot$ and $\top$ are erased during transformation to clauses. With this simplification, $P \equiv \bot$ is replaced by $\neg P$ and $P \equiv \top$ is replaced by P.

4.5 Isabelle Local Assumptions

Isabelle users may make assumptions during a proof by command, such as `assume`. These assumptions should be treated as part of the negated conjecture. If an assumption is ϕ and a conjecture is θ, then converting the negated goal $\neg(\phi \to \theta)$ into clauses is equivalent to converting ϕ and $\neg\theta$ to clauses separately. We have defined the function `isar_atp_h` to convert Isabelle local assumptions — taken from a current proof context — to clauses.

In Isabelle, local assumptions are stored as theorems, with type `thm`. Therefore, we use similar techniques to convert them to clause normal form as we do for existing Isabelle theorems. However, when we convert the generated Isabelle clauses to our internal clause format of type `clause`, we label them as *negated conjecture* clauses.

4.6 Isabelle Subgoals

When it is time to send goals to automatic provers, function `isar_atp_goal` negates all current subgoals and converts each of the negated subgoals to clauses.

Recall that Isabelle represents its proof state (§2.6.3) as a meta-theorem of type `thm`

$$[\![\psi_1; \ldots \psi_n]\!] \Longrightarrow [\![C]\!]$$

where C is the main goal (i.e. the theorem to be proved) and $\psi_1, \ldots, \psi_n$ are the current subgoals. In order to use Isabelle inference rules to perform negation and then clause normal form transformation to all the current subgoals, we have to operate on the entire

proof state. As we have to operate on a proof state, a backward reasoning style is more suitable, which means we need a tactic to perform negation normal form and clause normal form transformation on subgoals.

We define a tactic `atp_tac_tfrees` such that "`atp_tac_tfrees` k" negates the k-th subgoal and converts the negated subgoal to clauses. This operation consists of the following steps.

- Resolve the subgoal with a contrapositive inference rule

  $$\forall P\,[(\neg P \to \bot) \to P]$$

  This transforms the k-th subgoal ψ_k to an equivalent formula

  $$\neg\psi_k \to \bot$$

  where the assumption is the negated subgoal.

- Apply `skolemize_tac` to perform Skolemization on the new k-th subgoal $\neg\psi_k \to \bot$. In contrast to Skolemization on theorems, which requires a further use of the axiom of choice, this tactic simply drops the existential quantifiers after having pulled all of them to the front, using the axioms of choice. The result of this step is a new k-th subgoal $\psi'_k \to \bot$.

- Apply a tactic to convert ψ'_k to clauses.

- Finally, convert the clauses to TPTP format and write to a designated file.

As there may be more than one subgoal when an automatic prover is called, function `isar_atp_goal` repeatedly applies `atp_tac_tfrees` on all of the subgoals. Since we do not wish to leave an observable effect to the other parts of Isabelle program, and in particular we do not wish the Isabelle user to notice what has taken place, when function `isar_atp_goal` returns, the proof state sent to function `isar_atp_goal` is returned unchanged. This also ensures the Isabelle user can continue solving the original goal (with the proof state unchanged) if necessary.

4.7 Global Theorems and Other Facts

In the previous sections, we have described how to locate and convert Isabelle local assumptions, local theorems and negated goals to clause normal form. The only remaining items are the global theorems and Isabelle's type constructors' arities and type class relations. They should be converted to clauses as well.

Global theorems are usually stored with each Isabelle theory file. Therefore, we define the following functions.

- `cnf_classical_rules_thy` extracts all classical rules from a named theory and converts them to clauses of type `thm`.

- `clausify_classical_rules_thy` is similar to `cnf_classical_rules_thy` but also converts the Isabelle clauses to axiom clauses of type `clause`.

- `cnf_simpset_rules_thy` extracts all simplification rules from a named theory and converts them to clauses of type `thm`.

- `clausify_simpset_rules_thy` is similar to `cnf_simpset_rules_thy` but also converts the Isabelle clauses to axiom clauses of type `clause`.

In addition, Isabelle stores type class relations and type constructors' arities with theory files. Therefore, we define the following functions.

- `arity_clause` converts the multiple arities of a type constructor into a list of clauses of type `arityClause`.

- `arity_clause_thy` retrieves all the arities of all the type constructors from a named theory and converts them into a list of clauses of type `arityClause`.

- `classrel_clause` converts each $(subclass, superclasses)$ pair into a list of clauses of type `classrelClause`: each type class may have more than one direct superclass.

- `classrel_clauses_thy` converts each type class relation defined in a theory to a clause of type `classrelClause`.

The clauses generated from global theorems, type class relations and type constructors' arities can be permanently written to designated axiom files so that an automatic prover can read them whenever the prover attempts to prove a goal.

4.8 Calling Automatic Provers from ML Proofs

We have described our implementation approach to integrating Isar with any automatic prover. The functions that we have defined are mainly used for the Isar interface as well. We have put emphasis on the Isar interface largely because many more Isabelle users are now working under it than under the ML interface. However, we still experimented an integration of the ML interface with an automatic prover. In fact, we carried out our experiment on the ML interface even before we integrated Isar with automatic provers.

Integrating the ML interface with automatic provers is easier than integrating the Isar interface with automatic provers: for the ML interface, we only have to consider how to negate a named subgoal and convert it to clauses and finally write the clauses to files. However, this single task was also crucial for the Isar interface: the operation on goals is essentially the same regardless whether we are under the ML or the Isar interface.

As remarked above, we have to use tactics to negate a goal and transform it into clauses. Therefore, we defined a tactic `atp_tac`, which could be applied using the `by` command of the ML interface to perform the required operation to the named subgoal, and to write the negated conjecture clauses to a file.

Having briefly experimented with the ML interface, we used most of the functions, which we defined for `atp_tac`, as building blocks of `isar_atp_goal`. It performs a similar function in the Isar interface, but with more functionalities such as converting all subgoals, rather than a named one, to clauses. We later integrated `isar_atp_goal` with the rest of our program — such as extracting and converting local assumptions and local theorems at an appropriate point during a proof — to make the prover calling procedure completely

transparent, without users' interaction. The effect of this prover calling procedure running under the Isar interface is entirely different to the effect achieved from `atp_tac` in the ML interface.

However, we did not continue integrating the ML interface with automatic provers any further for two reasons.

First, unlike the Isar interface, the ML interface does not have a notion of proof modes. All proofs are linear and proofs are constructed by tactics. As a result, a goal is always subject to immediate changes. According to the analysis we have given previously, it is not feasible to let Isabelle call an automatic prover without a user's instruction if a goal may be changed instantly. This means that we cannot fully automate the automatic prover calling procedure with the ML interface, but this is one of the objectives of our integration. Therefore, the ML interface does not seem to be the most suitable Isabelle interface for our integration.

Second, the ML interface does not support structured proofs, and hence does not support local theorems. Consequently, it seems difficult to allow users to declare some local theorems, which are then sent to automatic provers. We have attempted to simulate the use of local theorems by defining a variant of the tactic `atp_tac` as `atp_ax_tac`. This new tactic can be supplied with a list of Isabelle theorems, which the user may wish to send to an automatic provers as local theorems. Tactic `atp_ax_tac` not only negates the goal and converts it negated conjecture clauses, but also transforms the supplied list of theorems to axiom clauses. All of these clauses are written to some designated files. However, this method is not consistent with our initial objective of not asking users to specify which theorems to use.

Although we are not going to integrate the ML interface with automatic provers any further, the tactics `atp_tac` and `atp_ax_tac` may still be useful for people who wish to work under the ML interface, perhaps for a quick proof of a goal. They may also be useful for our program debugging in the future if some more functions are added.

4.9 Concluding Remarks

In this chapter, we have described our approach to implement a program that runs between Isabelle and any background automatic prover and directs the communication from Isabelle to the automatic prover.

This program is closely integrated with Isabelle's Isar interface and extracts proof goals and all necessary Isabelle theorems and facts at an appropriate point during a proof. After getting these items, our program translates them to clauses and writes the clauses to designated files, which can be read by an automatic prover running in the background. Crucially, this process happens without any user's interaction and is invisible from the user's point of view.

A major task of this program is to automatically translate Isabelle formulae into clause normal form, which has to be performed inside Isabelle logic in order to ensure correctness and also to ensure the later proof construction is possible. We have implemented a clause normal form transformation function that converts Isabelle/HOL goals or theorems to clauses, including both first-order formulae and many higher-order formulae. Although the higher-order clauses generated from higher-order formulae will not be passed to an automatic prover based on first-order logic, these higher-order clauses may be useful if we

integrate other automatic provers, which are based on higher-order logic, with Isabelle in the future.

Furthermore, the clause normal form transformation function preserves type information, which can be used by any first-order automatic prover.

In order to make our integration more scalable, we have defined three internal clause formats, to which we translate all Isabelle formulae and type information. Since our target automatic provers work for first-order logic, the internal clause formats encode first-order clauses only. If a higher-order clause is generated from an Isabelle formula, then an exception will be raised. These internal clause formats can be converted to any automatic prover-specific format easily. We have also implemented a conversion to TPTP format.

In addition to integrating the Isar interface with automatic provers, we have also experimented an integration of the ML interface with automatic provers. We have defined two tactics for this. A user can use these tactics to send proof goals and named theorems to an automatic prover for proofs.

Our program finishes its job after it writes all the generated clauses to the designated files. This almost completes the entire automatic prover calling procedure. The only remaining task is to use interprocess communication mechanism to inform a process of a background automatic prover about the problem files being ready to be read. This task can be achieved by system programming. This process communication work has been done by Claire Quigley.

Chapter 5

Reducing The Number of Clauses

In the previous chapters, we have described our approach to accomplishing all the essential tasks that need to be done for calling automatic provers from Isabelle proofs. We have designed a practical method to translate Isabelle/HOL and Isabelle/ZF problems to first-order clauses. We have also implemented this translation. We have integrated this translation into a program that automatically extracts all proof data from Isabelle at an appropriate point of a proof, converts the data to clauses and sends them to background automatic provers. The work we have done so far has achieved a working link that directs communication from Isabelle to any first-order resolution prover.

From the previous experiments, we found having a large number of clauses harmed the performance of many automatic provers. In this chapter, we describe our investigation to this problem and the attempts we have made to solve it. This research work is intended to improve the performance of the integration.

5.1 Problems of Large Numbers of Clauses

The size of the search space — in terms of the number of clauses — plays a significant role in determining the performance of a resolution-based automatic prover. In order for a proof to be found, the prover must be able to pick up relevant clauses to participate in resolution steps. However, in the presence of a large number of clauses, it is difficult for the prover to decide which clauses should be selected. What makes the task even more difficult is that if some inappropriate clauses are selected to resolve, then the generated clauses will be useless for the proof as well. These newly generated irrelevant clauses can only enlarge the search space further.

The problem caused by the large search space first revealed itself during our previous experiments on formalizing Isabelle/ZF. We found that many resolution-based automatic provers were incapable of proving goals when we gave them a large set of axiom clauses, where many of these axiom clauses may have been irrelevant. Having found the problem, we carried out another set of experiments with a specific aim: to examine how well can some of the best-known automatic provers cope with a large set of axioms (§3.3.3).

During this set of experiments, we found fifteen Isabelle goals that were particularly difficult to be proved in the presence of the full axiom set (around 129 to 160 axiom clauses). I provided these problems to the TPTP library. The problems are COL088-1 to COL100-2 and SET787-1, SET787-2. Other researchers and I made ninety attempts on

these goals, using various provers. The results are summarized below.

- I tried to prove the fifteen problems using both Vampire (v6.03) and SPASS, with time limit set to 60 seconds. For the experiments on Vampire, I used the combination of five settings described in §3.3.1. Two problems were proved by Vampire v6.03 (which supports literal annotations; version 5.6 could not prove them) within the time limit.
  - COL088-2: 17.1 seconds.
  - COL089-2: 14.2 seconds.
- Geoff Sutcliffe tried them on three automatic provers — E, SPASS and Vampire — with his tool that is specifically designed to eliminate redundant clauses. Three problems were proved by SPASS within 300 seconds.
  - SET787-2: 154 seconds.
  - COL091-2: 75 seconds.
  - COL099-2: 43 seconds.
- Gernot Stenz ran the same problems on E-SETHEO. Two problems were proved.
  - COL091-2: 1 second.
  - COL099-2: 2 seconds.

Among the fifteen problems, five were proved. Moreover, out of the $15 \times 6 = 90$ proof attempts, seven succeeded.

Since we have used a wide variety of automatic provers to prove the goals, and many of them are leading provers, the experimental results above indicate that the large numbers of clauses may be a problem to most resolution provers.

5.1.1 The Cause of Large Numbers of Axiom Clauses

Clearly, the large number of axiom clauses has a negative impact on the proof performance of resolution provers. Furthermore, these axiom clauses are generated from Isabelle theorems. We need to identify the cause of the large number of axiom clauses that are present to an automatic prover in order to solve the problem. We have found that there are two major reasons.

First, interactive provers are usually used to formally specify and verify the correctness of complex systems. Therefore the verification usually starts by proving hundreds of lemmas to support the ultimate proofs of system properties. There is no exception with the use of Isabelle. Take the Isabelle/HOL theory `Main.thy` as an example. `Main.thy` is usually the starting specification theory, upon which other specifications are built. This theory contains numerous already proved lemmas: 427 classical reasoning rules and 1499 simplification rules. If we define a theory to be a descendant of `Main.thy`, then during a proof development in this descendant theory, there could be many more theorems proved and stored in the classical reasoner and equality reasoner. In addition, each of the Isabelle theorems is capable of generating multiple clauses. The total number of axiom clauses could be overwhelming.

On the other hand, if we look at Isabelle proof goals more closely, we will see that nearly each goal is proved by a small number of theorems — usually fewer than ten theorems. The great majority of theorems available during a proof are irrelevant. If the clauses generated from these irrelevant theorems are allowed to participate in resolution steps, the search space would explode. Consequently, we should try to help an automatic prover ignore the irrelevant axiom clauses or become less sensitive to them. Alternatively, we could also try to reduce the irrelevant theorems presented to an automatic prover.

In addition to the large number of theorems that are always present at any one time during a proof, some of the particularly complex theorems are making the situation worse. These theorems are used for *rule inversion*, which is a case analysis on an inductive definition. In Isabelle, many constants are defined by inductive definitions, which specify inductively defined sets. These inductive definitions are made up of introduction rules. When an inductive definition is made, Isabelle generates a set of elimination rules, which are used for case analysis based on the patterns of expressions belonging to that set. These elimination rules are subsequently used for rule inversion. Rule inversion is important for reasoning about operational semantics. It helps to identify which of the many rules of an operational semantics definition may have caused a given event. However, such a rule may generate a large number of clauses, if the inductive definition concerned is made up from many introduction rules. An example of such an elimination rule is `Ap_contractE`, which is defined in Isabelle/ZF theory file `Comb.thy`. Since it is an elimination rule, we need to convert it to an equivalent first-order formula, which is expressed as

$$\begin{aligned}\forall p\, q\, r\, [(app(p,q) \twoheadrightarrow r) \rightarrow \\ [r \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge p = app(\mathbf{K}, r)] \vee \\ \exists pa\, qa\, [pa \in \mathbf{comb} \wedge qa \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge \\ r = app(app(pa,q), app(qa,q)) \wedge p = app(app(\mathbf{S}, pa), qa)] \vee \\ \exists qa\, [p \twoheadrightarrow qa \wedge q \in \mathbf{comb} \wedge r = app(qa,q)] \vee \\ \exists qa\, [q \twoheadrightarrow qa \wedge p \in \mathbf{comb} \wedge r = app(p,qa)]].\end{aligned}$$

In this theorem, $\twoheadrightarrow$ represents an inductively defined reduction relation between combinators, and is written as an infix operator. The inductively defined set of combinators **comb**, is constructed from basic combinators **K**, **S** and the application of combinators through the function *app*. Moreover, since Isabelle/ZF is untyped, $r \in \mathbf{comb}$ represents a kind of type constraint: r is a variable of type **comb**.

Theorem `Ap_contractE` represents a case analysis on the inductive definition of $\twoheadrightarrow$, which consists of four introduction rules. It says that if $app(p,q) \twoheadrightarrow r$, then from the four cases of the inductive definition of $\twoheadrightarrow$ we can deduce the forms of and relations between variables p, q and r. The four cases are represented in this formula as four disjuncts.

The disjuncts above can generate three, five, three and three clauses respectively. Therefore, using the standard clause normal form (CNF) transformation, `Ap_contractE` will generate $3 \times 5 \times 3 \times 3 = 135$ clauses. We have tried to prove four goals, which required the use of `Ap_contractE`. Unfortunately, none of the goals were proved, when given 135 clauses. Clearly, we should try to reduce the number of clauses generated from these elimination rules.

5.1.2 Possible Solutions

Having found the cause of the generation of large numbers of clauses, we look at possible solutions to the two major problems mentioned above.

Minimizing Generated Clauses

As we have seen, some Isabelle theorems can generate huge numbers of clauses during transformation into CNF. This suggests that we should try to minimize the number of clauses generated via a technique called *formula renaming* [39].

Formula renaming involves defining new predicates in order to minimize the number of clauses being generated from the transformed formula. It preserves consistency and thus inconsistency. Therefore this technique is suitable for resolution theorem proving. We describe how we have used it and implemented it in sections §5.2 and §5.3.

Removing Irrelevant Theorems and Axiom Clauses

We should not only reduce the number of clauses generated during CNF transformation but also prevent irrelevant axiom clauses from being considered by resolution provers.

Obviously, we should not ask Isabelle users to select the theorems that are relevant to a proof goal, because the objective of integrating Isabelle and automatic provers is to improve Isabelle automation and reduce users' interaction. Therefore we need to have those irrelevant clauses to be removed automatically.

Automatically filtering out irrelevant lemmas from relevant ones is difficult. One attempt that has been made to perform automatic removal of irrelevant lemmas was carried out in the KIV-$_3T^AP$ integration. However, so far no satisfactory solution has been found. Nevertheless, we felt this was a useful research and therefore we carried out some preliminary investigations into the problem. The objective was to find some promising research directions.

The theorems are generated and sent from Isabelle. Subsequently, they are received as axiom clauses and used by an automatic prover. Therefore we could approach the problem from two directions:

- We send all the available theorems to an automatic prover using prover settings so that the irrelevant axiom clauses can be tolerated by the prover.
- We may be able to design an algorithm or heuristic to remove irrelevant theorems from a set of theorems available in a current context. Afterwards, we shall send the remaining theorems to an automatic prover.

The responsibility of removing irrelevant information could lie on either sides of our integration — Isabelle and our target automatic provers. We have experimented with both of these approaches, which are summarized below.

- We have carried out experiments on Vampire in order to find some suitable settings, which can deal with large numbers of axiom clauses more effectively.
- We have used another automatic prover, E, which has particular strength in distinguishing useful clauses from useless clauses, in order to see whether it can handle large numbers of axiom clauses.

- We have designed a heuristic to discard some irrelevant Isabelle theorems from the existing classical set and simplifier set, and then only give those possibly relevant theorems to an automatic prover.

We describe our attempts in sections §5.4 and §5.5.

5.2 Using Formula Renaming

During the transformation into clauses, the distributive law

$$(A \wedge B) \vee C \simeq (A \vee C) \wedge (B \vee C)$$

has to be applied repeatedly so that disjunction is pushed in until it applies directly to literals. The exhaustive application of the law can duplicate formulae and hence generate a vast number of clauses. In this case, all clauses generated from formula C are being duplicated.

In a more general case, consider a formula

$$\phi_1 \vee \forall x\, \phi_2$$

and for simplicity, let us assume x is the only free variable in ϕ_2. Suppose during the CNF transformation, ϕ_1 generates m clauses and ϕ_2 generates n clauses. Then by applying the distributive law, $\phi_1 \vee \forall x\, \phi_2$ will generate $m \times n$ clauses.

Formula renaming [39] is designed to solve this duplication of clauses by introducing new predicates to replace sub-formulae that may be duplicated. In the example above, we can introduce a new one-place predicate P to replace ϕ_2: x is free in ϕ_2 thus P should depend on x as well. With this replacement, the original formula becomes

$$\phi_1 \vee \forall x\, P(x).$$

Moreover, to ensure the consistency is preserved after we *rename* ϕ_2 to $P(x)$, we need a *definition* of P, which serves as a constraint and is therefore added as a conjunction to the new formula $\phi_1 \vee \forall x\, P(x)$. In this case, the definition of P is the formula $\forall x\, [P(x) \to \phi_2]$. Consequently, the result of applying formula renaming to the original formula is

$$[\phi_1 \vee \forall x\, P(x)] \wedge \forall x\, [P(x) \to \phi_2].$$

The new formula will generate $m + n$ clauses. Therefore formula renaming will reduce the number of clauses generated as long as there is

$$m + n < m \times n.$$

Formula renaming should be applied before CNF transformation, and it can be applied to any arbitrary formula. For instance, the replaced formula does not have to be a disjunct — it can be in any position such as an antecedent or a consequent of an implication, or it can be a branch of a conjunction. In addition, formula renaming can also be applied to replace a sub-formula of any form. Furthermore, the definition of each newly introduced predicate depends on the polarity of the replaced sub-formula in the top level formula and the new predicate must depend on all the free variables occurring in that sub-formula.

Definition 8. Formula renaming on formula P, replacing the sub-formula Q, consists of the following steps.

- Pick a new n-place predicate symbol R and use the atomic formula $R(x_1, \ldots, x_n)$ to replace Q, where $x_1, \ldots, x_n$ are all the free variables in Q. Suppose with this replacement P becomes P'.
- Put an additional formula D to be in conjunction with P'. The additional formula is seen as the definition of R and is defined as follows.
  - If Q has positive polarity in P, i.e. it occurs positively in P, then
  $$D \equiv \forall x_1, \ldots, x_n \, [R(x_1, \ldots, x_n) \rightarrow Q] \tag{5.1}$$
  - If Q has negative polarity in P, i.e. it occurs negatively in P, then
  $$D \equiv \forall x_1, \ldots, x_n \, [Q \rightarrow R(x_1, \ldots, x_n)] \tag{5.2}$$
  - If Q occurs on one side of an equality and thus has neither purely negative nor purely positive polarity in P, then
  $$D \equiv \forall x_1, \ldots, x_n \, [R(x_1, \ldots, x_n) \leftrightarrow Q] \tag{5.3}$$
- The result of the formula renaming transformation on P is
$$P' \wedge D$$

Moreover, for the application of formula renaming to be effective, the resulting formula $P' \wedge D$ must generate fewer clauses than P does during the subsequent clause form transformation. Therefore the condition for formula renaming to replace Q is

$$\varphi(P' \wedge D) < \varphi(P) \tag{5.4}$$

where $\varphi(X)$ represents the number of clauses generated by formula X.

For our problems, we have decided to apply formula renaming to theorems after they are converted to negation normal form, before the distributive law is applied. Since we never need to rename a literal (an atomic formula or a negated atomic formula), each replaced sub-formula has positive polarity in the top-level formula. As a result, each definition formula follows equation 5.1.

5.2.1 The Specialized Version of Formula Renaming

Our previous experiments show that most of the Isabelle theorems that generate large numbers of clauses are elimination rules used for case analysis. Therefore, we decided to apply formula renaming to those elimination rules first, in order to examine whether formula renaming could be helpful to improve the performance of automatic provers.

In this version of formula renaming, I designed a method that targets the cause of the generation of large numbers of clauses — the disjuncts representing the cases. This

specialized version does not require much computation and is easy to apply by hand. Therefore, I have used it in the experiments.

Recall that elimination rules in Isabelle are represented as

$$\forall P\,[A \to \forall \mathbf{x}_1\,(\mathbf{B}_1 \to P) \to \ldots \to \forall \mathbf{x}_n\,(\mathbf{B}_n \to P) \to P]$$

where all free variables are implicitly universally quantified. Before CNF transformation, we need to transform an elimination rule to an equivalent formula

$$A \to (\exists \mathbf{x}_1\,\mathbf{B}_1 \vee \ldots \vee \exists \mathbf{x}_n\,\mathbf{B}_n).$$

After it is converted to negation normal form, it becomes

$$\neg A \vee (\exists \mathbf{x}_1\,\mathbf{B}_1 \vee \ldots \vee \exists \mathbf{x}_n\,\mathbf{B}_n).$$

The standard step to do next is conversion to clauses. For an elimination rule, A is usually an atomic formula so it generates one clause. Moreover, suppose each $\mathbf{B}_i$ generates k_i clauses, then after this elimination rule is fully converted to clauses, we will have $k_1 \times \ldots \times k_n$ clauses generated. Typical to most theorems used for case analysis, n is quite large (usually more than three cases) and each k_i may be big too. As a result, $k_1 \times \ldots \times k_n$ is likely to be a large number.

In order to prevent the duplication of disjunctions, I replace a disjunct $\exists \mathbf{x}_i\,\mathbf{B}_i$ by a new atomic formula S_i provided $\mathbf{B}_i$ is not a literal. Moreover, for formula renaming to replace those non-literal disjuncts, we need to check that for each $k_1, \ldots, k_m$ from each non-literal disjunct

$$1 + k_1 + \ldots + k_m < k_1 \times \ldots \times k_m. \tag{5.5}$$

Definition 9. For an elimination rule that is in negation normal form

$$\neg A \vee (\exists \mathbf{x}_1\,\mathbf{B}_1 \vee \ldots \vee \exists \mathbf{x}_n\,\mathbf{B}_n), \tag{5.6}$$

if the condition (5.5) is satisfied, then we perform formula renaming as follows.

- For each $\exists \mathbf{x}_i\,\mathbf{B}_i$ such that $\mathbf{B}_i$ is not a literal, we introduce a new predicate symbol R_i and use the atomic formula $R_i(y_1, \ldots, y_{l_i})$ to replace it, where $y_1, \ldots, y_{l_i}$ are the free variables in $\exists \mathbf{x}_i\,\mathbf{B}_i$.
- The definition formula of R_i is

$$\forall y_1\ \ldots\ y_{l_i}[\neg R_i(y_1, \ldots, y_{l_i}) \vee \exists \mathbf{x}_i\,\mathbf{B}_i].$$

As we can see, this specialized formula renaming renames *each* case represented by a disjunct that contains more than one literal. A closer look at the formula above reveals that our formula renaming has transformed a case analysis theorem into a desirable format such that the output theorem preserves the information about how the input theorem is used in Isabelle for case analysis. This information will be useful for an automatic prover to find a proof. From A we can deduce n possibilities — $\exists \mathbf{x}_1\,\mathbf{B}_1, \ldots, \exists \mathbf{x}_n\,\mathbf{B}_n$. These cases do not overlap and thus should be considered separately during a proof. In order to prevent these cases from being intertwined in a resolution proof, we should forbid

the literals generated from different cases to get into a same clause. Clearly, this has been achieved by using new atomic formulae $S_1 \ldots S_n$ to abbreviate the n cases and then separate the cases by conjunctions.

Although this version of formula renaming may not generate the minimum number of clauses, it is sufficiently good to significantly reduce the number of clauses and can be used for our experiments.

Formula renaming on elimination rules has been shown very effective. Take theorem `Ap_contractE` as an example. Using our specialized formula renaming to rename it, we get

$$\begin{aligned}\forall p\, q\, r\, [(\neg(app(p,q) \twoheadrightarrow r) \vee (R_1 \vee R_2 \vee R_3 \vee R_4)) \wedge \\ (\neg R_1 \vee (r \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge p = app(\mathbf{K}, r))) \wedge \\ (\neg R_2 \vee \exists pa\, qa\, [pa \in \mathbf{comb} \wedge qa \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge \\ r = app(app(pa,q), app(qa,q)) \wedge p = app(app(\mathbf{S}, pa), qa)]) \wedge \\ (\neg R_3 \vee \exists qa\, [q \twoheadrightarrow qa \wedge p \in \mathbf{comb}]) \wedge \\ (\neg R_4 \vee \exists qa\, [q \twoheadrightarrow qa \wedge p \in \mathbf{comb} \wedge r = app(p, qa)])] \quad (5.7)\end{aligned}$$

where each R_i abbreviates a new atomic formula $Q_i(p,q,r)$. This formula will generate $1+3+3+5+3 = 15$ clauses during CNF transformation. In comparison, the original formula generates 135 clauses. We have achieved a huge reduction.

In addition, the experimental results showed that formula renaming gave positive effect on the proof search of automatic provers. Among the four previously failed proof goals that required the use of `Ap_contractE`, three goals were proved after we used formula renaming.

5.2.2 A Harder Example of Formula Renaming

Although we have been able to use formula renaming to reduce the number of clauses generated from some elimination rules, problems involving case analysis are still hard for most resolution provers. An example is the Isabelle problem `S2_parcontractD` from the Isabelle/ZF theory file `Comb.thy`. This problem requires a theorem `Ap_parcontractE`, which generates 90 clauses without formula renaming and 14 clauses with formula renaming. This problem was particularly difficult for most resolution provers, and has been put into the TPTP library as COL094-1 and COL094-2: COL094-1 contains a small set of necessary axiom clauses whereas COL094-2 contains a large set of axiom clauses, many of them irrelevant. Both COL094-1 and COL094-2 have clauses generated from `Ap_parcontractE` with formula renaming.

The problem `S2_parcontractD` proves a property about parallel contraction $\rightsquigarrow$ between combinators. We define the relation first.

Definition 10. The parallel contraction relation $\rightsquigarrow$ (written as infix) between combina-

tors is inductively defined, using four introduction rules, as follows:

$$p \in \mathbf{comb} \Longrightarrow p \rightsquigarrow p \tag{5.8}$$

$$p \in \mathbf{comb} \wedge q \in \mathbf{comb} \Longrightarrow app(app(\mathbf{K}, p), q) \rightsquigarrow p \tag{5.9}$$

$$p \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge r \in \mathbf{comb} \Longrightarrow app(app(app(\mathbf{S}, p), q), r) \rightsquigarrow app(app(p, r), app(q, r)) \tag{5.10}$$

$$p \rightsquigarrow q \wedge r \rightsquigarrow s \Longrightarrow app(p, r) \rightsquigarrow app(q, s) \tag{5.11}$$

where **comb**, **S**, **K** and *app* have the same meanings as in `Ap_contractE` above. Moreover, variables p, q, r and s are all universally quantified.

The problem `S2_parcontractD` is stated as

$$(app(app(\mathbf{S}, p), q) \rightsquigarrow r) \Longrightarrow \exists p'\, q'\, [r = app(app(\mathbf{S}, p'), q') \wedge p \rightsquigarrow p' \wedge q \rightsquigarrow q'].$$

Its proof requires the use of the elimination rule `Ap_parcontractE`, which after converted to an equivalent first-order form becomes

$$\begin{aligned} \forall p\, q\, r\, [(app(p, q) \rightsquigarrow r) \rightarrow \\ & [app(p, q) \in \mathbf{comb} \wedge r = app(p, q)] \vee \\ & [r \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge p = app(\mathbf{K}, r)] \vee \\ & \exists pa\, qa\, [pa \in \mathbf{comb} \wedge qa \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge \\ & r = app(app(pa, q), app(qa, q)) \wedge p = app(app(\mathbf{S}, pa), qa)] \vee \\ & \exists qa\, s\, [p \rightsquigarrow qa \wedge q \rightsquigarrow s \wedge r = app(qa, s)]]. \end{aligned} \tag{5.12}$$

`Ap_parcontractE` says what one can deduce from a parallel contraction $app(p, q) \rightsquigarrow r$: there are four possibilities based on the four cases of the inductive definition of $\rightsquigarrow$. Each case is represented by a disjunct.

In order to prove `S2_parcontractD`, we first need to deduce the forms of and the relations between r, p, q and **S**, from the assumption $app(app(\mathbf{S}, p), q) \rightsquigarrow r$. This is achieved by case analysis using the theorem `Ap_parcontractE`. Subsequently, assuming each possible case that can be deduced from $app(app(\mathbf{S}, p), q) \rightsquigarrow r$, we need to show the conclusion of `S2_parcontractD`,

$$\exists p'\, q'\, [r = app(app(\mathbf{S}, p'), q') \wedge p \rightsquigarrow p' \wedge q \rightsquigarrow q']. \tag{5.13}$$

We now give a brief proof.

Proof. From the theorem `Ap_parcontractE`, there are four possible cases that we can deduce from $app(app(\mathbf{S}, p), q) \rightsquigarrow r$, we need to prove that each case implies the conclusion (5.13).

case 1.

$$app(app(\mathbf{S}, p), q) \in \mathbf{comb} \wedge r = app(app(\mathbf{S}, p), q)$$

This case implies the conclusion (5.13), since we can assign $p' = p$ and $q' = q$. By the first rule (5.8) of the inductive definition of $\rightsquigarrow$, we have $p \rightsquigarrow p$ and $q \rightsquigarrow q$.

case 2.

$$r \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge app(\mathbf{S}, p) = app(\mathbf{K}, r)$$

By the freeness properties of **comb**, $app(\mathbf{S}, p)$ is not equal to $app(\mathbf{K}, r)$. Therefore this case is false, which trivially implies the conclusion (5.13).

case 3.

$$\exists pa\, qa\, [pa \in \mathbf{comb} \wedge qa \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge r = app(app(pa, q), app(qa, q)) \wedge app(\mathbf{S}, p) = app(app(\mathbf{S}, pa), qa)]$$

Similar to the reasoning in **case 2.** above, since

$$app(\mathbf{S}, p) \neq app(app(\mathbf{S}, pa), qa)$$

regardless the values of pa and qa.

case 4.

$$\exists qa\, s\, [app(\mathbf{S}, p) \rightsquigarrow qa \wedge q \rightsquigarrow s \wedge r = app(qa, s)]$$

By further case analysis on $app(\mathbf{S}, p) \rightsquigarrow qa$, using `Ap_parcontractE`, we can derive

$$qa = app(\mathbf{S}, p).$$

Therefore, we have

$$\exists s\, [q \rightsquigarrow s \wedge r = app(app(\mathbf{S}, p), s)]$$

This implies the conclusion (5.13) as we can assign $p' = p$ and $q' = q = s$.

□

Isabelle's `auto` proves `S2_parcontractD` automatically, and instantaneously. However, this problem is difficult for the 35 resolution provers listed on the TSTP[1] web site [63]. Without formula renaming, the theorem `Ap_parcontractE` (5.12) will generate $1 \times 2 \times 3 \times 5 \times 3 = 90$ clauses. It was obvious that with 90 clauses, resolution would be unable to prove it. Therefore, we applied formula renaming to it, so that the theorem became

$$\begin{aligned}\forall p\, q\, r\, [&(\neg(app(p, q) \rightsquigarrow r) \vee (R_1 \vee R_2 \vee R_3 \vee R_4)) \wedge \\ &(\neg R_1 \vee (app(p, q) \in \mathbf{comb} \wedge r = app(p, q))) \wedge \\ &(\neg R_2 \vee (r \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge p = app(\mathbf{K}, r))) \wedge \\ &(\neg R_3 \vee \exists pa\, qa\, [pa \in \mathbf{comb} \wedge qa \in \mathbf{comb} \wedge q \in \mathbf{comb} \wedge \\ &r = app(app(pa, q), app(qa, q)) \wedge p = app(app(\mathbf{S}, pa), qa)]) \wedge \\ &(\neg R_4 \vee \exists qa\, s\, [p \rightsquigarrow qa \wedge q \rightsquigarrow s \wedge r = app(qa, s)])]\end{aligned} \quad (5.14)$$

where each R_i abbreviates an atomic formula $Q_i(p, q, r)$ and Q_i is a new predicate symbol. The resulting theorem (5.14) generates 14 clauses.

We gave the problem with formula renaming applied to TPTP, but none of the 35 provers could prove it.

[1]The TSTP (Thousands of Solutions from Theorem Provers) Solution Library is a library of solutions to test problems for automated theorem proving systems. It also contains solutions to TPTP problems.

Recall from the previous section that our formula renaming procedure has already transformed the original `Ap_parcontractE` into the format (5.14), whose structure gives a clear indication about how it should be used during a proof by an automatic prover. In order to help a resolution prover use `Ap_parcontractE` (5.14) correctly, we only need to force the prover to select correct literals to resolve. We can see that during a proof, a resolution prover should reduce the assumption of a goal (in this case $app(app(\mathbf{S}, p), q) \rightsquigarrow r$) to the possible cases, by resolving the assumption with $(app(p, q) \rightsquigarrow r)$ of the theorem (5.14). This means that $(app(p, q) \rightsquigarrow r)$ should be resolved first in the clause

$$\neg(app(p, q) \rightsquigarrow r) \vee (R_1 \vee R_2 \vee R_3 \vee R_4).$$

The result of this resolution is

$$R'_1 \vee R'_2 \vee R'_3 \vee R'_4$$

where each R'_i is an instantiation of R_i. Subsequently, each R'_i should be resolved with each corresponding $\neg R_i$ in each clause, generated from the conjunct $\neg R_i \vee \ldots$ in (5.14). This means that for each clause generated from

$$\neg R_i \vee \ldots$$

$\neg R_i$ should be selected for resolution.

Recall that Vampire's new version v6.03 supports explicit literal tagging so that we can indicate which literal should be selected in a clause. Based on the analysis above, we have used Vampire v6.03 to label literals generated from the theorem (5.14) as follows:

$$\{{-}{-}{-}(app(p, q) \rightsquigarrow r), {+}{+}R_1, {+}{+}R_2, {+}{+}R_3, {+}{+}R_4\}$$
$$\{{-}{-}{-}R_1, {+}{+}(app(p, q) \in \mathbf{comb})\}$$
$$\{{-}{-}{-}R_1, {+}{+}(r = app(p, q))\}$$
$$\{{-}{-}{-}R_2, {+}{+}(r \in \mathbf{comb})\}$$
$$\{{-}{-}{-}R_2, {+}{+}(q \in \mathbf{comb})\}$$
$$\{{-}{-}{-}R_2, {+}{+}(p = app(\mathbf{K}, r))\}$$
$$\{{-}{-}{-}R_3, {+}{+}(pa' \in \mathbf{comb})\}$$
$$\{{-}{-}{-}R_3, {+}{+}(qa' \in \mathbf{comb})\}$$
$$\{{-}{-}{-}R_3, {+}{+}(q \in \mathbf{comb})\}$$
$$\{{-}{-}{-}R_3, {+}{+}(r = app(app(pa', q), app(qa', q)))\}$$
$$\{{-}{-}{-}R_3, {+}{+}(p = app(app(\mathbf{S}, pa'), qa'))\}$$
$$\{{-}{-}{-}R_4, {+}{+}(p \rightsquigarrow qa'')\}$$
$$\{{-}{-}{-}R_4, {+}{+}(q \rightsquigarrow s'')\}$$
$$\{{-}{-}{-}R_4, {+}{+}(r = app(qa'', s''))\}$$

where p, q and r are universally quantified variables; pa', qa', qa'' and s'' abbreviate fresh Skolem terms, which depend on variables p, q and r.

With this notation, Vampire proves problem `S2_parcontractD`. However, since other provers do not provide this literal selection facility, they cannot prove this theorem.

This example suggests that case analysis, which is a frequently used technique in Isabelle proofs, is hard to handle by resolution provers. There are two major reasons for this. First, the case analysis theorems can generate huge numbers of clauses without formula renaming and can still generate a large number of clauses with formula renaming. Second, most resolution provers do not seem to know how to use the generated clauses to perform case analysis.

Based on the analysis above, it seems that when we need to have a resolution prover prove a goal involving case analysis, we should first reduce the number of clauses using formula renaming. Subsequently, for each renamed elimination rule

$$[\neg A \vee (S_1 \vee \ldots \vee S_n)] \wedge (\neg S_1 \vee \exists \mathbf{x}_1\, \mathbf{B}_1) \wedge \ldots \wedge (\neg S_n \vee \exists \mathbf{x}_n\, \mathbf{B}_n),$$

we should help a resolution prover (perhaps by choosing a suitable ordering) to select the following literals:

- $\neg A$
- $\neg S_i$ in each clause generated from $\neg S_i \vee \exists \mathbf{x}_i\, \mathbf{B}_i$.

5.2.3 A Top-Down Approach

Our initial experiments on formula renaming demonstrated the benefit of using it. During the experiments, we have applied the specialized version by hand. Therefore, we decided to design and then implement a more general formula renaming method that can be applied to any Isabelle theorem.

I have devised two formula renaming procedures: one works in a top-down fashion and the other one works in a bottom-up fashion. I have found that the top-down approach generates fewer clauses than the bottom-up approach does. Therefore, I have decided to adopt the top-down approach in the general formula renaming method. Like the specialized version, the top-down approach is applied to theorems in negation normal form.

Definition 11. We apply formula renaming to an input formula P in a top-down manner by processing each sub-formula Q (starting from P) as follows:

- If Q is a literal, then nothing needs to be done.
- If Q has the form $A \otimes B$, where $\otimes$ is either $\wedge$ or $\vee$, then we first examine whether it is desirable to rename Q using the method given in Definition 8, by checking the condition (5.4):
  - if the condition (5.4) is satisfied, perform formula renaming on Q and update P. Then repeat the procedure on A and then B.
  - otherwise, leave Q unchanged but repeat the procedure on A and then B.
- If Q is $\forall x\, A$ or $\exists x\, A$ then check if Q should be renamed and rename if necessary. Subsequently, repeat the procedure on A.

We have applied this technique to case analysis theorems that can generate large numbers of clauses with the standard CNF transformation.

Recall that after an elimination rule is converted to negation normal form, it becomes

$$\neg A \vee (\exists \mathbf{x}_1\, \mathbf{B}_1 \vee \ldots \vee \exists \mathbf{x}_n\, \mathbf{B}_n).$$

It can be seen that the top-down formula renaming on elimination rules may only introduce new atomic formulae to replace the cases represented by disjuncts $\exists \mathbf{x}_i\, \mathbf{B}_i$. Furthermore, for each $\exists \mathbf{x}_i\, \mathbf{B}_i$ that is not replaced, we have found that it generates either one or only a few clauses. We have also found that the last disjunct $\exists \mathbf{x}_n\, \mathbf{B}_n$ is usually not renamed because doing so will not reduce the total number of clauses generated. Consequently, for those elimination rules where each k_i is large, we are likely to obtain

$$[\neg A \vee (S_1 \vee \ldots \vee S_{n-1} \vee \exists \mathbf{x}_n\, \mathbf{B}_n)] \wedge (\neg S_1 \vee \exists \mathbf{x}_1\, \mathbf{B}_1) \wedge \ldots \wedge (\neg S_{n-1} \vee \exists \mathbf{x}_{n-1}\, \mathbf{B}_{n-1}), \quad (5.15)$$

where each S_i abbreviates a new atomic formula $R_i(y_1, \ldots, y_{l_i})$ and $y_1, \ldots, y_{l_i}$ are free variables in $\exists \mathbf{x}_i\, \mathbf{B}_i$. In addition, all free variables in (5.15) are implicitly universally quantified. Moreover, if there is some $\exists \mathbf{x}_i\, \mathbf{B}_i$ that does not need to be renamed, it is left unchanged.

As we can see, the top-down formula renaming transforms an elimination rule into a similar structure as the specialized version does and hence preserves the usage information of the theorem as well. The top-down version also generates fewer clauses than the specialized version does by not renaming some unnecessary disjuncts.

5.3 Implementing Formula Renaming

Since Isabelle/HOL is the most widely used logic among Isabelle users and it has been our focus in the previous implementation, we have decided to implement formula renaming for Isabelle/HOL. Moreover, to ensure later proof reconstruction is possible, we have implemented it inside Isabelle.

Since the top-down approach is more general than the specialized version and can generate fewer clauses, it was the one we decided to implement.

A sub-formula should only be renamed if doing so can reduce the number of clauses generated, i.e. if the condition (5.4) is satisfied. A straightforward calculation on the number of generated clauses involves a lot of computation. A more efficient method — calculation of coefficients [39] — can calculate the difference between the number of clauses generated with and without formula renaming in polynomial time. Therefore, in our implementation, I used this method with some modification in order to check if any sub-formula should be renamed.

I have implemented a function `FR_td_ax`, with type `thm -> theory -> string -> thm list`. This function takes an Isabelle theorem, a theory and a string representing the name of the theorem. The return result is a list of Isabelle clauses, each with type `thm`, which are generated from the input theorem after formula renaming is applied. Function `FR_td_ax` performs the following steps:

- Convert the input theorem into negation normal form.

- Apply the top-down method to examine each sub-formula. If a sub-formula P should be renamed then a new predicate symbol R, derived from the input name string, is introduced. Find all free variables $\mathbf{x}$ in P and add the equation

$$R \equiv \lambda \mathbf{x}\, P$$

  as the definition of the new constant R into the input theory. Finally, update the top-level formula to reflect the change of the formula structure and continue with the top-down formula renaming procedure.

- After all sub-formulae are examined, for each definition $R \equiv \lambda \mathbf{x}\, P$ (of type `thm`) stored in the modified theory, prove two implications as Isabelle theorems:

$$P(\mathbf{x}) \rightarrow R(\mathbf{x}) \tag{5.16}$$
$$R(\mathbf{x}) \rightarrow P(\mathbf{x}) \tag{5.17}$$

- Replace all sub-formulae that should be renamed by the introduced atomic formulae by resolving the original input theorem (of type `thm`) with all the implications in the direction of (5.16), derived from the previous stage. The result of the resolution is another theorem `th1` of type `thm`. The implications `imps1` in the other direction (5.17) are used as the definition formulae of the new predicates (Definition 8).

- Convert both `th1` and `imps1` into clauses using the function that we have implemented before.

As an example, consider an Isabelle elimination rule `Ap_contractE` from the Isabelle/HOL theory file `Comb.thy`. After we convert it into an equivalent first-order form and then convert it to negation normal form, it becomes:

$$\begin{aligned}\forall p\, q\, r\, [\neg(p\#\#q \twoheadrightarrow r) \vee \\ p = \mathbf{K}\#\#r \vee \\ \exists x\, xa\, [p = \mathbf{S}\#\#x\#\#xa \wedge r = x\#\#q\#\#(xa\#\#q)] \vee \\ \exists x\, [r = x\#\#q \wedge p \twoheadrightarrow x] \vee \\ \exists x\, [r = p\#\#x \wedge q \twoheadrightarrow x]].\end{aligned} \tag{5.18}$$

Here, $\mathbf{K}$, $\mathbf{S}$ and $\twoheadrightarrow$ are combinators and their reduction relation defined in Isabelle/HOL. The symbol $\#\#$ is the combinator application function written as an infix operator.

As we can see this theorem can generate eight clauses using the standard CNF transformation. We apply the function `FR_td_ax` to it as:

```
val ap_contractE = FR_td_ax Ap_contractE Comb ``apcE'';
```

and obtain the result `ap_contractE` as six clauses:

$$\begin{aligned}&\neg(p\#\#q \twoheadrightarrow r) \vee p = \mathbf{K}\#\#r \vee R \vee r = sk_3\#\#q \vee r = p\#\#sk_3 \\ &\neg(p\#\#q \twoheadrightarrow r) \vee p = \mathbf{K}\#\#r \vee R \vee r = sk_3\#\#q \vee q \twoheadrightarrow sk_3 \\ &\neg(p\#\#q \twoheadrightarrow r) \vee p = \mathbf{K}\#\#r \vee R \vee p \twoheadrightarrow sk_3 \vee r = p\#\#sk_3 \\ &\neg(p\#\#q \twoheadrightarrow r) \vee p = \mathbf{K}\#\#r \vee R \vee p \twoheadrightarrow sk_3 \vee q \twoheadrightarrow sk_3 \\ &\neg R \vee p = \mathbf{S}\#\#sk_1\#\#sk_2 \\ &\neg R \vee r = sk_1\#\#q\#\#sk_2\end{aligned}$$

where R abbreviates the new atomic formula $apcE_1(p, q, r)$. sk_1, sk_2 and sk_3 are Skolem terms abbreviating the following ϵ-terms:

$$sk_1 \equiv \epsilon x.\, \exists xa\, [\neg R \vee p = \mathbf{S}\#\#x\#\#xa \wedge r = x\#\#q\#\#(xa\#\#q)]$$
$$sk_2 \equiv \epsilon x.\, \neg R \vee p = \mathbf{S}\#\#sk_1\#\#x \wedge sk_1\#\#q\#\#(x\#\#q)$$
$$sk_3 \equiv \epsilon x.\, \neg(p\#\#q \twoheadrightarrow r) \vee p = \mathbf{K}\#\#r \vee R \vee r = x\#\#q \wedge p \twoheadrightarrow x \vee r = p\#\#x \wedge q \twoheadrightarrow x$$

Finally, p, q and r are implicitly universally quantified.

As we can see, `Ap_contractE` of Isabelle/HOL generates fewer clauses then its counter part in Isabelle/ZF. This is because Isabelle/ZF is untyped and the extra clauses such as $r \in \mathbf{comb}$ are used as type constraints. Nevertheless, formula renaming can still reduce the number of clauses generated and should be applied when necessary.

5.4 Having Automatic Provers Find Relevant Clauses

Having an automatic prover distinguish relevant from irrelevant clauses during a proof search is a possible solution to the large number of axioms problem. Most resolution provers implement numerous search strategies with the aim to quickly find a proof by choosing promising clauses to resolve. These provers aim to make their strategies fair to ensure the resolution proof is complete, not only in theory but also in practice. Therefore, we decided to try this solution first, with the aim to identify the settings most suitable to handle large numbers of clauses.

We have run a series of experiments on Vampire in order to find which settings can handle a large number of axiom clauses. We have found set of support strategy is useful, but may be incomplete.

While we carried out experiments on formalizing Isabelle/ZF and Isabelle/HOL, our experimental results showed that the literal selection strategy was one of the most important factors in determining the performance of the proof search. We also realized that the literal selection function may be able to improve or worsen the performance of an automatic prover when facing a large number of axiom clauses. Although the literal selection function may not be able to help a prover to ignore irrelevant axiom clauses, it can restrict the growth of the search space. A literal selection function determines which literals and how many literals may be selected from a clause to participate in a resolution step. Each selected literal from a clause can resolve with another literal from some other clause and then generates a new resolvent clause. Consequently, the more literals are selected, the more clauses are generated. As a result, a good literal selection function that selects only "proper" literals can help to prevent unwanted clauses from being generated.

When using automatic provers to prove Isabelle goals, which literal in an axiom clause should be selected often depends on from what kind of theorem that axiom clause is generated (§3.6). Therefore, for each axiom clause, we have a rather good idea on which literal should be selected. As a result, we need to convey this information to the literal selection function of an automatic prover.

We have used the literal tagging annotation provided by Vampire v6.03 to express which literals in axiom clauses should be selected, in accordance with how the corresponding Isabelle theorems are used in Isabelle. We have also carried out experiments to

prove goals with a large set of axioms using this version of Vampire. This new version indeed proved two goals that were not proved by the old version (§5.1). The finding suggests that the literal selection function may be an important factor that affects the proof search in the presence of large numbers of clauses.

We have briefly looked at other provers, such as SPASS. However, most of these provers do not support explicit indication on which literals should be selected. Instead, they base their literal selection criteria on some term ordering. For instance, they may pick maximal literals. These literal selection heuristics may be very effective to most problems in general. However, these term orderings usually do not match our requirement on literal selection. Furthermore, they do not explore Isabelle-specific information and thus cannot solve our problem exactly.

As we have discussed in the background chapter, clause selection is a factor that directly affects the performance of the proof search when there are huge numbers of clauses. If a clause selection heuristic can successfully ignore irrelevant axiom clauses by not selecting them to resolve with other clauses, then the presence of these irrelevant clauses will not cause any harm.

As a result, we turned to another prover, E, and ran some experiments on it. E has a particular strength in clause selection strategy. In addition to the standard clause selection heuristic — the age-weight ratio — E also implements many other heuristics. The other heuristics are flexible because they allow users to define flexible clause selection criteria by inserting clauses into any number of priority queues, from which a clause is selected.

I tried to use E to prove three of the hardest problems we found from previous experiments, which are also stored in the TPTP library as COL088-2, COL089-2 and SET787-2. I tried several settings as well as an auto mode, which allows E to find a suitable clause selection heuristic based on the characteristic of the goals. However, none of these problems were proved. Finally, I gave these problems to Stephan Schulz. He kindly ran the problems on E. Nevertheless, our problems were too difficult to be proved.

5.5 Automatic Removal of Irrelevant Theorems

Formula renaming generates a minimal number of clauses from a formula. However, it cannot decide whether a given theorem is relevant to any particular proof goal. We have investigated how to remove irrelevant axiom clauses from the consideration of an automatic prover by adjusting the prover's settings. However, that could only partly solve the problem. We have briefly looked at an alternative approach, which tries to remove irrelevant theorems before submitting the relevant ones to an automatic prover.

From a set of theorems, it is virtually impossible to decide exactly which theorems are relevant to any given goal. Therefore, we should try to have a safest approach, which removes as many irrelevant theorems as possible, but never removing any relevant ones. Moreover, for a particular goal, its relevant theorems must be at least syntactically relevant, i.e. they must share some constants.

I have considered an algorithm to solve the problem. When given a set of theorems from a theory context and a goal to be proved, this algorithm constructs an inductively defined set of theorems that is a subset of all the given theorems. The base case of the set is all the theorems that contain some constants occurring in the goal and the inductive

step is that a theorem is added into the set if it shares some constant with another theorem already in the set. Therefore the resulting set of theorems contain all relevant theorems. In addition to the goal, this algorithm takes a theory context as another input because Isabelle's global classical and equality reasoning rules are stored with a background theory context.

The algorithm works as follows.

1. Retrieve all the theorems from the input theory and construct a table Tab, which is indexed by constants. The value associated with each constant is a list of theorems, in which the constant occurs.

2. Let C be the set of all constants in the goal.

3. From Tab, find L — a list of theorems that contain some constants in C.

4. Let C_L be the set of all constants that occur in L.

5. Let $C' = C \cup C_L$. If $C = C'$ then stop, else update $C := C'$ and goto 3.

We have tried this algorithm on Isabelle/HOL problems. Unfortunately, we have found that this algorithm is not strong enough: many irrelevant theorems are not filtered out. There are two reasons behind it.

First, Isabelle/HOL is typed and many operators (constant functions and predicates) are polymorphic. Therefore it is very likely that these polymorphic operators are used in multiple theorems, possibly with different type instances. Therefore, many completely irrelevant theorems are considered relevant by the algorithm above because they share the polymorphic operators with the goal or with other theorems that have been added to the retained set of theorems.

Second, this algorithm decides which theorem may be relevant, based on the syntactic structures of goals and lemmas. However, syntactic structural information alone is not sufficient to determine whether a theorem may be used: it is possible that an irrelevant theorem may seem relevant to a goal as they contain the same constants. This problem is more prominent in the presence of multiple occurrences of polymorphic operators in different theorems.

For the first problem, I have considered a modified version of the algorithm, which decides whether to add a theorem to the retained set of theorems not only by considering whether they share any constants, but also by considering whether they share constants with the same types, or types that can be unified. This modified algorithm should be able to remove many irrelevant theorems due to different types. However, we have already achieved this effect by formalizing Isabelle/HOL's type system and polymorphic types in first-order logic and sending goals with type information included to automatic provers. In the future, it may be worth implementing this modified algorithm and carrying out some experiments to examine whether it can help improve the performance of automatic provers in reality.

The findings above suggest that an effective automatic theorem removal algorithm should not depend purely on syntactic structures of theorems and goals but also some more complex information. For instance, a system verification project is usually divided into structured theory files. A proof of a goal in one theory file may be more likely

to depend on the lemmas proved in the same theory file or an immediate predecessor theory than a more ancient theory. It may be useful to exploit the structural relationship between theories when deciding which theorems are relevant to a goal. Moreover, instead of having a safe approach to remove irrelevant theorems, which guarantees no relevant ones are removed, it may be worth trying a heuristic, which removes most of irrelevant theorems but possibly some relevant ones. If a proof attempt fails then more theorems can be added in. However, due to the time constraint, we did not pursue this topic further.

5.6 Concluding Remarks

The large number of irrelevant axioms turned out to be a problem while we did experiments on formalizing Isabelle/ZF and Isabelle/HOL in first-order logic. Although currently the problem is not our research's major concern, we have made several preliminary investigations.

We have found two major factors that lead to large numbers of axiom clauses being sent to an automatic prover: irrelevant Isabelle theorems and the generation of large numbers of clauses during CNF transformation by some elimination rules. We have made several attempts to tackle the two problems.

We have successfully used formula renaming to reduce the number of clauses generated from Isabelle formulae during CNF transformation. We have designed a specialized version that targets directly those elimination rules that are used for case analysis. We have also designed and implemented a more general version that applies formula renaming to any input theorem in a top-down manner. Our experimental results showed that Vampire indeed benefited from the reduction of generated clauses and formula renaming is a practical method.

However, our experimental results on some particularly hard problems involving case analysis also showed that resolution provers were not good at handling case analysis, which is common to Isabelle proofs. Therefore, for those hard problems, formula renaming alone is sometimes not sufficient. The performance of resolution provers can be improved if we can give them more explicit information about how those elimination rules should be used.

How to remove irrelevant theorems or axiom clauses is the hardest problem to solve, as we have expected. We have tried to solve the problem from both automatic provers' side and interactive prover's side.

We have first studied various settings of a resolution prover, which can influence the proof performance in the presence of a large set of axioms. We also carried out experiments on Vampire and E. We have found set of support strategy is useful for this problem. In addition, the literal selection function can help to exploit Isabelle-specific information. However, most of the provers are facing the general public and do not provide facilities that match our requirement. The clause selection function directly affects which clauses are selected for resolution. However, our experimental results showed that they were not particularly effective for our problem. This might be because that most provers aim to have a fair clause selection strategy, which means all clauses should be selected eventually. Perhaps, this is not really what we want: we want irrelevant axiom clauses never to be selected, or as infrequently as possible. On the other hand, if a selection is guaranteed to be fair then an Isabelle goal should definitely be proved if it has one, but possibly taking

a long time. This may be useful if we run proofs in a batch mode.

In addition, we have tried to remove irrelevant theorems before sending them to automatic provers by designing an algorithm that automatically filters out the irrelevant theorems, based on the syntactic structures of goals and theorems. This algorithm guarantees that none of the relevant theorems will be discarded, but is not strong enough so that many irrelevant theorems are still considered relevant. A more effective method, possibly a heuristic, may require more detailed knowledge on the structure of Isabelle theory files.

The findings above seem to suggest that the problem caused by large numbers of irrelevant axiom clauses or theorems may not be completely dealt with if we work on one side of the integration only. Currently, most of Isabelle specific information such as how theorems are used and the structural relationship between theories are not exploited by automatic provers. If an automatic prover can be supplied with more knowledge about Isabelle, then perhaps the problem can be handled more effectively. Although we have only partly resolved the problem, the findings should be useful for later research in this area.

Chapter 6

Higher-Order Reasoning

Our formalization of Isabelle/HOL leaves out some of its higher-order constructs. In this chapter, we discuss how to extend the existing integration by having automatic provers prove Isabelle/HOL's higher-order goals automatically.

6.1 Proving Higher-Order Logic Problems

Our integration between Isabelle/HOL and first-order resolution provers still leaves out some aspects of higher-order logic. They include constructs like λ-terms, predicate variables and function variables.

Currently, higher-order logic goals and lemmas are not sent to our target automatic prover. In order to have these goals proved automatically, there are several approaches that we could take.

One possible solution that we have considered is to formalize the higher-order constructs in first-order logic. Function and predicate variables can be dealt with relatively easily if we define a first-order function *app* to represent explicit function application and then represent functions and predicates as zero-place first-order terms. Therefore, quantifications over functions and predicates are simply quantifications over term variables.

The only remaining task is to represent λ-terms in first-order logic so that the bound variables can be eliminated. This can be solved by using combinators to represent them. In a basic form of translation, we will only need primitive combinator constants **K** and **S**, where **K** and **S** satisfy the following combinator reduction relations

$$\mathbf{K}\,P\,Q \twoheadrightarrow P \tag{6.1}$$
$$\mathbf{S}\,P\,Q\,R \twoheadrightarrow P\,R\,(Q\,R) \tag{6.2}$$

There is a standard mapping from a λ-term to a combinator expression. The mapping function M between λ-terms and combinator expressions is defined as follows.

$$M[\lambda x\,x] \equiv \mathbf{S}\ \mathbf{K}\ \mathbf{K} \tag{6.3}$$
$$M[\lambda x\,P] \equiv \mathbf{K}\,P \tag{6.4}$$
$$M[\lambda x\,(P\,Q)] \equiv \mathbf{S}\,(M[P])\,(M[Q]) \tag{6.5}$$

where the proviso of equation (6.4) is that x must not be free in P.

As it shows, the result of applying this transformation on a higher-order formula will be a first-order formula, which consists of the combinator constants, the function *app*, first-order constants and free variables. Furthermore, in addition to the existing equality between terms, we can also map combinator reductions to equalities. This is a possible approach to formalize Isabelle/HOL's higher-order constructs in first-order logic.

Joe Hurd has implemented a conversion from λ-terms to combinators when he integrated an automatic prover Metis with HOL [27]. However, he has observed that it is too weak to use combinators to prove higher-order problems, mainly due to two problems.

First, the **S** reduction (6.2) duplicates the expression R, which may be very large. Therefore, if a sequence of β-conversions in the original λ-representations has to be carried out via a sequence of **S** reductions, then a blow up on the size of combinator terms will result. Although one can introduce combinators **B** and **C** to handle special cases of **S** reduction, without **S**, the system will not be Turing complete.

Second, it is difficult to simulate the direction of combinator reductions using the built-in equality literals of most automatic provers. For instance, in order to perform the **S** reduction, an automatic prover should replace the left hand side of (6.2) by the corresponding right hand side via equality rewriting. Recall that for a resolution prover, the term ordering determines ordered resolution, superposition and simplification. Most automatic provers implement reduction ordering, such as KBO, so that bigger terms can be replaced by smaller terms. It is not difficult to see that the right hand side of (6.2) is actually bigger than the left hand side. Consequently, combinator reduction may not take place in the correct direction.

Hurd has found that the method above would work if combinators are only used to statically represent λ-terms without having to perform any λ-reductions via combinator reductions. Alternatively, the proofs that only require the **K** reductions can also be found. It is best to ignore the **S** reductions in most of the cases. Nevertheless, many higher-order problems still require the use of combinator **S**.

Because of the problems above, we have decided to explore another solution. We have considered whether it would be possible to use an automatic prover based on higher-order logic to prove those higher-order goals of Isabelle/HOL. In order to assess this possibility, we have used TPS [3] in its automatic proof mode. TPS has been used as an external automatic proof tool for the ΩMEGA system. We were interested in discovering if TPS could also be used to prove Isabelle's goals. Another reason for choosing TPS is that its automatic proof mode is relatively easy to use: we can simply submit a goal to TPS and wait for a proof result. Moreover, TPS is probably the best higher-order prover that is completely automatic.

We have investigated problems such as in what format we should send Isabelle proof goals and already proved lemmas to TPS and also whether Isabelle/HOL's types can be translated directly to TPS's types.

We have also carried out a set of preliminary experiments in order to examine whether it is feasible to use TPS to prove Isabelle's goals automatically, using our method of translating Isabelle/HOL's formulae to TPS format.

6.2 The TPS System

TPS [3] is a theorem proving system for classical type theory, which is higher-order logic in its original form. It is based on the typed λ-calculus and it supports automatic proofs, interactive proofs and a mixture of both. Since the objective of our integration is to use TPS as an automatic proof tool to assist Isabelle's interactive proofs, the automatic proof mode is obviously our choice.

TPS accepts an input formula in any arbitrary format. However, unlike resolution provers whose input can be easily split into clauses, TPS requires all goal information to be expressed as one single formula. Subsequently, the goal may be proved by a natural deduction calculus, expansion proofs or proof steps that interleave the two proof styles. TPS provides facilities to convert one proof to another.

The natural deduction calculus is more human-readable and is normally used in interactive proofs. Users can carry out proofs in this style by applying inference rules. These may be standard built-in rules, such as conjunction introduction, modus ponens for first-order logic and λ-conversion rules for higher-order logic. In addition, users can define their own inference rules by first proving the rules and then adding them to some user-defined library.

In order to assist interactive proof search, TPS also provides some support for automatic application of inference rules. This is achieved by *tactics* and *tacticals*. A tactic can be used to reduce a current goal to several subgoals in a backward proof. It can also be used to derive new assumptions from existing ones in a forward proof. Moreover, a tactical can be used to build a tactic by combining several other tactics. However, tactics have mainly been designed to facilitate interactive proofs but are not required to prove a goal. Therefore, a tactic will stop applying inference rules to a goal when a decision point is reached, where there may be more than one applicable inference rule. At this point, TPS will ask for the user's decision. After the decision is made, tactics can be called again to continue with the rest of a proof, until a next decision point is reached.

In contrast to the natural deduction proofs, the expansion proofs are less human-readable, but are suitable for TPS to conduct the automatic proof search. A key technique used in TPS's expansion proofs is the *mating search procedure* [2], which is essentially proving by refutation. Using mating search, TPS tries to find an expansion proof of a goal automatically. Usually, after a goal is proved, its expansion proof is translated to a natural deduction proof so that the proof can be more readable.

As with many other automatic provers, the behaviour of TPS proof search is controlled by various flags, which can be set by users. Examples of these flags are `MAX-MATES`, which determines the maximum number of times a literal can occur in a mating, `DEFAULT-MS`, which determines which mating search strategy should be used and `REWRITE-EQUALITIES`, which defines how equality rewriting should take place. Since there are more than 300 flags available, users can group a collection of flag settings into a *mode*. In addition, TPS uses search lists. A search list is a list of search items, where each item contains a flag, the default value of the flag and the valid range of values, which can be assigned to the flag.

TPS provides several *top levels*. Each top level represents an environment, in which certain proof development actions can take place. Therefore, each top level has its own set of commands. For instance, a user usually works in the main top level to carry out

natural deduction proofs. However, if he wishes to carry out mating search by hand, then he can enter `MATE` top level. Another useful top level is `REVIEW`, in which a user can set values to various flags or review the existing flags' settings.

6.2.1 TPS Type System

TPS is typed and each atomic type is represented by a single character string. For instance, string "O" represents the proposition type, and string "OI" represents the function type $I \to O$. For complex compound types, type abbreviation is allowed. For example, string "S"abbreviates "$(O(OI))$", which stands for natural numbers. Users can explicitly indicate types of terms. For example, $f(OI)$ represents a function f whose type is $I \to O$[1]. Alternatively, TPS can automatically infer types with a type inference mechanism based on the algorithm proposed by Milner [36].

TPS also supports some polymorphism, which is used for polymorphic abbreviations or library constants. For example, the $\leqslant$ operator is a library constant and is polymorphic. Moreover, a subset relation `SUBSET` is defined to abbreviate a big λ-term. Therefore, if `SUBSET` has multiple occurrences in a term, then each occurrence may have a different type. However, in other cases, type variables are treated as fixed and thus cannot be instantiated.

In order to use TPS as an automatic proof tool for Isabelle, we would like to directly send Isabelle formulae to TPS without having to define any Isabelle operator as a TPS constant. Therefore, the restriction of TPS polymorphism means that we cannot make use of its polymorphic type system. As a result, we will have to treat TPS's type system as if it were monomorphic.

6.2.2 TPS Automatic Proof Mode

As we have mentioned before, TPS's automatic proof search relies on mating to find an expansion proof. Mating search is similar to the resolution procedure in that they both prove a theorem by refutation using unification of literals. However, the difference between them is that resolution requires formulae to be in clause normal form, whereas mating avoids the exponential blow-up in the clause form by only converting formulae to negation normal form, with existential variables removed using Skolemization.

Subsequently the formula is displayed in a two dimensional graph, where disjunctions and conjunctions are placed horizontally and vertically respectively. Finally, mating search starts, which aims to find suitable pairs of literals occurring in the graph that are complementary to each other via unification, which lead to a contradiction of the formula. Once an acceptable mating is found, an expansion proof is returned.

Mating search can be conducted by hand, as well as by TPS automatically using its automatic proof mode. One of the simplest ways to carry out an automatic proof search is by using the command `DIY`: a user can state a goal to be proved and wait for `DIY` to find a proof. The behaviour of `DIY` is highly influenced by the value of `DEFAULT-MS`. Currently, there are eight possible settings for this flag.

[1]In order to avoid confusion between types of terms and function applications, TPS uses square brackets to represent function applications.

TPS provides another more useful command, namely `UNIFORM-SEARCH`. Essentially, this command asks TPS to find correct settings of flags for a goal so that the goal can be proved using those settings. Recall that a collection of TPS's flags' values are usually put into a mode. `UNIFORM-SEARCH` begins by taking a mode (by default, its value is `UNIFORM-SEARCH-MODE`) and a search list; with these, it starts the proof search. It first sets the flags with the values taken from the given mode. Subsequently, it tries to modify the values of the flags according to the search list, until a proof is found. Once a proof is found, the flags' settings — possibly different to the original ones listed in the input mode — are grouped into another mode. The output mode is then stored so that it can be used again for another problem with possibly similar characteristics.

6.3 Formalizing Isabelle/HOL in TPS Format

If we use TPS to assist Isabelle proofs, we must send it all the previously proved lemmas and the goals to be proved.

TPS provides many facilities for users to define their own inference rules, library constants and modules. One apparently easy approach to formalize Isabelle theorems is to translate each theorem to a TPS inference rule, so that Isabelle's forward and backward chaining rules can be simulated directly by TPS's natural deduction rules. However, at the moment, inference rules in TPS can only be used in natural deduction proofs but cannot participate in the automatic proof search. This means that if we want Isabelle theorems to be used by TPS in the automatic proof search, then we cannot translate them to TPS's inference rules.

On the other hand, even if TPS's inference rules could participate in the automatic proof search, there would be other difficulties involved in formalizing Isabelle theorems as TPS's inference rules. TPS's inference rules have to be proved first before being used. However, this is not feasible for our integration problem: an Isabelle proof goal may require the use of a local lemma that is not yet translated to a TPS's inference rule. Therefore, in order to use this local lemma as an inference rule, we will have to re-prove it first in TPS — a waste a time. Therefore, we have decided to translate each Isabelle theorem to an ordinary TPS formula, which can be used as an assumption of the goal to be proved.

Since TPS accepts its input to be one single formula, we must encode all Isabelle theorems and the goal into one big formula. Although mating search works by refutation, the procedure of negating the goal is performed by TPS, rather than a user. Therefore we can send our goal, un-negated, to TPS. This suggests that we could send a list of Isabelle theorems $L_1, \ldots, L_n$ and a goal G to TPS as

$$L_1 \wedge \ldots \wedge L_n \rightarrow G.$$

Mating search requires a negated goal to be in negation normal form. Therefore, we thought performing certain amount of negation normal form transformation to the input formula may be helpful in order to improve the performance of TPS. Therefore we decided to translate each Isabelle rule L_i to negation normal form as A_i. Moreover, TPS will have to negate our input before carrying out proof search, therefore goal G will have to be negated. As a result, we decided to leave G unchanged in the input formula.

Next, we need to consider how to deal with Isabelle/HOL's types and polymorphic operators. Since we are using TPS as monomorphically typed, we may have to translate an Isabelle polymorphic operator into several different TPS operators: one for each type instantiation. This is because that if a TPS input formula contains multiple occurrences of a polymorphic operator, each having different types, then TPS's type checker will raise a type error. For instance, the relation $\leqslant$ may have types as $nat \to nat \to bool$ and $\alpha\, set \to \alpha\, set \to bool$. When we translate two lemmas involving these two instances of the $\leqslant$ relation with these two different types, we must translate each of them into a unique TPS predicate. Since we rely on TPS's type inference algorithm to infer the terms' types, we do not have to include types in our input formula.

Finally, we translate each Isabelle/HOL higher-order construct directly to a higher-order construct written in TPS format. For instance, we translate an Isabelle function variable to a TPS function variable, an Isabelle λ-term to a TPS λ-term. Since TPS contains some already-defined library constants, we must ensure our translated formula will not contain any predicates or functions that share the names with those library constants. Therefore, we need to generate a unique name for each predicate and function occurring in an input formula.

Based on the considerations above, we have translated Isabelle goals and lemmas in the following way:

- For each Isabelle lemma L_i, we translate it into negation normal form A_i, written in a format suitable for TPS.
- Translate the Isabelle goal G to a formula G' in TPS format.
- Send the formula $A_1 \wedge \ldots \wedge A_n \to G'$ as the theorem to be proved to TPS.

This seems to be the only way to formulate Isabelle proof goals and the necessary lemmas in TPS format. However, a disadvantage of this formalization is that if there are hundreds of Isabelle lemmas, the input to TPS will be huge.

6.4 Experimental Results

I have carried out some experiments on using TPS to prove Isabelle goals. The primary objective of our experiments was to investigate whether TPS can automatically prove Isabelle goals and if so, how well it can cope with large numbers of already proved Isabelle lemmas.

We have formalized the first-order aspect of Isabelle/HOL so that Isabelle/HOL's first-order problems can be efficiently proved by first-order resolution provers. Therefore, our aim of linking Isabelle with TPS is to let TPS prove only Isabelle/HOL's higher-order goals or goals that require the use of higher-order lemmas. Consequently, during the experiments, I took existing higher-order proof goals from the Isabelle/HOL theory file `Set.thy`. I formalized the required theorems and the goals in TPS format using the method that we have described in the previous section. I then tried to reproduce the proofs using TPS.

Since it was not clear what settings were most suitable for problems like ours, I have used TPS's `UNIFORM-SEARCH`, so that several suitable proof modes may be found and can be reused later.

Goal	*1st Attempt*	*2nd Attempt*	*3rd Attempt*
1	2: 0.11 secs	6: 462 secs	11: ∞
2	2: 0.02 secs	3: 0.12 secs	5: ∞
3	2: 30.42 secs	11: 882 secs	17: ∞
4	4: 186 secs	8: ∞	
5	1: 60 secs	5: 66 secs	8: ∞
6	4: 186 secs	5: ∞	

Table 6.1: TPS Proofs Results

During the first run of the tests, I only gave TPS those necessary Isabelle lemmas for the proofs of the goals, which were around two to four lemmas. I gave TPS fourteen goals, and TPS proved eleven of them. This indicates that TPS can indeed prove Isabelle's higher-order goals automatically.

Subsequently, I tried to increase the number of irrelevant Isabelle lemmas to examine how well TPS can cope with large numbers of lemmas. Unfortunately, a few extra lemmas quickly made many goals unprovable or significantly slowed down the proof search. I looked at six goals — out of the eleven proved goals from the first run of the experiments — in more detail. For each of them, I made several attempts in order to determine what was the maximum number of lemmas TPS could handle. The results are shown in Table 6.1. On average, TPS could only prove goals within a reasonable amount of time when facing no more than six lemmas and in the best case it could prove a goal in the presence of eleven lemmas.

In each cell of the table, the number before the colon is the total number of Isabelle lemmas that I gave TPS and the number after it is the amount of time spent by TPS on mating search — measured in terms of the internal run time minus the garbage collection time — to find a proof. The "∞" indicates no proof was found for that goal with that set of lemmas within 25 minutes. In the first attempt (*1st Attempt*) for each goal, I gave only necessary lemmas to TPS. For instance, during the first attempt on the first goal, TPS spent 0.11 seconds on mating search to prove it in the presence of two (necessary) lemmas. Furthermore, for the same problem, I gave TPS eleven lemmas, nine of them irrelevant, in the third attempt. Unfortunately, TPS could not prove it within 25 minutes.

Among the goals that were proved by TPS, most of them were proved by mating search procedures `MS88`, `MS91-6` and `MS91-7`.

6.5 Concluding Remarks

In this chapter, we have discussed some possible approaches to use an automatic prover to prove higher-order problems of Isabelle/HOL. We have looked at the possibilities of using a first-order resolution automatic prover and a higher-order automatic prover. In order to use first-order provers, one approach is to formalize higher-order logic in first-order logic using combinators. Since this approach has some practical limitations, we decided to try the other method first.

We used TPS as our target higher-order prover and carried out investigation on how to formalize Isabelle/HOL's higher-order theorems and goals in TPS format. We also

conducted a series of experiments in order to assess whether it would be practical to use a higher-order automatic prover like TPS to assist Isabelle proofs. Since it was a pilot study on TPS, we carried out a relatively small amount of experiments only — if these preliminary experiments indicated TPS could be used for our integration then more experiments would be carried out.

Our experimental results show that TPS finds it difficult to handle even small numbers of lemmas. Moreover, TPS takes more time than first-order resolution provers to find a proof. Although TPS has been used to assist the proofs of the ΩMEGA system, it is not suitable to run as a background automatic prover for Isabelle to give quick proofs to Isabelle's goals.

These findings suggest that if we want to use automatic provers to prove higher-order problems of Isabelle/HOL, then we may still have to use first-order resolution provers, by finding an effective translation from higher-order formalism to first-order clause form.

Nevertheless, the experimental results also show that given enough time and having many irrelevant theorems removed, TPS should be able to find proofs for Isabelle/HOL's higher-order goals. Therefore, if we could later design an effective algorithm to filter out most of the irrelevant rules, then perhaps the performance of TPS in proving Isabelle's problems would be improved. In addition, when a batch job involving higher-order problems is to be conducted, it may be useful to run the job on TPS, with a sufficient amount of time given to TPS to run.

Chapter 7

Related Work

In the previous chapters, we have described our approach to integrating higher-order interactive proof with first-order automatic theorem proving, by linking Isabelle with several resolution-based automatic provers. My research concerns the part of the integration that directs the communication from Isabelle to automatic provers.

In this chapter, we discuss some related research work carried out by various researchers, which represents other attempts in combining interactive and automatic proofs, with the objective of improving the automation of interactive proofs. In addition to describing their work, we also compare our system with theirs.

7.1 MESON Procedure

Model elimination based theorem proving, and in particular its variant, the MESON procedure was introduced by Loveland [30]. It is a Prolog-like resolution proof procedure and is complete for first-order logic. Its usage became more widespread following Stickel's Prolog Technology Theorem Prover (PTTP) [61]. Its development is also the basis for some other theorem provers, such as SETHEO [29].

MESON proves a goal by refutation. A goal is first negated, Skolemized and converted to a set of clauses. However, unlike resolution-based theorem proving, each n-literal clause then generates n *contrapositives*, which are Prolog-like clauses. Suppose an n-literal clause is

$$L_1 \vee \ldots \vee L_n$$

then each of its contrapositives has the form

$$L_i \leftarrow \neg L_1 \wedge \ldots \wedge \neg L_{i-1} \wedge \neg L_{i+1} \wedge \ldots \wedge \neg L_n.$$

For an n-literal clause, there are n such contrapositives, one for each i.

In addition, for each negative clause, where all literals are negative, there is an extra contrapositive clause

$$\bot \leftarrow \neg L_1 \wedge \ldots \wedge \neg L_n$$

where $\bot$ is falsity. These negative clauses are usually called *goal clauses*, where each $\neg L_i$ is a goal. Given this set of contrapositives, a backward Prolog style resolution proof (with unification) is carried out, starting from goal clauses to prove $\bot$. During this process, a *reduction rule* may be applied to eliminate a current goal, if this current goal is the

complement of some ancestor goal (that occurred in a previously derived goal clause), possibly through unification. Resolution and reduction rules are repeatedly applied until a proof is found when all goals are finally eliminated.

John Harrison has described several refinements that other people have made to the Prolog implementation [24], which aimed to improve the performance of the MESON procedure. They include different search strategies such as depth-first search, breadth-first search and depth-first iterative deepening search. Among them, depth-first iterative deepening search is probably most effective. This search strategy also has some other variants, such as inference-bound search.

7.1.1 MESON in Isabelle and HOL

The MESON procedure has been implemented in Isabelle/HOL by Larry Paulson as an automatic proof tactic `meson_tac`.

A feature unique to this implementation of MESON is that `meson_tac` has been implemented inside Isabelle's logic using Isabelle's native inference rules. As a result, a proof found by `meson_tac` — which consists of a sequence of Isabelle inference rules — does not have to be translated to an Isabelle proof: it is an Isabelle proof already. This was made possible for several reasons.

First, Isabelle's inference system bears a strong resemblance with Prolog resolution: Isabelle's inference rules are represented as generalized Horn clauses. Furthermore, Isabelle's resolution mechanism, which is one of the key components of a proof search, is higher-order Prolog resolution. Therefore, the MESON procedure calls Isabelle's resolution engine. Moreover, the clause normal form transformation is implemented using recursive inference rules. Since both the goal transformation and the proof search are carried out using Isabelle's inference rules, no proof translation is necessary.

In addition, Isabelle's logical framework provides many generic search strategies as proof tacticals. Examples are depth-first, best-first and depth-first iterative deepening search. These generic search strategies can work for the MESON procedure.

Isabelle's `meson_tac` can prove many higher-order goals, and is very suitable for goals that involve many quantifiers. However, `meson_tac` does not refer to any previously proved lemma during a proof search.

After the MESON procedure was implemented in Isabelle, John Harrison also implemented it as a tactic `MESON_TAC` in the HOL system. Although it has been improved so that existing lemmas can be used to participate in an automatic proof search, users still have to manually name the relevant lemmas to `MESON_TAC`.

7.1.2 Comparison to Our Approach

In comparison to Isabelle's `meson_tac`, our integration combines two stand-alone systems into one. Therefore the translation between different logic formalisms and later proof reconstruction make our integration more complicated. However, the benefit we receive from this more complex integration is more automation. One reason is that modern resolution provers are more efficient than `meson_tac`. Resolution provers can also deal with equality much better then MESON can. Because resolution provers can handle more lemmas compared with the MESON procedure, our integration utilizes all previously

proved lemmas by sending all of them to our target automatic prover, so that the prover can use the lemmas as axioms during a proof search.

In contrast to HOL's `MESON_TAC`, our integration aims to free users from having to decide what lemmas are relevant by sending all available lemmas to an automatic prover. More importantly, we invoke automatic provers automatically, without a user's interaction.

7.2 Isabelle's Generic Tableau Prover

One of the most powerful automatic tactics of Isabelle is `blast` [46], which is a generic tableau theorem prover. It performs forward and backward chaining using any lemmas supplied by the user. Unlike Isabelle's `meson_tac`, `blast` is implemented as an independent tableau prover and is integrated with Isabelle. Therefore, translation of formulae from Isabelle formalism to tableau format is necessary. In addition, proofs found by the tableau prover are translated back to Isabelle for verification.

7.2.1 The Generic Tableau Prover

Isabelle is generic so that it can easily embed a multiplicity of object-logics. In order to fully support Isabelle proofs, the tableau prover was made generic as well.

Like other tableau provers, `blast` operates on branches. It attempts to expand branches using a given set of rules. However, since `blast` is generic, it does not implement any built-in rule: `blast` is supplied with a collection of rules and information about how to use them. The supplied rules include the four standard rules that are also implemented by other conventional tableau provers: α-rules for replacing a conjunction by two conjuncts on the same branch; β-rules for splitting a branch with a disjunction into two branches, each having one disjunct; γ-rules for instantiating universal variables and δ-rules for Skolemizing existential variables. In addition to these four rules, Isabelle's generic tableau prover allows any formalized theory to add its own rules as special tableau branch expansion rules. This significantly improves efficiency as the tableau prover can now directly work in any application-specific domain.

A standard tableau calculus usually attempts to expand branches repeatedly using the four standard rules until nothing more can be done: at this point, a list of literals are left on each branch. It then tries to close a branch: a branch is *closed* if a literal and its negation (possibly via instantiation and unification) are found on it. A proof is found if all branches are closed. In contrast, `blast` always tries to close a branch before expanding it. Moreover, unlike other standard tableau provers, such as Leantap [9], `blast` tries to close a branch as soon as a formula and its complementary are found in the same branch, even if the formula is compound. This significantly speeds up the proof search. Furthermore, as `blast` is generic, there are other situations under which a branch can be closed. This may happen when a rule representing a contradiction is found on a branch.

Backtracking may happen at several places during a proof search. For example, `blast` may be given some tableau rules that involve variable instantiations. When `blast` applies such rules, it will have to instantiate variables, even though it is not the time yet to try closing a branch for a standard tableau prover. Furthermore, an expansion rule may correspond to an Isabelle's unsafe rule, which when applied causes lost of information.

Both these cases may require backtracking. The implementation of `blast` ensures a rule can be undone if other rules are applicable.

Depth-first iterative deepening has been implemented as the search strategy of the prover. The bound concerned is the number of γ-rule applications: a γ-rule instantiates universal variables, and it can easily blow up the search space unless it is limited. In fact, the problem caused by variable instantiation of γ-rule is so problematic that the application of γ-rule is deferred to as late as possible. It is hoped that by having formulae of other types expanded first, the number of possible instantiations of variables in a γ-formula can be reduced.

7.2.2 Integrating the Generic Tableau Prover with Isabelle

Two major tasks were involved in the integration of the tableau prover and Isabelle: translating Isabelle goals and theorems to initial tableau and tableau rules respectively; and translating proofs found by `blast` to Isabelle proofs for verification.

The `blast` tableau calculus is sort of sequent calculus. Moreover, Isabelle's introduction rules and elimination rules resemble sequent calculus' right and left rules respectively. As a result, Isabelle's rules can be directly translated to tableau rules. Moreover, an Isabelle goal is negated and translated to a tableau.

In order to verify and reconstruct a proof found by `blast`, all tactics that correspond to inferences performed by `blast` are recorded, in the order they are used. An Isabelle proof is then reconstructed based on this sequence of tactics. Finally, the constructed proof steps are applied to the original Isabelle goal and the whole process finishes.

Since the implementation of `blast`, it has been one of the most frequently used Isabelle tactics.

7.2.3 Comparison to Our Approach

Isabelle's `blast` does not require users to decide which previously proved lemmas are relevant: all lemmas in the classical reasoner are taken into account when it searches for a proof. This significantly reduces user interaction compared with other tactics such as HOL's `MESON_TAC`. Our approach to integrating Isabelle and automatic provers preserves this policy. However, there are many differences between our approach and `blast`.

First, we use resolution provers to prove goals automatically. Within the realm of first-order logic, resolution is much more powerful than tableau method. Although `blast` can prove goals expressed in logics other than first-order logic, we have found that most of the Isabelle goals that are suitable for automatic proof tools to prove are in fact first-order. Therefore, using resolution provers as automatic proof tools should be able to provide faster and shorter proofs.

Second, `blast` is usually used to prove goals using classical reasoning. It does not prove goals involving equality very well. Equality reasoning is performed by Isabelle's other tactics such as `auto`. Moreover, these different tactics have to be applied separately and a user has to decide which tactic is applicable and invoke it explicitly. In contrast, resolution and paramodulation provers can effectively combine classical and equality reasoning. A goal that requires both of these reasonings can be directly sent to a resolution prover and we can then wait for a proof to be found. In addition, our integration does not ask

users to invoke background resolution provers: our target resolution provers are called automatically and this invocation is invisible to users.

7.3 HOL and Metis

Joe Hurd has integrated his automatic prover Metis with the HOL system [27]. This prover is called from HOL by invoking a tactic called `METIS_TAC`. Since its implementation, `METIS_TAC` has been used heavily.

Metis is a first-order resolution prover and is written in Standard ML. During a proof development in HOL, if a user wishes to let Metis prove a goal, then he needs to specify which lemmas are relevant to the goal and then invoke `METIS_TAC`. Subsequently, the set of lemmas and the negated goal are converted to clause normal form and then sent to Metis for a proof. When a proof is found, it is translated back to a HOL proof so that the proof can be verified.

During the conversion from higher-order logic to first-order logic, one particular problem is the translation of λ-terms. For `METIS_TAC`, the combinators **S**, **K**, **C** and **o** are used: a λ-term at or beneath the literal level is replaced by a combinator expression.

In higher-order logic, there is no difference between terms and formulae, which means different treatments are required for translating boolean and non-boolean higher-order terms. For `METIS_TAC`, non-boolean higher-order applications are mapped to first-order terms using the operator @, which represents explicit first-order function application. Boolean higher-order terms require additional lifting: after mapping a boolean higher-order term to a first-order term using @, it is lifted to a first-order formula by the predicate B. For instance, the higher-order term $m \leqslant n$ is converted to first-order formula $B(@(@(\leqslant, m), n))$. However, equality is translated differently: it is mapped to first-order equality directly.

`METIS_TAC` also translates HOL types and embeds the types in first-order clauses. First, each HOL type is mapped to a first-order term. Subsequently, the terms representing types are embedded in the first-order formulae such that each function or predicate is bound with its type. Moreover, each constant or variable is also bound with its type if necessary.

A proof found by Metis is translated back to a HOL proof for verification. In order to facilitate the proof translation, `METIS_TAC` utilizes a logical kernel. This logical kernel contains five ML functions, which correspond to the essential resolution inference rules that operate on clauses. Furthermore, the results of these functions have ML type `thm`, which represents valid theorems.

Metis uses this logical kernel to derive theorems. Every time a deduction is performed, the corresponding primitive rule and the theorems used are recorded in a proof log so that when a proof is found, the complete sequence of deduction steps can be found. In order to translate the sequence of deduction steps(expressed in the logical kernel's primitive rules) into a sequence of HOL proof steps, for each primitive rule of the kernel, a HOL inference rule is defined. The two inference rules perform the same logical inference, except that the HOL version works on HOL terms, substitutions and theorems. Moreover, the axioms and assumptions in the proof expressed in the logical kernel are translated to HOL theorems that correspond to the clauses when clause normal form procedure is applied to the user-specified lemmas and the negated goal.

7.3.1 Comparison to Our Approach

Our system is similar to the HOL-Metis integration in that we both use first-order resolution provers to assist interactive proofs. In addition, we both need to translate higher-order logic to first-order logic formalism. On the other hand, there are many differences between our system and theirs.

First, Isabelle/HOL's type system supports order-sorted polymorphism. In addition to formalizing types in first-order logic, which was also performed in the HOL-Metis integration, we have also formalized axiomatic type classes, subclass relations and type constructors' arities.

Second, unlike `METIS_TAC`, our system does not ask users to pick up relevant theorems or to carry out explicit invocations to automatic provers.

`METIS_TAC` uses combinators and explicit function applications to formalize higher-order logic constructs. Although we have not fully formalized Isabelle/HOL in first-order logic, we have investigated other alternatives to prove Isabelle/HOL's higher-order problems, which we have discussed in the previous chapter.

7.4 The ΩMEGA System

ΩMEGA [58] is an interactive prover designed mainly for proof developments in the mathematical domain. It employs the idea of knowledge-based proof planning at an appropriate level of abstraction for proof construction. Some external reasoning systems have been integrated with ΩMEGA in order to improve automation.

The ΩMEGA system is built up from several independent modules, which are connected via the mathematical software bus MATHWEB-SB so that these modules can be running at several severs over the Internet and can be accessed by many different research institutions.

ΩMEGA's inference mechanism provides various levels of abstraction. At the lowest level is an interactive prover based on a higher-order natural deduction (ND) variant of a soft-sorted version of Church's simply typed λ-calculus. Users can interactively carry out proofs at this calculus level. Alternatively, users can also construct proofs at a higher level of abstraction using tactics or methods. The system also attempts to conduct automatic proof search at an abstract level, which is called *proof planning*.

The central data structure of the ΩMEGA system is the *Proof plan Data Structure* ($\mathcal{PDS}$), which stores the current proofs and proof plans at various levels of abstraction.

Once a proof development starts, a proof construction is carried out either by direct user interaction or through the supports from several subcomponents of the system. These subcomponents include

- A proof planner MULTI, which attempts to find a proof plan automatically.
- A suggestion mechanism Ω-ANTS.
- Several external reasoning systems. They include
  - *Computer Algebra Systems* including MAPLE [14] and GAP [21]. They are mainly used to guide the proof search and to carry out some complex algebraic computation.

- *Automated Theorem Proving Systems* including Bliksem [17], EQP [34], Otter [33], Protein [7], SPASS [65], WaldMeister [25], TPS [3] and LEO [11].These ATPs are used for solving subgoals.
- *Model Generation Systems* including SATCHMA [32] and SEM [69]. Their use is to guide the proof search by finding counter examples and hence showing some subgoal is not a theorem.
- *Constraint Solvers* including $\mathcal{CoSIE}$ [35]. It helps to construct mathematical objects and to reduce the proof search by checking constraints inconsistencies.

When an external prover finds a proof of a subgoal, the result will be converted to a format suitable to be stored in $\mathcal{PDS}$ and will be inserted into $\mathcal{PDS}$. Several interface modules between ΩMEGA and these provers are used to perform proof transformation and to translate and send goals from ΩMEGA to these provers.

Very importantly, these subcomponents run in the background without any explicit user invocation. Both a user and all of these subcomponents can contribute to the proof development by modifying the $\mathcal{PDS}$ until a complete proof plan is found. Since a complete proof plan is usually expressed at an abstract level, it must be expanded to the level of ND calculus which is then verified by ΩMEGA's proof checker.

7.4.1 Comparison to Our Approach

ΩMEGA is probably the system closest to our integration because we share a key idea that assistants should run as background processes. The interactive prover user should not have to notice that a certain tool may prove a certain subgoal: it should be attempted automatically.

On the other hand, ΩMEGA and Isabelle are designed for different purposes, which result in differences between their system and our integration system. ΩMEGA is designed to support working mathematicians, and it has been combined with a large number of other reasoning tools. Our aim is to support formal verification, and we are trying to achieve the best possible integration with one or two other reasoning tools. Creative mathematics and verification are different applications: the mathematician's main concern is to arrive at the right definitions, while the verifier's main concern is to cope with fixed but enormous definitions.

Another major difference between our integration and theirs is largely due to the difference between Isabelle and ΩMEGA. All formula transformation in Isabelle has to be carried out using kernel defined functions and thus is more restricted. However, ΩMEGA is not an LCF prover and reasoning at an abstract level can be performed with greater flexibility, which is only expanded to the natural deduction calculus when a proof is verified.

7.5 KIV and ${}_3T^AP$

KIV (Karlsruhe Interactive Verifier) [49] is an interactive prover designed mainly for software specification and verification. It is an LCF style tactic theorem prover and proofs are carried out by reducing goals to subgoals using tactics. Users can either interactively

tell the system how to use the available tactics or let the system choose the appropriate tactics using built-in heuristics. In addition, KIV has been integrated with an automatic prover ${}_3T^AP$ [8] so that many simple subgoals can be proved automatically.

${}_3T^AP$ is an automatic tableau-based theorem prover for many-valued first-order logic with sorts. It handles full first-order logic with any finite number of truth values. Formulae for ${}_3T^AP$ do not have to be in any normal form. Some particular connectives such as `if-then` are used to enforce ordering on branch expansion.

The integration between KIV and ${}_3T^AP$ allows ${}_3T^AP$ to be used as a tactic to prove KIV's first-order goals or subgoals. It can be invoked either by users or by KIV's heuristics. In either case, a user will require a response from ${}_3T^AP$ in a reasonable amount of time; this is ensured by setting a time limit on each run of ${}_3T^AP$.

Since KIV's logic is an extension of many-sorted first-order logic, which is supported by ${}_3T^AP$, the translation from KIV formulae to ${}_3T^AP$ inputs can be dealt with relatively easily.

When ${}_3T^AP$ finds a proof, the tableau proof is translated back into a sequent proof in KIV format, which can then be verified by KIV. Although there is no direct mapping from rules used in ${}_3T^AP$ to the ones used in KIV (due to the rules such as the cut-rule), the proof translation is not too difficult, because the tableau calculus is a graph representation of the sequent calculus. This proof translation from ${}_3T^AP$ to KIV is mainly carried out by using KIV's sequent rules to simulate ${}_3T^AP$ rules.

A major problem encountered for their integration is the huge search space faced by ${}_3T^AP$, which is partly caused by the large number of lemmas used by KIV. In addition, some problem specific information, such as how a KIV lemma should be used for a proof, is by default, not exploited by ${}_3T^AP$.

KIV has attempted to remove many irrelevant lemmas and only send those possibly relevant ones to ${}_3T^AP$. Their technique relies on the fact that formal specifications are usually well structured theories. KIV's specifications are built up from elementary first-order theories with structuring operations such as union, enrichment, parameterization, actualization and renaming. As long as these operations satisfy certain properties[1], such as hierarchy persistency, axioms can be safely reduced in the sense that if a theorem holds in the reduced axiom set, it also holds in the original axiom set.

KIV implements an algorithm based on four reduction criteria [50] in order to remove irrelevant axioms. The first criterion *minimality criterion* allows the algorithm to take on a set of specifications and the goal to be proved and then, with the knowledge about structuring relation between specifications, find a minimal specification that covers the signature of the goal. It is then safe to retain only those axioms and lemmas in the minimal specification. Among the retained axioms and lemmas, some will never be used to prove the goal as the operators do not appear in the goal or any specification outside their definition specification. The second criterion *structure criterion* allows these axioms or lemmas to be removed. The other two criteria improve the performance of the algorithm by splitting specifications and performing the algorithm recursively until the retained set of axioms or lemmas is stable.

The algorithm based on the four reduction criteria have been proved safe for modular

[1] specifications built up in this way are called *modular*.

specifications but is only a heuristic otherwise. Moreover, although some axioms or lemmas can be safely removed, it may not be wise to do so as it could take longer to find a proof. For example, a lemma may be considered redundant as it can be inferred by the retained set of lemmas. However, it may be an important intermediate lemma for the final proof goal and will have to be derived again after being removed.

KIV has also attempted to convey some problem specific information to ${}_3T^AP$, such as how rules are used in KIV. For instance, KIV tried to simulate the rewrite rules using equalities of ${}_3T^AP$, by defining an ordering between related function symbols. However, the problem was not solved completely as the ordering is partial when variables are involved.

7.5.1 Comparison to Our Approach

Both KIV and Isabelle are used for software verification, which means proof developments in the two systems involve a large number of previously proved lemmas. The large number of lemmas is a problem for both our integration and the KIV-${}_3T^AP$ integration. We approached the problem from different directions. KIV attempted to design a technique that automatically removes irrelevant lemmas. We have also tried an algorithm to automatically remove irrelevant lemmas. Whereas ours is not strong enough, theirs may be a bit too strong. In addition, we have tried other methods to reduce the number of clauses generated. We feel the problem of a large number of lemmas cannot be solved from one side of an integration system alone: a practical solution may involve both automatic and interactive provers concerned.

In addition, we have both tried to convey interactive prover-specific information to our target automatic provers. For instance, they have tried to define new connectives in ${}_3T^AP$ and to assign weights to function symbols. We have tried to solve the problem by running experiments on our target resolution provers in order to find the best settings to solve Isabelle goals.

On the other hand, this two integration approaches differ in many aspects, which is mainly due to the different interactive and automatic provers involved in the integrations. First, the difference between Isabelle's logic and first-order logic is greater than the difference between KIV's logic and ${}_3T^AP$'s logic. Therefore, we need to spend more effort in formula translation. Although we could have used a tableau prover as the automatic proof tool, we feel resolution provers are more powerful. Second, Isabelle supports a more complex type system, which must be formalized in addition to translating formulae.

7.6 Coq and Bliksem

The Coq [6] system is an LCF style interactive theorem prover. It is based on an axiom-free type theory called the Calculus of Inductive Constructions. Coq supports intuitionistic higher-order logic. It can also support classical logic by adding the axiom of excluded middle.

All Coq terms are typed. In addition, there are two basic *sorts*, namely *Set* and *Prop*. The sort *Prop* denotes a class of propositions: if a term M belongs to *Prop*, written as $M : Prop$, then M is a proposition. As with other type-theoretical systems, a term

t of type M (written as $t : M$) is a proof of M. Therefore, to check if t is a proof of M amounts to check whether t has type M. The sort *Set* represents the type of specifications. An object of type *Set* is a usual set. Since Coq is based on type theory, it offers an expressive formalism. However, this expressiveness also means that the system is difficult to automate.

In order to improve the automation of Coq, Bezem, Hendriks and Nivelle [12] integrated it with Bliksem [17], so that Bliksem can be called via a Coq tactic. Bliksem is an automatic theorem prover, which implements resolution and paramodulation. Since Bliksem's inputs have to be in clause normal form, it is necessary to convert any output from Coq to clauses. Instead of carrying out clause normal form transformation on propositions of type *Prop* directly, a two level approach was adopted in the integration.

First, an inductive set o of type *Set* is defined, where each element of o denotes a first-order proposition. Then an interpretation function $[\![\_]\!]$ is defined so that it maps each element of o to a proposition in *Prop*. The idea is that for each formula φ in *Prop*, there is an object φ' in o, such that φ and $[\![\varphi']\!]$ are convertible. Objects in set o are called *object-level propositions* whereas propositions of type *Prop* are called *meta-level propositions*. One of the major reasons behind this two-level approach is to apply powerful symbolic transformations, such as clause normal form transformation, to object-level propositions. In comparison, meta-level propositions do not lend themselves to easy syntactical manipulations.

A clausification procedure is then defined for objects in set o. This procedure consists of a sequence of functions, such as negation normal form transformation, Skolemization, and finally transformation to clause form. The clausification procedure has been proved to be sound. After an object of set o is converted to clauses by the clausification procedure, Bliksem attempts to find a proof from these input clauses.

When Bliksem finds a proof, the resolution proof is converted to a type theory proof term. This is achieved by first translating each resolution inference rule to a λ-term. Subsequently, a sequence of λ-terms are generated that correspond to all inferences involved in a resolution proof. By assembling these λ-terms in a correct order, the final λ-term π is produced that is the representation of the resolution proof in type-theoretic format. Finally, Coq verifies the validity of the resolution proof by proving that the type of the proof term π matches the original goal. In other words, to check if a proof term π is indeed a proof of a proposition M that belongs to set $Prop$, it is sufficient to verify $\pi : M$. Together with the proof that the clausification procedure is sound, the proof verification completes.

7.6.1 Comparison to Our Approach

Although both Coq-Bliksem integration and ours use resolution provers as automatic proof tools and both Coq and Isabelle are LCF provers, there are many differences between our work. We list three major differences below.

First, we apply clause normal form transformation directly to Isabelle formulae. In comparison, Coq does not transform their propositions directly, but transform terms that belong to an inductively defined set.

Second, the methods used for formula transformation are different. We carry out clause normal form transformation inside Isabelle logic by applying kernel-defined infer-

ence rules to an Isabelle formula. Therefore, the result of this transformation is guaranteed a theorem. Coq defines a function that performs clause normal form transformation and then generates a proof term to show the transformation is sound.

Third, like many of the other systems that we have mentioned, Coq-Bliksem integration requires a user to apply a tactic to invoke Bliksem and also requires the user to name lemmas that should be used to prove the goal. In comparison, our system lets this process happen automatically.

7.7 Concluding Remarks

In this chapter, we have discussed seven other attempts on linking interactive provers with various automatic provers. Each of these integrations has its own strength.

Compared with their work, we feel our contribution has made certain improvements, which we summarize below.

- We do not require a user to decide which previously proved lemmas are relevant to a proof goal. We have tried several methods to reduce the effect of irrelevant theorems with some successful results.
- We do not ask a user to invoke any target automatic prover: the whole procedure happens automatically.
- We translate Isabelle formulae inside Isabelle logic, which ensures the correctness of the translation.
- We have been using external resolution provers, which are by far the most powerful provers, especially when proving problems in first-order logic. Although resolution provers are significantly different to Isabelle and thus the translation between different logical formalisms is more technically involved, the pay-off we receive is greater automation.

Chapter 8

Conclusion

In the previous chapters, we have discussed our approach to the integration between Isabelle and resolution provers and other related work in this field. In this chapter, we summarize the work we have achieved and point out future research directions.

8.1 Summary

Our research concerns the integration of higher-order interactive theorem proving with first-order automatic proofs. The aim of the research is to investigate how to effectively use automatic provers to prove interactive provers' goals so that the overall automation of interactive provers can be improved. We carried out this research by investigating how to link Isabelle with several resolution provers, including Vampire, SPASS and E.

We started by identifying the major questions that needed to be answered and the problems that needed to be solved for an integration between two completely different systems — Isabelle and resolution provers — to be effective. All the subsequent research was conducted in order to solve all the identified problems.

One of the most important tasks for our integration is to formalize Isabelle's object-logics in first-order logic. Therefore, we carried out this research first. We have formalized first-order aspects and some non-first-order constructs of Isabelle/ZF and Isabelle/HOL including Isabelle/HOL's order-sorted polymorphic type system in first-order clauses. We have fine-tuned our translation so that it is not only correct but also helps resolution provers improve the proof search performance. We also carried out a series of experiments on Vampire and some on SPASS and let them prove Isabelle goals using our translation method. The results show our translation is practical and resolution can indeed support Isabelle proofs. From our experimental results, we have also found those settings of Vampire, which are most essential for proofs of Isabelle problems. They have been recorded so that they can be useful for future proof tasks. We have also investigated how to make a tighter integration between Isabelle and resolution provers, so that more Isabelle-specific information can be conveyed to resolution provers — this can further improve the proof performance. We have achieved this by using a new version of Vampire (v6.03). However, through experiments, we have also found a major obstacle: large numbers of axiom clauses are difficult for most resolution provers to handle.

Having formalized Isabelle's logics in first-order logic, we implemented a program that automatically translated Isabelle/HOL's formulae into clause normal form inside Isabelle

logic. Translating formulae inside Isabelle's logic not only ensures translation is correct but also makes later proof reconstruction possible.

Finally, we have implemented a program that runs between Isabelle and any background automatic prover. When an automatic prover should be called to prove some Isabelle goals, this program automatically extracts all goals and available lemmas without a user's interaction. After getting these items, our program translates them to first-order clauses, using the automatic clause normal form translation we have implemented, and writes the clauses to designated files, which can be read by an automatic prover running in the background. This whole procedure is invisible from users' point of view.

The work above completes our research on integrating Isabelle and a resolution prover. In addition to this, we have also conducted some additional research in two areas, which aims to find ways to improve the performance of using automatic provers to assist Isabelle's proofs.

We first investigated the problem caused by large numbers of axiom clauses. Our investigation showed that two major factors were responsible for the generation of large numbers of clauses: some elimination rules can generate a vast number of clauses using the standard clause normal form transformation, and there are many irrelevant theorems being sent to an automatic prover as axiom clauses. We have successfully used formula renaming to solve the first problem. However, the problem caused by the large number of irrelevant theorems is harder. We have approached this problem from two directions: both the automatic provers' side and Isabelle's side. We have found some settings of a resolution prover that are useful to ignore the presence of irrelevant axioms. We have also found some settings that, in theory, should help solve the problem. For these settings, we have made some suggestions on possible improvements an automatic prover can make. In addition, we briefly tried an algorithm that could help remove as many irrelevant lemmas as possible. Although the research was only preliminary, and so complete solution has not been found, we feel the findings could give us some hints into future research direction.

Moreover, we have tried to use a higher-order automatic prover, TPS, as an alternative to first-order resolution provers, to prove Isabelle/HOL's higher-order goals. However, we have found the performance of higher-order provers is much worse than first-order resolution provers. Therefore, we may still have to use first-order automatic provers to prove Isabelle/HOL's higher-order goals. This would require us to formalize Isabelle/HOL's higher-order constructs in first-order logic.

Since Isabelle is a representative interactive prover and many other resolution-based provers employ similar techniques as Vampire and SPASS do, the knowledge and results gained from our research can be applied to the integration of other interactive and automatic provers.

For those people who feel formal verification using interactive provers is too costly due to the amount of human experts' skills and interaction required, we feel our research should be able to ease their verification task. We hope our research will boost the application of formal verification in both research and industry. We also hope our findings may be of interest to people who are also keen in developing theorem provers and give some inspiration to their future research.

We feel our research can show the developers of automatic provers that there is a promising field in which automatic provers can be applied. We hope our experimental results on their performance when proving verification problems can be useful to them.

8.2 Future Work

Our part of the integration is largely complete. However, several problems emerged during our research and we have conducted preliminary investigations on them. Further research on them based on our findings will be beneficial. In addition, other extension tasks could also be added to our current integration. We now list them blow.

- Using automatic provers to prove Isabelle/HOL's higher-order problems. As we have discussed in §6.5, Isabelle/HOL's higher-order goals may have to be proved by first-order provers, by formalizing higher-order constructs in first-order logic. The new translation mechanism may be applied to higher-order goals and lemmas only, while the method we have designed in chapter 3 can still be applied to translate Isabelle/HOL's first-order formulae if the goal concerned is purely first-order. Since most of Isabelle's goals are first-order and can be proved by first-order theorems, the new formalization used for higher-order constructs can be called only when necessary.

- Solve the problem caused by the large number of irrelevant Isabelle theorems. Our findings (§5.5) suggest it may be useful to exploit more structural relationship between Isabelle theories in order to determine which already proved lemmas may be more relevant to a proof goal, and possibly pass on this information to our target automatic provers. In addition, some settings of resolution provers seem to be more relevant to the solution of the problem. A successful solution may require the co-operation between Isabelle and our background automatic provers.

  Several other attempts at this problem have been made by various people. Apart from the attempt carried out by the KIV-$_3T^AP$ integration (§7.5), most of the other methods are used by automatic provers to remove irrelevant data. In addition, the approaches depend on the proof calculi and search strategies employed by the provers. We list a few examples below.

  - Marc Fuchs and Dirk Fuchs designed a method [20] that uses *abstractions* to delete irrelevant clauses for the *connection tableau calculus* and *iterative deepening search* methods. Their technique has been evaluated by the prover SETHEO [29].
  - Robert Veroff has developed a *hints strategy* [64], which is based on subsumption, to determine the values of generated clauses so that appropriate clauses can be selected to participate in the proof search. Their method targets on the resolution procedure and has been implemented in Otter [33].
  - Stephan Schulz and Felix Brandt have designed a *learning inference control heuristic* for an equational theorem prover [56], which guides a proof search for a goal based on the previously proved problems that have similar characteristics with the current goal.
  - David Plaisted and Adnan Yahya designed a method [48] that calculated a set of relevant clauses for resolution procedure. Their method utilizes existing set of support and the relevance of each clause is parameterized by a distance function that calculates the number of intermediate clauses necessary to connect the clause to support set.

- Carry out an implementation that formalizes Isabelle/ZF in first-order logic, based on our experiments on Isabelle/ZF. Although Isabelle/HOL is the most widely used logic by Isabelle users, it is worth translating other logics into first-order logic so that their goals can be proved automatically. Moreover, the task should be simpler than that for Isabelle/HOL.

- Implement automatic tagging of literals with Vampire's special syntax `+++` and `---`. Our experiments show the use of this tagging facility allows us to convey Isabelle-specific information to Vampire and proof performance has been improved. Currently, our automatic clause normal form transformation does not automatically label literals with this tagging; it should be useful to include automatic tagging as part of the translation procedure on Isabelle theorems. However, apart from Vampire, no other prover supports this syntax or equivalent facility at the moment. Therefore it would be helpful if automatic provers' developers can include this as an option.

Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integrating automated and interactive theorem proving. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction— A Basis for Applications*, volume II. Systems and Implementation Techniques, pages 97–116. Kluwer Academic Publishers, 1998.

[2] Peter B. Andrews. Theorem proving via general matings. *Journal of the ACM*, 28:193–214, 1981.

[3] Peter B. Andrews, Matthew Bishop, and Chad E. Brown. System description: TPS: A theorem proving system for type theory. In David A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 164–169. Springer, 2000.

[4] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge, March 2005. Submitted for publication.

[5] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.

[6] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gérard Huet, César A. Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual: Version 8.0. Technical report, INRIA (Institut National de Recherche en Informatique et en Automatique), France, 2004.

[7] P. Baumgartner and U. Furbach. Protein: A prover with a theory extension interface. In A. Bundy, editor, *Automated Deduction-CADE-12*, pages 769–773. Springer, Berlin, Heidelberg, 1994.

[8] Bernhard Beckert, Reiner Hähnle, Peter Oel, and Martin Sulzmann. The tableau-based theorem prover $_3$*TAP*, version 4.0. In *Proceedings, 13th International Conference on Automated Deduction (CADE), New Brunswick, NJ, USA*, LNCS 1104, pages 303–307. Springer, 1996.

[9] Bernhard Beckert and Joachim Posegga. LeanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[10] Giampaolo Bella, Fabio Massacci, Lawrence C. Paulson, and Piero Tramontano. Formal verification of cardholder registration in SET. In F. Cuppens, Y. Deswarte, D. Gollman, and M. Waidner, editors, *Computer Security — ESORICS 2000*, LNCS 1895, pages 159–174. Springer, 2000.

[11] Christoph Benzmüller and Michael Kohlhase. System description: LEO - a higher-order theorem prover. In Claude Kirchner and Hélène Kirchner, editors, *CADE*, volume 1421 of *Lecture Notes in Computer Science*, pages 139–144. Springer, 1998.

[12] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. In David A. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, volume 1831 of *Lecture Notes in Computer Science*, pages 148–163, Pittsburgh, PA, USA, June 2000. Springer.

[13] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[14] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton Leong, Michael B. Monagan, and Stephen M. Watt. *First Leaves: A Tutorial Introduction to Maple V*. 1992. Also available in Japanese, ISBN 4-431-70651-8.

[15] Avra Cohn. Correctness properties of the viper block model: the second level. In Graham Birtwistle and P. A. Subrahmanyam, editors, *Current trends in hardware verification and automated theorem proving*. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[16] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice, 1986.

[17] H. de Nivelle. Bliksem 1.10 user manual. Technical report.

[18] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[19] Anthony Fox. Formal specification and verification of ARM6. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40. Springer-Verlag, 2003.

[20] M. Fuchs and D. Fuchs. Abstraction-Based Relvancy Testing for Model Elimination. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 344–358. Springer-Verlag, 1999.

[21] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4*, 2004. (`http://www.gap-system.org`).

[22] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[23] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[24] John Harrison. Optimizing proof search in model elimination. In Michael A. McRobbie and John K. Slaney, editors, *13th International Conference on Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 313–327, New Brunswick, NJ, USA, July 1996. Springer.

[25] Th. Hillenbrand, A. Jaeger, and B. Löchner. System description: WALDMEISTER – improvements in performance and ease of use. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *LNAI*, pages 232–236. Springer-Verlag, 1999.

[26] G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The* Coq *Proof Assistant - A tutorial - Version 8.0*, April 2004.

[27] Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Copenhagen, Denmark, July 2002. Springer.

[28] Gerwin Klein. *Verified Java bytecode verification*. PhD thesis, Technische Universität München, 2003.

[29] Reinhold Letz, Johann Schumann, Stefan Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *J. Autom. Reasoning*, 8(2):183–212, 1992.

[30] Donald W. Loveland. Mechanical theorem proving by model elimination. *Journal of the ACM*, 15(2):236–251, April 1968.

[31] Donald MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust.* MIT Press, Cambridge, Mass., 2001.

[32] R. Manthey and F. Bry. Satchmo: A theorem prover implemented in prolog. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 415–434. Springer-Verlag, 1988.

[33] W. McCune. OTTER 3.3 Reference Manual. Technical Report ANL/MCS-TM-263, Argonne National Laboratory, Argonne, IL, 2003.

[34] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[35] Erica Melis, Jürgen Zimmer, and Tobias Müller. Integrating constraint solving into proof planning. In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems – Third International Workshop, FroCos 2000*, volume 1794, pages 32–46, 2000.

[36] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[37] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

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

[39] Andreas Nonnengart and Christoph Weidenbach. Computing Small Clause Normal Forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 6, pages 335 – 367. Elsevier, Amsterdam, Netherlands, 2001.

[40] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification: 8th International Conference, CAV '96*, LNCS 1102, pages 411–414. Springer, 1996.

[41] Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[42] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[43] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, Cambridge University Computer Laboratory, 1993.

[44] Lawrence C. Paulson. Generic automatic proof tools. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 3. MIT Press, 1997.

[45] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[46] Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–87, 1999.

[47] Lawrence C. Paulson. Isabelle's logics: FOL and ZF. Technical report, Cambridge University Computer Laboratory, 1999.

[48] D.A. Plaisted and A. Yahya. A Relevance Restriction Strategy for Automated Deduction. *Artificial Intelligence*, 144(1-2):59–93, 2003.

[49] W. Reif. The KIV Approach to Software Verification. In M. Broy and S. Jähnichen, editors, KORSO*: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*. Springer, 1995.

[50] Wolfgang Reif and Gerhard Schellhorn. Theorem proving in large theories. In Maria Paola Bonacina and Ulrich Furbach, editors, *Int. Workshop on First-Order Theorem Proving (FTP'97)*, RISC-Linz Report Series No. 97-50, pages 119–124. Johannes Kepler Universität, Linz (Austria), 1997.

[51] A. Riazanov and A. Voronkov. Efficient checking of term ordering constraints. Preprint CSPP-21, Department of Computer Science, University of Manchester, February 2003.

[52] Alexander Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning — First International Joint Conference, IJCAR 2001*, LNAI 2083, pages 376–380. Springer, 2001.

[53] John Rushby and Friedrich von Henke. Formal verification of the interactive convergence clock synchronization algorithm. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, Feb 1989. Revised August 1991.

[54] S. Schulz. System Abstract: E 0.61. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, number 2083 in LNAI, pages 370–375. Springer, 2001.

[55] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[56] S. Schulz and F. Brandt. Using Term Space Maps to Capture Search Control Knowledge in Equational Theorem Proving. In Russell I. and A. Kumar, editors, *Proceedings of the 12th Florida Artificial Intelligence Research Symposium*, pages 244–248. AAAI Press, 1999.

[57] Sharangpani and Barton. Statistical analysis of floating point flaw in the pentium (tm) processor. Technical report, Intel Corporation, November 1994.

[58] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, Immanuel Normann, and Martin Pollet. Proof development in OMEGA: The irrationality of square root of 2. In Fairouz Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, Kluwer Applied Logic series. Kluwer Academic Publishers, 2003. ISBN 1-4020-1656-5.

[59] Michael Norrish Peter Sewell Michael Smith Keith Wansbrough Steve Bishop, Matthew Fairbairn. TCP, UDP, and sockets: rigorous and experimentally-validated behavioural specification. volume 1: Overview. Technical Report 624, March 2005.

[60] Michael Norrish Peter Sewell Michael Smith Keith Wansbrough Steve Bishop, Matthew Fairbairn. TCP, UDP, and sockets: rigorous and experimentally-validated behavioural specification. volume 2: The specification. Technical Report 625, March 2005.

[61] Mark E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.

[62] Geoff Sutcliffe and Christian Suttner. The TPTP problem library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, October 1998.

[63] Geoff Sutcliffe and Christian Suttner. TSTP problems. On the Internet at `http://www.cs.miami.edu/~tptp/cgi-bin/DVTPTP2WWW/view_file.pl?Category=Solutions&Domain=COL`, 2004.

[64] R. Veroff. Using Hints to Increase the Effectiveness of an Automated Reasoning Program: Case Studies. *Journal of Automated Reasoning*, 16(3):223–239, 1996.

[65] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[66] M. Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: TPHOLs '97*, LNCS 1275, pages 307–322. Springer, 1997.

[67] Markus M. Wenzel. *The Isabelle/Isar Reference Manual*.

[68] Lawrence Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The concept of demodulation in theorem proving. *J. ACM*, 14(4):698–709, 1967.

[69] Jian Zhang and Hantao Zhang. SEM: a system for enumerating models. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 298–303, Montréal, Québec, Canada, August 20–25 1995.