Technical Report

UCAM-CL-TR-867
ISSN 1476-2986

Number 867

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Higher-order proof translation

Nikolai Sultana

April 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 2014 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity 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

The case for interfacing logic tools together has been made countless times in the literature, but it is still an important research question. There are various logics and respective tools for carrying out formal developments, but practitioners still lament the difficulty of reliably exchanging mathematical data between tools.

Writing proof-translation tools is hard. The problem has both a theoretical side (to ensure that the translation is adequate) and a practical side (to ensure that the translation is feasible and usable). Moreover, the source and target proof formats might be less documented than desired (or even necessary), and this adds a dash of reverse-engineering to what should be a system integration task.

This dissertation studies proof translation for higher-order logic. We will look at the qualitative benefits of locating the translation close to the source (where the proof is generated), the target (where the proof is consumed), and in between (as an independent tool from the proof producer and consumer).

Two ideas are proposed to alleviate the difficulty of building proof translation tools. The first is a proof translation framework that is structured as a compiler. Its target is specified as an abstract machine, which captures the essential features of its implementations. This framework is designed to be performant and extensible.

Second, we study proof transformations that convert refutation proofs from a broad class of consistency-preserving calculi (such as those used by many proof-finding tools) into proofs in validity-preserving calculi (the kind used by many proof-checking tools). The basic method is very simple, and involves applying a single transformation uniformly to all of the source calculi's inferences, rather than applying ad hoc (rule specific) inference interpretations.

Acknowledgements

Working on this thesis has gained me some knowledge and a long list of people to thank for helping me become a better researcher. I thank my supervisor, Larry Paulson, for his advice and feedback over the years. I was also lucky to have the opportunity to work with Christoph Benzmüller and Jasmin Blanchette on work that informed this dissertation. Christoph kindly hosted me during a three-month visit to the Free University of Berlin. I also thank Geoff Sutcliffe, Stephan Schulz, Chad Brown, and Makarius Wenzel for helpfully and patiently replying to my many questions about systems they maintain. I thank my examiners, Mateja Jamnik and Christian Urban, for their helpful remarks, which helped improve this dissertation.

At the Computer Lab, I learned a lot from conversations with other people, and with other members of the Automated Reasoning Group in particular. I am especially grateful to Jannis Bulian, Pierre Clairambault, William Denman, Ramana Kumar, and Grant Passmore for reading this dissertation and giving me feedback. Mike Gordon also provided advice relating to various practical matters. I thank Lise Gough for her patience and advice, and Piete Brooks for his help with technological wizardry on many occasions. Surpassing even Mary Poppins, my officemates in 65058 were awesome in every possible way.

I had the good fortune to intern at Microsoft Research on different occasions. I found these internships to be superb learning experiences, during which I had the pleasure of working with Moritz Becker, Markulf Kohlweiss, Hillel Kugler, Boyan Yordanov, Youssef Hamadi, and Christoph Wintersteiger.

This dissertation would not have been possible without the financial support I received from Trinity College, in the form of a research studentship. I am also very grateful to the Computer Lab and the Cambridge Philosophical Society for travel funding; the DAAD (German Academic Exchange Service) for funding a research visit to Berlin; the organisers of the VTSA (Verification Technology, Systems and Applications) summer school; and the organisers of the Marktoberdorf summer schools for additional support.

It would be remiss of me not to acknowledge the work of those who made the software I used, particularly the software made freely available: the Isabelle community, David Matthews for developing Poly/ML, the OCaml community, the GNU project, the Linux, Python and Mozilla communities, the maintainers of TEX engines and various packages, and countless others.

Outside of research, I was prevented from having a dull life by my friends, including people I met at Trinity, the Lab, around Cambridge, and back home. I am immensely lucky to have them as friends, almost as lucky as they are to have me. I particularly thank those who helped me during one of my many moves around Cambridge, during my months of vagabondage: Alberto, Benjamin, Damian, Guilhem, Josh, Kevin, Louise, Matej, Pierre, Rebekah, Rok, and Steffen. With great fondness I remember tea at Dom and Ed's, and the friends I met there, who I dearly miss.

Most of all, my immediate and extended family have been a continuous source of support and courage. I cannot thank them enough.

Finally, I dedicate this work to my teachers, past and present, particularly those who are not employed as such.

La nikber nixtieq insir gadazz Brikkun u Xtruppat, u li l'ebda għodwa ma' tiġi moħlija.

Contents

Glossary

AF *Annotated Formula* is the basic unit in a TPTP problem or proof. See Section 2.5.1 for a definition and examples. 19–22, 96

ATP system Automated Theorem Prover, a computer program used to prove mathematical theorems with minimal human oversight. xiii, 9, 11, 13, 14, 19, 21, 23, 24, 31, 32, 37, 42–45, 55, 57–59, 61, 65, 66, 68, 70, 72, 74, 79, 117, 131–139, 141–145

CNF *Clause Normal Form*, also *Conjunction Normal Form*. A description of this normal form is a staple of most logic textbooks (Ben-Ari, 2012). In this form, formulas consist of conjunctions of disjunctions of literals. Getting free-form formulas into CNF is often part of the preprocessing phase of an ATP system. Nonnengart and Weidenbach (2001) described methods that make this process efficient. CNF is also the name of a TPTP language, in which formulas are encoded in clause normal form. 20, 59, 60, 98

FOF *First-Order Form* is a TPTP language for encoding formulas in free-form first-order logic. 20, 44, 98

FOL *First-Order Logic*. Also called elementary logic. In this language the universe of discourse consists only of individuals, and quantification is restricted to individuals. xiii, 24, 38, 42, 43, 45, 66, 67, 78, 101, 130

HOL *Higher-Order Logic*. In this language the universe of discourse is partitioned into different types of elements: individuals, formulas, functions over other types, and so on. One can quantify over any of these types. Unlike in FOL, one can also quantify over formulas. xiv, 5, 22, 27, 29, 45, 50, 66, 78, 131, 141, 176

RUE *Resolution with Unification and Equality* (Digricoli and Harrison, 1986) is a form of proof calculus that extends the resolution calculus (Robinson, 1965) to interpret equality. It seeks to address the shortcomings of paramodulation-style calculi (Nieuwenhuis and Rubio, 2001), promising shorter proofs and more efficient operation. RUE calculi predated superposition calculi (Bachmair et al., 1992), and do not rely on term orderings. xiv, 24

SMT *Satisfiability Modulo Theories*, the extension of the satisfiability problem to include atoms in theories of interest (such as integers, arrays, bit-vectors, etc) for which specialised decision procedures exist. 103, 133, 134, 136

TFF *Typed First-Order Form* is a TPTP language for encoding formulas in free-form multi-sorted first-order logic. 20

THF *Typed Higher-Order Form*, one of the constituent languages of the TPTP project, for encoding formulas in HOL. xiv, 20, 21, 44, 118, 123

THF0 *Typed Higher-Order Form level 0* is a syntactic subset of THF corresponding to Church's simply-typed λ-calculus (i.e., types consist of individuals, propositions, and functions over them) extended with arbitrary additional base types (over which λ-functions may also be defined). 128

TPTP This refers to the *Thousands of Problems for Theorem Proving* project, or one of its languages or tools. See Section 2.5 for an elaboration. The project is hosted at `http://www.cs.miami.edu/~tptp/`. xiii, xiv, 11, 19–21, 23, 32, 48, 51, 80, 91, 92, 94–96, 98, 116, 118, 122–124, 128, 130–132, 135, 141–144

Chapter 1

Introduction

The words "average" and "mean" are synonyms. What it means to be average is widely known. But what does it average to be mean? People have long delighted in and been vexed at the expressiveness of natural language. It is not known what the *average* sentiment towards expressiveness is, but some feel very *mean* about it—and would happily sacrifice humour and poetry, if only to make clearer the meaning of sentences.

It is a long-standing convention in maths, and in logic in particular, to rely on a sublanguage with clear semantics to ensure clear communication. In symbolic logic, meaning is distilled into a handful of symbols that can be combined to express statements of interest. For instance, '$A \wedge B$' is understood to mean "A and B", where A and B stand for subexpressions. The collection of allowable statements is called the *formula language*.

Other than providing the means of *expressing* statements, logicians have also sought to develop and study means of *proving* statements to be true, or *valid*. A proof is a graduated reduction of a statement into (what should be) self-evident truths. If each step of this reduction preserves truth, then the original statement must be true. A logic provides a small collection of allowable inference steps, and only those inference steps may be made when arguing a proof. The collection of allowable steps is called the *proof calculus*. For example 'conjE1(A,B)' is interpreted to mean that "if $A \wedge B$ is true, then conclude that A is true", which can be written using the following notation:

$$\frac{A \wedge B}{A}$$

As before, A and B stand for expressions in the formula language. A logic is *sound* if one can prove that the inference steps it allows can never lead to a non-truth being proved. Note that proving such a property involves proving a statement *about* that logic, rather than *in* that logic. Such proofs are carried out in a *meta-language*—a language in which we can reason about an *object-language*. The meta-language could be a natural language such as English, or perhaps another logic (Harper et al., 1993; Paulson, 1994; Pfenning and Schürmann, 1999). Famously there are limits to the formalisation and proof of statements in a logic (Gödel, 1931; Tarski, 1936).

Logics are typically defined in a way that makes the semantics of formalised statements and their proofs absolutely clear. Unlike natural language, there is no possibility of ambiguity. Better still, logics can be *mechanised* to have a computer help us find or check a proof. If the logic is sound, and the implementation is faithful, then the computer cannot infer false conclusions from valid assumptions.

Mechanisation is a significant advance. It is a kind of advance particularly valued by Whitehead (1911), who believed that "Civilisation advances by extending the number of important operations which we can perform without thinking of them." He co-authored Principia Mathematica (Whitehead and Russell, 1910), which sought to develop the thesis that mathematics can be entirely defined on a logical foundation. Across its three volumes, Principia Mathematica contains many pages of unforgivingly dense, manually constructed symbolic proofs. Already half a century later, there were significant advances in proving many of these theorems using early computers (Wang, 1960). Having automated support frees us from managing lower-level details, and allows us to direct our efforts at solving problems at a higher level of abstraction.

Having overcome the problem of the ambiguous semantics of language by using logic, and with the benefit of mechanisation, other problems come into view. One such problem is how to relate different logics, to move formulas and proofs between them. There are many logics, some of which differ only in the proof calculus—that is, they have the same formula language (say, first-order logic), but a different collection of inference rules. Each computer proof tool often implements its own version of a proof calculus: two automated provers for first-order logic might parse formulas described by the same syntax definition, but might implement very different proof calculi—for instance, one based on resolution (Robinson, 1965), and the other based on analytic tableaux (Fitting, 1996). Choosing which tool to use can be difficult. Some tools are better than others when it comes to finding certain kinds of proofs. One often cannot identify a better tool a priori. So why not try to use *all* available tools simultaneously? There is evidence to suggest that this is a good strategy (Sutcliffe and Suttner, 2001; Nieuwenhuis, 2002; Hamadi et al., 2009; Böhme and Nipkow, 2010; Barrett et al., 2013).

But how difficult is it to consume the output from different tools? More fundamentally, can different tools work together if they do not share the same proof calculus? If two tools accept the same formula language then we can feed them the same input. Since they are likely to have different proof calculi, we cannot use their proof output interchangeably or readily combine their output. This highlights the importance of *translating* proofs between calculi, for checking or reuse in a different system. Proof translation is the focus of this dissertation. In the coming chapters we will study the problem of organising the generation and consumption of proofs to facilitate their translation across calculi.

1.1 Proof translation

Proof translation involves transforming the description of a proof, encoded in one proof calculus, into a proof encoded in a different proof calculus, without changing the theorem being proved. Devising a precise translation is often tedious. It usually requires an in-depth understanding of both the source and target systems. If the proof is being translated between computer proof tools, often one needs to understand a prover's implementation, and not only the proof calculus it purports to implement. The source and target systems can be less documented than desired (or even necessary), and the behaviour of either system can deviate from what is documented or expected (particularly in research prototypes). Implementing a proof translation can feel like carrying out forensics *as well as* system integration. It can turn out to be an inductive problem, guided by examples, rather than a purely transformational process.

These qualities make it an enticing research problem. The problem is part theoretical and part practical. The theoretical component relates to ensuring that the translated proof does indeed prove the same theorem as the source proof. We need to rely on theoretical models to provide assurance about a translation's adequacy (its soundness and completeness). The practical component relates to the usability and feasibility of the implementation—both in terms of resource use during run-time, and also in terms of conceptual difficulty of realising an implementation.

The dissertation will explore, and attempt to solve, these difficulties. We start by elaborating *why* this matters.

1.1.1 Motivation

Different proof tools often have complementary strengths, yielding better tools when combined. We might want to use one system for finding a proof, and a different system for checking it. For example, the recent formalisation of Gödel's god-existence proof (Benzmüller and Woltzenlogel-Paleo, 2013) was carried out in different systems, and featured various proof-checking systems relying on various proof-finding systems.

People also sometimes want to re-use a proof in a different system to benefit from its libraries of formalised mathematics, or because they are more familiar with that system. The proof of the Kepler conjecture (Hales et al., 2010) made use of several tools, of various kinds.

The quality of a combination of tools is questionable unless we have some way of combining the *evidence* (such as proofs) from different tools. For proofs, this could be achieved by translating one proof into the calculus of another tool, or by translating the proofs of different tools into a single host calculus.

Lack of proof translation therefore impedes collaboration between proof tools (and their users), and the reuse of formalised proofs. We have already seen that the construction of proof translators is complicated by the diversity between proof tools. This makes it difficult to translate between different systems.

So far we have seen why proof translation is relevant and difficult. It is also timely: there continues to be enthusiastic research into the development and application of logic tools in various domains, including software testing (McMillan, 2011), security (Becker et al., 2012), distributed systems (von Gleissenthall and Rybalchenko, 2013) and biology (Kugler et al., 2014), as well as research in facilitating the translation and checking of proofs produced by different tools (Keller, 2013; Chihani et al., 2013).

1.1.2 Thesis

My thesis is that translating proofs between several tools can be simple, feasible and reliable. Different techniques for proof translation are described in the next chapters, that combine to yield a translation framework. I also argue that this framework could handle *hybrid* proofs. That is, proofs that contain segments formalised in different proof calculi. Such segments would usually be contributed by different proof tools.

I also explore three kinds of translators that have different proximities to the source and destination of a proof: a translator as part of the proof-producing tool, a translator as part of the proof-consuming tool, and an independent translator that converts proofs between formats.

In this dissertation I focus on translations targetting higher-order logic (HOL), as implemented in the Isabelle/HOL proof assistant (Nipkow et al., 2002), and on translating purely *machine-found* proofs (as distinguished, for instance, from *human-authored* proofs).

1.2 Contributions

- In Chapter 5 I describe a novel and extensible translation framework for importing proofs into a proof assistant. The framework is designed to lower the complexity of implementing proof translations. A translation ultimately relies on proof analysis and transformation. The framework forces a modular approach that separates the proof analysis/transformation into two phases: analysing and transforming the proof's *structure*, and analysing and transforming the individual *inferences*. This is appealing because it lets us separate two concerns:

  1. The embedding of the source calculus into that of the proof assistant can be described in terms of schemes and tactics that validate the inferences of the source proof.
  2. Individual inferences are then composed into a proof. The strategy for replaying the proof is devised by analysing and transforming the structure of the source proof. There might be inferences that require a non-local transformation (such as splitting rules, described in Section 4.2.3) and which therefore cannot be handled at the inference level. Naturally, these transformations do not alter the theorem being proved.

  This framework is specified abstractly in terms of *cut machines* (Section 5.4) in order to make it widely applicable—not coupled to specific source and target proof systems. To use this framework one would implement a cut machine and the associated compiler, as described in Section 5.4.2. Chapter 5 describes the application of this method to import Leo-II (Benzmüller et al., 2008c) proofs into Isabelle/HOL.

  I also argue that this approach facilitates the composition of proofs, and this in turn enables the translation of hybrid proofs.

- In Chapter 4 I describe two methods to map proofs between broad classes of calculi. The source calculi are consistency-preserving, while the target calculi are validity-preserving. These methods are specified abstractly to show their essential features, and they are proved sound and complete. The second method is derived from the first, and it is implemented in the framework described in the previous contribution.

  The first method works by applying a single transformation uniformly to all of the source calculi's inferences, rather than applying ad hoc (rule specific) inference interpretations. This makes the method very simple. This inference-level transformation is lifted to yield a proof-level transformation.

  To use this method one needs to implement the inference-level transformation, lifted as described in the chapter, to obtain a sound and complete translation. The chapter includes examples of mapped inferences from the calculi of Leo-II and E (Schulz, 2002b) to Isabelle/HOL. A prototypical implementation is described, mapping proofs produced by E into Isar scripts (Wenzel, 2002) that can be used in Isabelle/HOL.

- In Section 3.2 I review and extend the work of Teucke (2011). His work is a rare example of an ATP system that exports scripts that can be checked and reused directly by proof assistants. I argue that this approach is viable, but also that it is subject to the constraints of the target language. That is, to use Teucke's method, one would need to print a proof's representation into an input syntax that a proof assistant can check. This would need to make use of a custom theory that formalises the ATP system's calculus in the target system. For the checking to be fully automatic however, one might also need to implement a driver at the receiving end to compensate for the input language's lack of expressiveness, as is the case with Isar.

1.2.1 Publications

This dissertation draws from three articles that have previously been published:

- "LEO-II and Satallax on the Sledgehammer test bench" (2012), coauthored with Jasmin C. Blanchette and Lawrence C. Paulson.
- "Understanding LEO-II's Proofs" (2012), which was coauthored with Christoph Benzmüller.
- "LEO-II Version 1.5" (2013), also coauthored with Christoph Benzmüller.

1.3 Dissertation outline

The next chapter describes the background ideas on which the dissertation builds. Then we begin a journey covering the entire life time of a proof: from searching for and generating proofs (Chapter 3), to translating proofs (Chapter 4), and ending with their consumption by a different tool (Chapter 5). Each chapter builds on the previous chapter, and each describes contributions made in generating, translating, and importing proofs. The dissertation ends with a survey of related work, and some closing thoughts and ideas for further work.

Chapter 2

Background

This chapter describes the ideas on which this dissertation builds. It defines basic concepts, outlines different approaches to proof translation, and describes the frameworks and technologies that we will rely on in later chapters.

2.1 Proofs and translations

A proof constitutes evidence, and this dissertation studies ways of translating this evidence from one symbolic form into another, without changing the theorem that is being proved.

We will focus on the production, translation, and consumption of proofs by computerised tools implementing logic procedures. A proof is produced by a tool and encoded in the *source* logic. It is then the job of the *translator* tool to translate the theorem and its proof into a *target* logic. The resulting proof can then be checked or otherwise used by a tool that interprets the target logic. To coin a phrase, the proof of the proving is in the translating.

Before examining the translation of proofs further, it is worth considering the different forms that proofs and translations can take.

First, we distinguish between *syntactic* and *semantic* proofs: a syntactic proof is written down, usually as a chain of inferences; a semantic proof is a credible assertion of a statement. For example, Isar scripts (Wenzel, 2002) in Isabelle/HOL convey syntactic proofs. We will encounter examples of such scripts in this dissertation, the first of which occurs in Section 3.2.2. When an Isar script is checked by Isabelle, its lemmas and theorems will correspond to elements of a datatype in Isabelle's implementation, called *thm*. This is an abstract datatype, the membership of which constitutes a semantic proof. Isabelle is related to the LCF system (Gordon et al., 1979), from which it kept several ideas—not least the encapsulation of its core inferences in this abstract datatype. The implementation of this datatype, and that of a collection of other modules, determine whether a formula can be considered to be a theorem. This collection of code is usually called the *kernel* of the system. If this code is free from bugs, then a formula can only gain membership of the type thm if that formula is a theorem.[1] As a consequence of the trusted nature of the thm datatype, once the theorem-hood of a formula has been determined, we can forget its proofs: the formula's membership in thm is sufficient evidence as far as the system is concerned. This feature is appealing because it reduces the proof assistant's memory footprint.

[1]Usual provisos apply: this also assumes that the Standard ML (Milner et al., 1997) system on which Isabelle is run, and the underlying architecture, is free from bugs, and that the laws of physics behave as expected.

Forgetting proofs is not desirable if we want to compute over the proof itself—for instance, to extract an algorithm (Berger et al., 2002)—so it is possible to require Isabelle to keep a representation of the proofs it checks (Berghofer, 2003). In this dissertation we will only be concerned with translating proofs, not with extracting computation-related information from them.

Having looked at different kinds of proofs, we next distinguish between different kinds of operations on proofs. Some of the names of these operations are not always carefully defined in the literature, and their names are often used interchangeably. It is therefore worth defining how they will be used in this dissertation at this point.

- A proof *transformation* is a mapping between proofs. A proof might be mapped to another proof in the same logical calculus, or to a proof in a different logical calculus. Usually the theorem proved by a proof is not affected by the transformation. Perhaps the best-known proof transformation is cut elimination (Gentzen, 1969). Andrews (2005) reflects on the importance of proof transformations, and suggests that they might help us arrive at a notion of *quintessential* proof: that is, a proof that "embodies the essential features of many intertranslatable proofs of [a] theorem." He describes different forms of proof transformations, all of which are covered by the classification in this section.

- A proof *translation* is a transformation that maps a proof from one logic to a proof in a different logic. This is also called a proof *embedding*, usually with the nuance that the translation is total. Examples of such transformations abound (Andrews, 1980; Pfenning, 1987). There are two popular ways of translating proofs:

  - If a syntactic proof is in the domain of a translation into another logic, then we can *replay* the proof in the target logic by making use of the translation. This approach is de rigueur in proof theory. Several examples were given by Troelstra and Schwichtenberg (2000). This was also the method initially developed in Sledgehammer (Meng et al., 2006). Sledgehammer is a tool within Isabelle that interfaces with external theorem provers. It translates Isabelle/HOL problems into inputs to the provers, and translates proofs from the provers back into Isabelle/HOL theorems. Thus Sledgehammer brings the power of Automatic Theorem Provers, or ATP systems, to users of the Isabelle proof assistant.
  - If we lack an embedding or the means of using an embedding, then we could attempt to *re-find* the proof in the second logic using the available information (Paulson and Susanto, 2007). This approach is appealing because of its conceptual simplicity, treating the source proof calculus as a black box. The replaying-based approach was found to be brittle, so Sledgehammer was changed to use the refinding-based approach in later developments (Paulson and Blanchette, 2010).

- Proof *reconstruction* involves attempting to recover a proof for a formula, possibly in a different logic. The proof is not necessarily the proof encoded in the source logic. When a proof is represented syntactically, its representation is often imperfect. Certain details might be omitted—this is often deliberate, to make proofs shorter. For instance, minor syntactic operations—such as reordering conjoined formulas, or basic simplifications—might not be recorded in the proof. This information might need to be reconstructed to give a full proof prior to translation. Examples of reconstruc-

tion include the work by Paulson (1999) to reconstruct tableau proofs in Isabelle, and by Teucke (2011) to reconstruct Satallax proofs in ad hoc finite tableau calculi.

- Proof *compression* involves filtering certain contents of a proof (Berghofer, 2003). Usually this information can be reconstructed later.

- Two proofs can be *spliced* together. This can be seen, for example, when Leo-II returns the combined output of its proof search in collaboration with that of E (Sultana and Benzmüller, 2012).

Finally, we distinguish between different granularities of translation. A translation is *fine-grained* if it uses most of the information in the source proof to produce the target proof. Proof replay is a fine-grained translation since each inference of the proof is translated and chained together in the target proof. A *coarse-grained* translation makes less use of the source proof, and mostly relies on proof search to obtain a proof of the theorem in the target logic.

There are translations whose granularity is variable. For example, proof re-finding can be implemented at different granularities. Re-finding can be rather fine-grained if the proof structure is preserved and the inferences themselves are re-found (Zummer et al., 2004; Paulson and Susanto, 2007; Dunchev et al., 2012). We could also delete parts of the proof, and use search to recreate them in the target logic. In general, re-finding involves treating the source proof as a collection of hints to guide proof search. If we discard most of the information in the proof before re-finding, by retaining only the axioms, definitions and problem conjecture, then it becomes a coarse-grained translation. Some implementations allow users to vary the granularity by varying a parameter (Paulson and Susanto, 2007; Paulson and Blanchette, 2010). Coarse-grained approaches rely less on the source prover than fine-grained approaches. At the very least, they use the source prover to obtain some assurance about a formula's theorem-hood. They often also use the source prover for *relevance filtering*: when trying to re-find a proof, it is not necessarily limiting to discard axioms that were not used in the source prover's proof (despite being available to the source prover).

2.2 Designing translators

The task of a translator is to emulate the inferences of one calculus by using a different calculus. How should we design a translator? Should its design be influenced by that of the source prover?

Let us use a problem from the TPTP for a concrete example. The TPTP is, among other things, a collection of problems for ATP systems. In this dissertation we will frequently make reference to problems from the TPTP collection. Owing to the key role of the TPTP project in the development of ATP systems, we will look more closely at the TPTP syntax for encoding problems and proofs in Section 2.5.

As an example problem, we will use the TPTP problem identified as SET014^4, which conjectures that the union of two subsets of a set is itself a subset of that set:

$$\forall X, Y, A.\ (X \subset A)\ \wedge\ (Y \subset A)\ \longrightarrow\ (X \cup Y) \subset A \tag{2.1}$$

Figure 2.1 shows all the clauses considered by Leo-II during proof search. They form a graph: vertices are formulas, and two formulas are connected by an edge if one of the formulas was derived from the other (in an inference step). The graph is implicitly directed

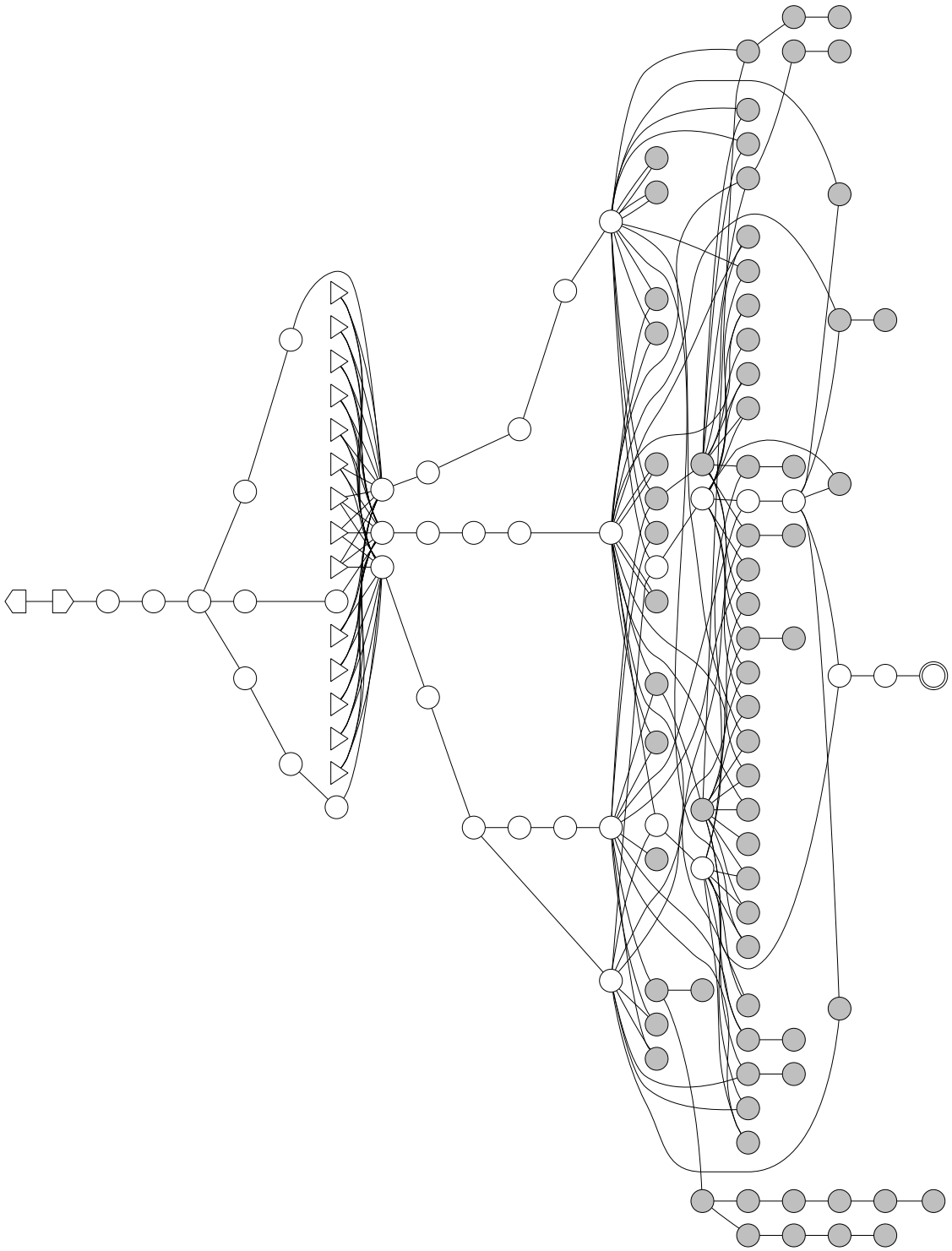


Figure 2.1: Graph showing every single clause produced by Leo-II while attempting to prove Formula 2.1, and how they are related. White nodes are the ones that ultimately form part of the proof produced by Leo-II. The grey clauses do not contribute to the proof. The graph is implicitly directed from the left side of the page to the right. This figure uses the presentation cues used by Trac et al. (2006) in their IDV system. The left-pointing pentagon is the conjecture, right-pointing pentagon is the negated conjecture, inverted triangles are definitions, and circular nodes are derived clauses. The double-circled node is the final clause (the derivation of falsity, since Leo-II works by refutation).

from the left side of the page to the right; arrows are not marked on the graph to reduce clutter.

Leo-II breaks down the conjecture (the left-pointing pentagonal node) using logical inference, then uses the definitions to break down the conjecture's derivative clauses further. After further logical inference, it terminates with the double-circled node, which indicates that a contradiction has been derived. It is well-known that the growth of clause sets can be prolific. The subgraph formed of the white nodes in Figure 2.1 indicates the proof found by Leo-II for formula 2.1. That is, all other clauses can be erased since they do not contribute to Leo-II's proof. Figure 2.1 shows us that, in order to translate this particular proof, we need much less information compared to the information generated during proof search. (In fact, this turns out to be the case for the vast majority of proofs.)

So does this mean that we should treat the source prover as a black box? In an ideal world, the design of the translator need not emulate that of the source prover, and the translator would be able to directly map one proof into another. Unfortunately our world is seldom ideal in this respect. One often finds that a proof's description is underdefined, and that translation very often involves reconstruction. Underdefinition of proofs is not entirely harmful, since it keeps proofs small, and does not require the ATP system to expend resources logging minute details. Such details of the proof's construction may be logically uninteresting, but useful for translation. For instance, the order of conjuncts in a conjunction may change during the application of an inference, but this change might be an accidental or undocumented feature of the inference's implementation. As a result, a proof translation needs to be robust to such changes. Moreover, these details might vary between releases of the prover, or even between different runs of the same prover if it is operating in different modes. The literature on proof translation invariably contains reflections, and occasionally laments, related to this problem (Hurd, 1999; Zummer et al., 2004; Meng et al., 2006; Paulson and Susanto, 2007; Paulson and Blanchette, 2010; Dunchev et al., 2012; Böhme and Weber, 2011). We will explore this issue in more detail in Chapter 3, using various examples based on the widely-used TPTP problem set.

Because of underdefined proofs, perhaps some degree of emulation of the source ATP system by the translator is unavoidable. To what extent should the translator emulate a prover? Clearly, a full emulation would be more detailed than justified, as we have already seen. It would also be more expensive to implement: the translator would be *replicating* a prover. This is done, to some extent, in the implementation described in Chapter 5.

Rather than provide a detailed emulation, it might be easier to extend the source prover itself to produce more detailed proofs, or to produce a proof in a desired target format. We will explore this further in Chapter 3. Alternatively we could equip the translator with some limited proof search capabilities, to attempt to recover the missing information. This is used in the prototype implementation of the method described in Chapter 4. Even if we were to closely emulate a source system, having some proof search capability during translation could yield a more robust reconstruction, capable of withstanding undocumented variations carried out by the source ATP system. We will also use this in the implementation of the framework described in Chapter 5.

2.3 Design space of translators

2.3.1 Properties of translators

What can we expect of a proof translator? The most important theoretical property is *soundness*: if the translation succeeds, then the translated proof should prove the same theorem as the source proof. It would be preferable if the translation were also *complete*, meaning that all source proofs can be translated.

Other features involve a mix of theoretical and engineering-related qualities. Ideally the translation should be efficient in time and space. *Robustness* is the ability to handle small changes in the behaviour of the source or target provers. The change in behaviour might arise from the prover's configuration and might affect which inferences it uses, or how they manipulate formulas. A translator's *usability* is improved when the user need only provide minimal information to the translator—ideally the user need only provide the proof to be translated. A translator is *maintainable* when it can easily be updated to stay current with both the source and target provers. Finally, an *extensible* translator is one that can easily be adapted to handle new source/target provers, or proof formats.

Many of these qualities are fairly common engineering-related desiderata. They are particularly important to translators however, since a translator's usefulness suffers considerably otherwise. ATP systems are often research prototypes: they can change abruptly and rapidly from one version to the next; their behaviour can vary with only a flick of a parameter switch; and it is often the case that most of their documentation consists of source-code. The pace and scope of research makes it impossible to impose strictly normative quality standards across different systems: this would only be possible (or necessary) in industrial settings. The changes brought about by research can be rapid and unpredictable, and ideally a translator should be able to withstand them.

2.3.2 Kinds of translators

A translator can be positioned in three ways in relation to the source and target provers:

- An *exporter* is a translator that forms part of the source prover. It has access to the state of the source prover, and it is tasked with interpreting a proof for a target prover. Satallax's production of Coq scripts is an example of an exporter (Teucke, 2011).

- An *importer* forms part of a target prover. It can access and manipulate the internal state of the target prover, and it carries out a translation by interacting with this internal state. Sledgehammer is able to import proofs produced by various kinds of ATP systems to Isabelle/HOL (Paulson and Blanchette, 2010).

- A *transducer* does not have any internal knowledge about either party. Naturally, it is able to parse the proof of the source prover, and should be able to produce syntax that can be parsed by the target prover. An example of such a system will be given in Chapter 4.

If a transducer makes use of modules or libraries forming part of the source or target system, then the transducer becomes biased towards that system. This does not necessarily turn it into an importer or exporter. Writing reliable transducers is difficult since a transducer might need to simulate parts of the source or target prover.

Implementations of translators usually form pipelines for moving information from one calculus to another, without changing the theorem being proved.

2.3.3 State of the art

To my knowledge, the state of the art consists of five systems. All of them but one are *importer*-style translators. The exception is the system by Teucke (2011), which *exports* Satallax refutations in different formats. We build on this work in Section 3.2.

Two importers are currently implemented in Sledgehammer, and target the Isabelle/HOL proof assistant. The first of these uses the Metis ATP system to re-find proofs, and can generate Isar scripts checkable using Isabelle/HOL as described by Paulson and Susanto (2007). The second importer was developed by Böhme and Weber (2010), and it reconstructs resolution proofs produced by the Z3 SMT solver. It relies on the assistance of theory-specific solvers available in Isabelle/HOL. Both these systems work by re-finding, and have been extended to work for HOL proofs (Sultana et al., 2012). We will describe a third Isabelle/HOL importer in Chapter 5 that reconstructs proofs produced by the Leo-II ATP system.

The third system was developed by Keller (2013), to check resolution-based proofs produced by SMT solvers. Her approach consists of using a verified SMT solver. This solver makes use of verified theory-specific solvers. In order to check an SMT proof, the proof must first be translated into a form that can be checked by the verified solver. This transduction is implicit in the Sledgehammer system mentioned above, in the sense that it occurs as part of the importing process (by parsing the formulas in the source proof as formulas in Isabelle/HOL). Keller's takes a different approach: the transduction is done *outside* the proof checker. As with the systems described above, this translation is not verified, but it need not be, since the resulting proof is checked by a trusted system.

The last system is being developed by Chihani et al. (2013), and it is currently intended to reconstruct FOL proofs. Unlike the previous approaches, which are largely concerned with classical logic, the approach by Chihani et al. has a more principled method for handling non-classical logics, but at the cost of a more complex host calculus (Liang and Miller, 2011). Unlike the approach taken by Keller above, Chihani et al. do not rely on a preprocessing step during which the source proof is translated into a form that can be checked by the trusted proof checker. Instead, the mapping of the calculus is done by proof search, using a mechanism they call *clerks and experts*. This is a more sophisticated formula-interpretation idea compared to that used in Sledgehammer and in this dissertation.

2.3.3.1 Comparison

This section compares the five systems described above with the work in this dissertation. A more detailed description of related work is provided in Chapter 6.

I extend the work of Teucke, who developed a proof exporter, to target a new system. A disadvantage of proof exporters is that they cannot easily be reused by other ATP systems since they are tightly bound to a representation of proofs used in the source system. In contrast, *importers* are more amenable to be adapted to work with other proof sources, since one would expect to reuse the importer's backend. In Chapter 7 I will contrast Teucke's exporter-style approach with transducer and importer approaches I develop in this dissertation.

Keller's approach is elegant and sophisticated, involving the creation of a verified SMT proof checker by extracting its code from a constructive proof formalised in Coq. This can-

not easily be done in a system like Isabelle/HOL. Keller's method can be used both to *check* proofs using the verified proof checker, and to *import* proofs into a proof assistant. Importing a proof implicitly involves checking it. The method described in this dissertation does not require the use of a very expressive type system such as that of Coq, yet it allows the importing of proofs in a proof assistant (and therefore, the checking of those proofs).

The method of Chihani et al. uses a theoretically rich and sophisticated proof system. Their method is still being developed. To my knowledge, their work is currently not integrated with existing ATP systems, and it is currently mostly theoretical in nature. I opt for a simpler proof system, and focus more on implementation-oriented considerations of proof translation. In contrast with the work in this dissertation, Chihani et al. are currently focussed on first-order logic rather than higher-order logic.

The Sledgehammer work currently relies on using Metis and Z3 for re-finding. I want to lessen the reliance of re-finding using a different system such as Metis and Z3, and to make more use of the proof itself to improve the success of reconstruction.

In summary, this dissertation seeks to address a gap in knowledge about the three different kinds of translators, and to develop an extensible reconstruction framework.

2.4 Reasoning framework

The reasoning framework used in this thesis is classical higher-order logic encoded in simple type theory (Church, 1941). This is also called *classical type theory* (Andrews, 2001). This reasoning framework is summarised in this section. Benzmüller et al. (2004) give a detailed account of the logic and its semantics.

Intuitively, each element of the universe of discourse can be referenced by means of a *term*, and the universe is partitioned into *types*. Each term can only be of (or semantically *belong to*) a single type.

Throughout the rest of the dissertation, the *mention* (as distinguished from the *use*) of a symbol or token, will be indicated semantically in English. For example, in the phrase "the symbol τ is a Greek letter, and it is not of type τ", the first occurrence of τ involves mentioning it, while the second occurrence involves using it. One could use quoting to differentiate meaning from use, but this adds clutter, harming the presentation. As explained later, we will also make use of background shading to emphasise subterms, or to avoid clutter due to parentheses.

Types. Let the symbol τ range over types, and the symbol $\mathcal{T}$ over terms. There are two kinds of types. *Function* types denote elements that can be applied to other elements (say, of type τ_1) to yield elements of a possibly-different type (say, τ_2). Such a function type is written $\tau_1 \to \tau_2$, where the infix symbol $\to$ can be regarded to be a binary right-associative operator at the level of types. *Base* types denote elements that do not behave as functions, as their elements cannot be applied to other terms. The universe of discourse contains at least two base types: propositions, written o, and individuals, written ι. In this framework, formulas are but propositionally-typed terms in the language. The denotation of the proposition type in the universe of discourse contains only two elements, which are directly described by the constant terms True and False. All formulas describe a value of type o, and proof serves to establish the theorem-hood of a formula—showing that it describes the same semantic element as True.

Syntax categories, grouping and levels. To aid readability, different categories of syntax will be distinguished through the use of different typefaces and alphabets. Calligraphic text will be used for metavariables, such as the term metavariable $\mathcal{T}$.

As usual, syntax will be grouped using brackets to disambiguate the intended tree structure of a term. Only curved brackets will be used for this, since square and curled brackets will be given special meaning.

Where convenient, or to highlight a particular syntactical structure, syntax may be grouped using background shading instead of brackets. Thus $((p = \mathsf{True}) \vee (p = \mathsf{False}))$ could be written as $(p = \mathsf{True}) \vee (p = \mathsf{False})$ or instead as

$$p = \mathsf{True} \vee p = \mathsf{False}$$

Terms. Terms are built using four kinds of primitives. We have already encountered *constants*, such as True and False. Such terms have fixed denotations. Typographic conventions will be used to distinguish different kinds of syntax, and constant names are written using sans serif Latin characters. A *variable* is a term that denotes some element in a fixed type. Unlike constants, the denotation of variables is not fixed. Variables are named using italic Latin characters, except for o, which is unfortunately indistinguishable from the Greek o, and which is reserved for the type of propositions as seen earlier. Finally we come to the primitives that are used to describe and apply functions. *Abstraction* describes terms of function type, using the notation $\lambda x.\, \mathcal{T}$. Recall that $\mathcal{T}$ is a metavariable ranging over terms, and that x is a variable. $\mathcal{T}$ is called the *body* of the abstraction term. Let x have type τ_1 and $\mathcal{T}$ be of type τ_2; then $\lambda x.\, \mathcal{T}$ is of type $\tau_1 \to \tau_2$. The λ symbol *binds* the variable x. That is, occurrences of the symbol x in the body depend on the occurrence of x at λ for its meaning, unless they occur within another λ-occurrence of the same variable. That is, in the term $\lambda x.\lambda x.\, x$ True, the highlighted occurrence of x depends on the inner λ. In this calculus, λ fixes the meaning of x using the argument received by this function. The last form of term formation is *application* of a function to an argument. Application is described by juxtaposing the function term and its argument. For example, $(\lambda x.\, x)\ \mathsf{True}$.

Typing annotations. When the type of a term is ambiguous but relevant, it may be specified via an explicit *typing*, for instance $x : o$ is equivalent to the term x but indicates that x is a propositional variable. Similarly, $y : (\iota \to o)$ tells us that y ranges over predicates over ι.

Term equivalences. We now turn to some important relations over terms. We will define these relations over terms of the same type. We start with the identity relation over terms, giving us a strong notion of equivalence. This equivalence can be weakened to yield a relation called α-equivalence, by disregarding the names of bound variables (but not at the expense of binding structure). For example, this would make $\lambda x.\, x$ and $\lambda y.\, y$ equivalent, but would distinguish $\lambda x.\, (y\ x)$ from $\lambda y.\, (x\ y)$. Intuitively this is sensible since $\lambda x.\, x$ can be replaced by $\lambda y.\, y$ and still behave as the identity function, whereas the *free* (i.e., unbound) variables in the other examples make those terms' meanings dependent upon the context (i.e., the containing term) in which they occur. If all the variables in a term are bound, then the term is said to be *closed*.

β-equivalence is another weakening to the identity relation. Intuitively, it relates applications of abstraction terms to the term resulting from the function's application. For example, $(\lambda x.\ \mathcal{T}_1)\ \mathcal{T}_2$ is β-equivalent to $\mathcal{T}_1[\mathcal{T}_2/x]$, where the phrase $[\mathcal{T}_2/x]$ denotes a meta-level operation over terms that substitutes all free occurrences of x with $\mathcal{T}_2$. Thus, for example, $(\lambda x, y.\ x)\ \mathsf{True}$ is β-equivalent to $\lambda y.\ \mathsf{True}$. The comma notation used in $(\lambda x, y.\ x)\ \mathsf{True}$ abbreviates repeated application of a binder; in this case, the formula has the same meaning as $(\lambda x.\ \lambda y.\ x)\ \mathsf{True}$.

Another weakening of the identity relation involves equating certain function-valued terms. Specifically, we equate $\mathcal{T} : \tau_1 \to \tau_2$ with terms like $\lambda x.\ \mathcal{T}\ x$ if x does not occur free in $\mathcal{T}$. The type of the resulting term is unchanged; intuitively this equivalence means that any term of function-type can be written as an abstraction term. This is called η-equivalence.

In what follows we will take $\alpha\beta\eta$-equivalence to be the intended equivalence over terms. Furthermore, the canonical form of terms is the *reduced* forms of β-equivalence and *short* form of η-equivalence. Translating terms into their canonical form is called *normalising* the term. The canonical form for a term always exists, and it is unique. Further details on the meta-theory of the typed λ-calculus are given by Barendregt et al. (1977).

Interpreted constants. So far we have encountered the constants True and False, both typed o. The constants interpreted by a language are specified by the language's *signature*, usually denoted by the symbol Σ. There are several constants that are considered meaningful for logical reasoning. We will not fix a signature and interpretation here; we will only describe some symbols and their intended meaning. For instance, the constants $\mathsf{not} : o \to o$ and $\mathsf{or} : o \to o \to o$ usually denote the negation and disjunction functions in the semantics. The family of constants $\mathsf{all}_\tau : (\tau \to o) \to o$ (at each type τ) denotes the universal quantification combinator. Following the usual convention, we write $\forall x.\mathcal{T}$ instead of $\forall(\lambda x.\mathcal{T})$. The combinator has a uniform behaviour across each type τ: that is, $\mathsf{all}_\tau\mathcal{T}_1$ maps to the denotation of True if for each term $\mathcal{T}_2$ typed τ, $\mathcal{T}_1\mathcal{T}_2$ maps to the denotation of True. Otherwise, $\mathsf{all}_\tau\mathcal{T}_1$ is equivalent to False. The family of constants $\mathsf{eq}_\tau : \tau \to \tau \to o$ denotes equality, and it has a denotation for all types τ. The meaning of equality will be explained below.

Following convention, we will use the symbol $\neg$ for not, infix $\vee$ for or, $\forall_\tau$ for all_τ and infix $=_\tau$ for eq_τ. The type annotations in all_τ and $=_\tau$ will not be shown when unimportant.

Constants can be defined using terms. For instance, the constant $\longrightarrow$ can be defined in terms of $=_{o\to o\to o}$, $\neg$ and $\vee$:

$$\longrightarrow\ =_{o\to o\to o}\ \ \lambda x, y.\ \neg x \vee y$$

Note that the implication arrow $\longrightarrow$ has a completely different role from that of the type arrow $\to$, and the two cannot be used at the same level: that is, $\longrightarrow$ can only be used at the level of terms, and $\to$ at the level of types. Both arrows are right-associative, and in some formalisms they are indeed identified (Sørensen and Urzyczyn, 2006), but not in the one we are using.

Models. To give meaning to formulas written in this language, we will rely on the semantical framework described by Henkin for higher-order logic, as described and developed by Benzmüller et al. (2004). In this framework, a model $\mathcal{M}$ is a four-tuple $(\mathcal{D}, @, \mathcal{E}, v)$ where $\mathcal{D}$ is the universe of discourse (partitioned by *types*), $@$ is the function application operator (defined over elements of the universe of discourse), $\mathcal{E}$ is an evaluation function mapping

terms into elements of $\mathcal{D}$, and v maps the denotations of object-level propositions (forming a distinguished subset of $\mathcal{D}$) into a meta-level propositional domain. In our setting, v simply maps to the Boolean domain. As is standard, we say that a formula (or a set of formulas) is *satisfiable* if there exists a model in which that formula (or every formula in the set) is true. We will use the words satisfiable and *consistent* interchangeably. A formula is said to be *valid* if it is true in *all* models.

The framework described by Benzmüller et al. (2004) allows one to focus on different model classes, according to the semantics sought. The logic we will use is *classical*, in the sense that the denotation of a formula is equal to the formula's double negation. The logic is also *functionally extensional*, meaning that if two functions have identical images, then the two functions are identical: $\forall(f : \tau_1 \to \tau_2)\, g.\ (\forall x : \tau_1 . f\, x =_{\tau_2} g\, x) \longrightarrow f =_{\tau_1 \to \tau_2} g$. The logic is also *propositionally extensional*, meaning that the constant $=_o$ is semantically identified with bi-implication. As a result, we do not need to specify an additional symbol for bi-implication. The intended class of models for this logic is that of *Henkin models*.

2.5 TPTP

The Thousands of Problems for Theorem Provers (TPTP) project gathers a library of problems used for testing and evaluating Automated Theorem Provers (ATP systems) (Sutcliffe, 2009). Prior to TPTP, ATP systems often had their own ad hoc languages, and test problems existed in disparate sources—often in hard-copy publications! The TPTP project has driven the creation of an open standard input language used across a broad variety of ATP systems, and a suite of tools to transform and analyse the problems in the TPTP library.

2.5.1 Annotated Formulas

The basic unit of a TPTP-encoded problem or proof consists of an *annotated formula* (AF). A TPTP problem or proof consists of a sequence of AFs. A TPTP-encoded problem could consist of a set of axioms followed by a conjecture. A TPTP-encoded proof could consist of references to the TPTP axioms and conjecture from the problem, and a collection of inferences showing how the conjecture is a consequence of the axioms. These inferences are themselves AFs annotated with inference-specific information (such as which inference rule was used in that inference step). In this dissertation we will frequently encounter use of TPTP-encoded examples of problems and proofs.

An AF is encoded using a 7-bit ASCII character encoding, and consists of a five-tuple representing a formula in a specific logic. It carries the following information:

1. *Language*. This states the language in which the formula (and, in some cases, some parts of the annotation) are encoded. There are various kinds of TPTP languages, such as CNF, FOF, TFF, and THF. These languages are described in the Glossary at the beginning of this dissertation. TPTP languages share similar syntactic features, and this allows TPTP syntax to describe a broad class of languages.

2. *Name*. Formulas are named to allow them to be referenced in a proof.

3. *Role*. This is used to indicate how the formula may be used, or how it is obtained. Example roles include *axiom*, *conjecture*, and *type*. The *type* role is used when declaring the type of a symbol.

4. *Formula*. The syntax of the formula depends on the language being used. For instance, when using FOF (first-order form) we can only quantify over individuals, but in THF (typed higher-order form) formulas can quantify over values of any type.

5. *Annotation*. This provides further information about the formula's provenance or derivation, such as the inference rule used to derive the formula, and the rule's premises. It is not always included in an AF, as we shall see below. We encounter it very frequently in TPTP-encoded proofs however.

For ease of reference, TPTP problems are given unique names. For example, the TPTP problem PUZ082^1 encodes the following puzzle: prove that despite Peter claiming that everything he says is a lie, not everything he says is a lie.

This is encoded in THF, the syntax for higher-order logic within TPTP, as follows. First, the signature is spelt out. It contains two constants: peter (an individual) and says (a function typed $\iota \to o \to o$). The constant peter is declared as follows:

```
thf(peter, type, (peter: $i)).
```

The token `thf` indicates the TPTP language, `peter` is the AF's name, `type` is the role, and `peter : $i` is the formula—it is actually a typing here. The symbol `$i` is the TPTP syntax for the type ι of individuals. As we can see from the syntax, this AF does not carry an annotation.

The constant says is declared similarly:

```
thf(says, type, (says: $i > $o > $o)).
```

Note that the symbol `$o` is the type of propositions, and the symbol `>` is the type constructor for functions.

Next, we have the axiom that Peter says that everything he says is a lie. This is formalised as

$$\mathsf{says}\ \mathsf{peter}\ (\forall\, x.\ \mathsf{says}\ \mathsf{peter}\ x \longrightarrow \neg x)$$

and in THF this is encoded as follows. Following the structure we have seen before, `thf` is the language, `ax1` is the formula name, and `axiom` the role. In standard mathematical notation, application is expressed by juxtaposition, but in THF it is noted explicitly using the symbol `@`. The symbol `=>` is the implication arrow, and the notation `! [X: $o]` is the universal quantification over symbol `X`, which has propositional type.

```
thf(ax1, axiom,
    ( says @ peter
    @ ! [X: $o] :
        ( ( says @ peter @ X )
       => ~ ( X ) ) )).
```

Finally, we have the conjecture

$$\neg(\forall\, x.\ \mathsf{says}\ \mathsf{peter}\ x \longrightarrow \neg x)$$

which in THF is encoded as

```
thf(thm, conjecture,(
    ~ ( ! [X: $o] :
          ( ( says @ peter @ X )
         => ~ ( X ) ) ) )).
```

We will now look at a fragment of proof encoded in THF, generated using the Leo-II ATP system.[2] As in a TPTP-encoded problem, a proof consists of AFs. Each AF describes a type declaration or some axiom or conjecture. A proof also contains other AFs to describe the steps of the proof. Each AF could be regarded as a named vertex, and the proof as a DAG—similar to the white-coloured subgraph we saw in Figure 2.1.

An inference-related AF in a proof also contains other information, such as whether the inference is validity-preserving. The AFs shown below are from Leo-II's proof of the previous problem. The first AF, labelled 17, is the premise of the inference; the second AF, labelled 20, is the conclusion.

```
thf(17,plain,(
    ! [SV1: $o] :
      ( ( ( says @ peter @ SV1 )
        = $false )
      | ( ( ~ ( SV1 ) ) = $true ) ) ),
    inference(extcnf_not_pos,[status(thm)],[14])).

thf(20,plain,(
    ! [SV1: $o] :
      ( ( SV1 = $false )
      | ( ( says @ peter @ SV1 )
        = $false ) ) ),
    inference(extcnf_not_pos,[status(thm)],[17])).
```

Together, the AFs form the inference shown below. The notation used will be explained further below, but it is summarised briefly here. The symbol SV_1 appearing outside the braces is a clause-level variable: it is implicitly universally quantified, and its scope extends throughout the adjacent clause. The clause contains literals, using the notation ${}^{p}\boxed{F}$, where F is a HOL formula, and $p \in \{+, -\}$ is the literal's polarity.

$$\frac{SV_1 \left\{ {}^{-}\boxed{\text{says peter } SV_1}, {}^{+}\boxed{\neg\, SV_1} \right\}}{SV_1 \left\{ {}^{-}\boxed{SV_1}, {}^{-}\boxed{\text{says peter } SV_1} \right\}}$$

The annotation relevant to this inference occurs in the second annotated formula: `inference(extcnf_not_pos,[status(thm)],[17])`. This tells us the inference rule used (i.e., `extcnf_not_pos`), that the inference was validity-preserving (through the flag `status(thm)`), and that the inference has a single premise clause, named `17`.

The standardisation of formula languages through the TPTP initiative enables developers and researchers to understand and analyse a large, common collection of logic-encoded problems. This also enables the reuse of parsing code for processing both problems and their proofs. However, when it comes to proofs, this advantage is only limited to the ability to read specific instances of inferences. We may have partly escaped the babelesque chaoticdom by having a standardised language of formulas, but since the language of *inferences* is *not* standard we are not completely spared. An approach to standardising the language of inferences, via embeddings into a common logical framework, will be discussed in Section 5.4. Similar approaches in the literature are described in Chapter 6.

[2] The TPTP-encoded Leo-II proofs referenced in this dissertation are available online at the URL `http://www.cl.cam.ac.uk/~ns441/files/recon_framework_results.tgz`

2.6 Systems

This section outlines key characteristics of the ATP systems that were modified as part of the work described in this dissertation.

In practice, almost every computer proof tool implements its own ad hoc version of a logic. Even systems claiming to share the same logic often vary in what they implement. For instance, Leo-II, Isabelle/HOL and Satallax are all tools for reasoning in classical higher-order logic, but they have different proof calculi and base signatures (i.e., primitive constants). Moreover, both Leo-II and Satallax accept command-line switches that vary their proof calculi (by enabling or disabling certain rules). Unless otherwise specified, the results described in this dissertation were obtained using Leo-II version 1.6, Satallax version 2.7, Isabelle/HOL 2013, and E version 1.8.

2.6.1 Leo-II

Many ATP systems are designed around the idea that a set of formulas is processed iteratively, using *consistency-preserving* inferences, until it is trivial to check whether the resulting set is consistent or trivially inconsistent. It is possible that the computation never terminates if the logic is undecidable. The algorithms that transform this set preserve the consistency or inconsistency of the formulas in the set.

The overall process binding together the algorithms and the set of formulas can be organised in various ways. One of the best-known is the *given-clause algorithm*, pioneered by McCune (1990). This algorithm partitions the set of formulas into the *passive* and *active* sets. Initially, all formulas are in the passive set. These formulas (or their generalisations) are then activated and moved into the active set during successive iterations of the algorithm. Leo-II uses a variant of this algorithm called the DISCOUNT loop (Avenhaus et al., 1995).

Figure 2.2 shows the sequence of internal states of Leo-II during proof search, as it iterates to solve the simple problem posed by Formula 2.1.[3] The figure shows the contents of the active and passive clause sets at each iteration of Leo-II's *main loop*. An ATP system often operates in at least two stages. The *preprocessing* stage usually involves bringing the input problem into some normal form, and carrying out simplifications. This stage then hands over to the *main loop*, which iteratively processes a set of formulas until they can be determined to be consistent or inconsistent. Naturally, this process is not guaranteed to terminate if the logic is undecidable.

Leo-II implements a RUE (Resolution with Unification and Equality) proof calculus (Sultana and Benzmüller, 2012), and uses an extended unification algorithm to handle interpreted constants specially. For instance, the unification algorithm recognises that equality is a commutative operator. The proof calculus uses splitting (Nonnengart and Weidenbach, 2001) to make clauses smaller where possible. It also uses a shared term representation and indexing for improved performance (Theiss and Benzmüller, 2006). Other features include strategy scheduling and a variety of translations into first-order logic (Benzmüller and Sultana, 2013). The translations are used by Leo-II to collaborate with FOL ATP systems.

Leo-II's proof calculus is detailed in Appendix A.1. Almost all of its rules are *validity-preserving*. That is, the conclusion is valid (satisfied by all models) if the premises are valid. We can embed the rules in Isabelle/HOL using schematic meta-rules, or tactics when more

[3] Leo-II was instrumented to produce this information if the GIVENCLAUSEGRAPH preprocessor definition is active at compile time.

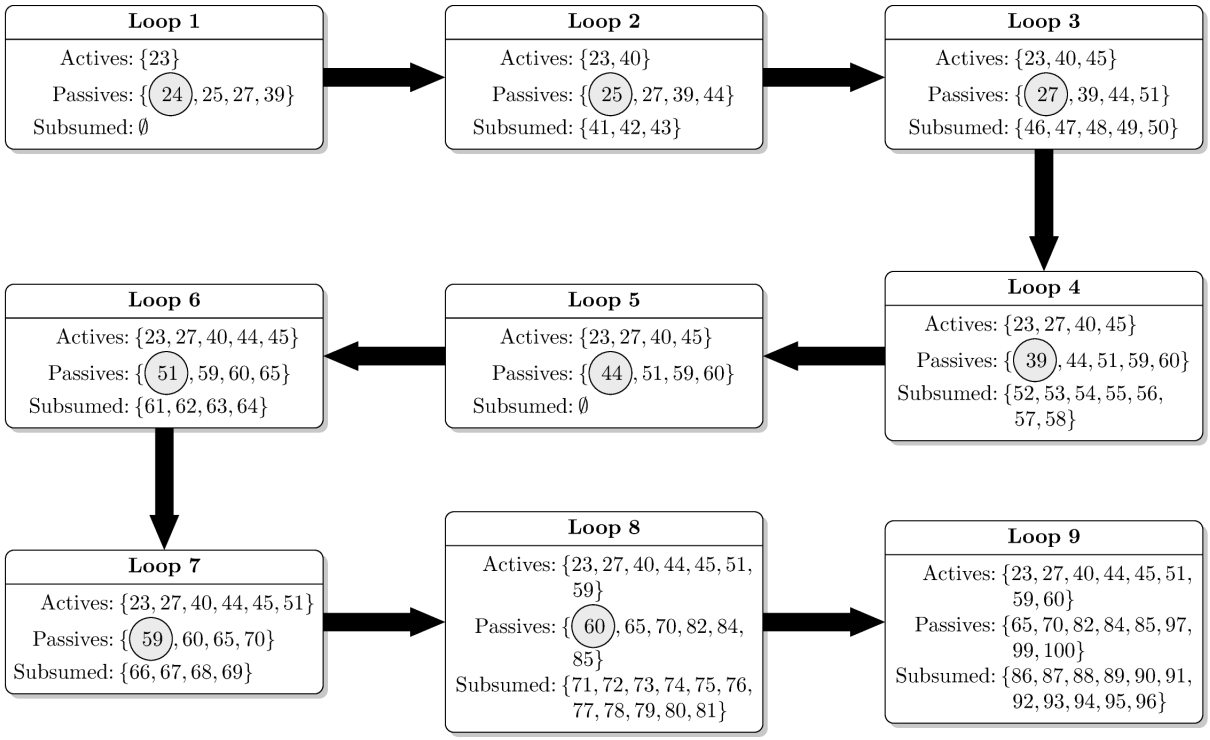


Figure 2.2: A typical execution of Leo-II, in terms of the contents of its main datastructures: the active and passive clause sets. The execution is shown as a sequence of states. Each state shows the contents of the datastructures at the end of Leo-II's main loop. Clauses are represented by their numerical indices. Following the given-clause algorithm, the contents of the active and passive sets persist across loops, while the set of subsumed clauses is computed at each loop. New clauses that are not subsumed are added to the passive set. At each loop, a passive clause is selected. The index of a clause sometimes changes when the clause is moved from the passive to the active set, so here we indicate the passive clauses that are activated in the *next* loop by circling them. For example, clause 24 is circled in the first state because it is activated during the second loop, but Leo-II renames it to 40 when it is added to the active set in the second loop. Note that Leo-II is always picking the oldest passive clause: that is, the one with the smallest index. The execution starts with a set of clauses consisting of the conjecture and a collection of set-theoretical definitions (such as that for subset and union). Clauses are numbered sequentially. Leo-II starts by preprocessing the problem. Note that definitions are unfolded during preprocessing. During the first execution of the main loop, Leo-II activates the 23rd clause (as we can see, it is the sole member of the active set). We can see that none of the clauses produced by clause 23 in the first loop are subsumed. During the second run of the loop, Leo-II picks clause 24 for activation. Clause 24 has its variables renamed, and the resulting clause is called 40 when it is added to the active set. The processing of clause 24 adds a clause (44) to the passive set. Three other clauses it produces (41, 42, 43) are subsumed away. This continues on, until a refutation is found during the tenth iteration, which is not shown here.

expressiveness is needed. The only two exceptions are Skolemization and splitting. These rules are consistency-preserving, not validity-preserving, and they cannot be used directly in a validity-preserving calculus such as that of Isabelle/HOL. We handle these rules by applying proof transformations; we will focus on this in Chapter 4.

Formulas in Leo-II are represented using *general clauses*. *Clauses* consist of collections of *literals*. A literal consists of an atomic formula and a *polarity*. The polarity simply indicates whether the atom is being asserted positively or negatively. *General clauses* can have arbitrary formulas in literals, rather than simply atomic formulas. In what follows, we refer to general clauses simply as "clauses". We will use the following notation for literals:

$$^{+}\mathcal{T} \qquad ^{-}\mathcal{T}$$

where $\mathcal{T}$ is a propositionally-typed term. Semantically, $^{+}\mathcal{T}$ is equivalent to the formula $\mathcal{T} = \mathsf{True}$, and $^{-}\mathcal{T}$ is equivalent to $\mathcal{T} = \mathsf{False}$. For example, the literal $^{-}\mathsf{a} \vee x$ has the same meaning as the formula $\neg\,(\mathsf{a} \vee x)$.

Semantically, a clause corresponds to the disjunction of its literals. An empty clause is usually denoted by $\square$, and it is semantically equal to the denotation of False. A singleton clause is called a *unit* clause. We will use the following notation for clauses:

$$x, y \left\{ ^{+}\mathsf{a} \vee y\,,\ ^{-}\mathsf{a} \vee x \right\}$$

This clause encodes the formula $\forall x, y.\ (\mathsf{a} \vee x) \longrightarrow (\mathsf{a} \vee y)$. Note that literals *may* have free variables, but these *must* appear in the clause's prefix, otherwise the clause is malformed. Variables appearing in this prefix are universally quantified.

For another example, the following scheme represents the class of all clauses containing a positive tautology literal.

$$\left\{ ^{+}\mathcal{T} \vee \neg\,\mathcal{T}\,, \ldots \right\}$$

In a clausal calculus, proofs usually take the form of refutations: the negated conjecture is used to derive further truths, perhaps with the assistance of other axioms, until the empty clause is derived.

Inference Rules. Inference rules are schemes that can be instantiated to form the chain of reasoning in a proof. We will use the standard notation for inference rules. We have already seen an example of a specific inference in Section 2.5.1. The following example shows an inference rule from Leo-II's calculus. It breaks up a disjunction in a positive literal of the premise clause, forming two positive literals in the conclusion clause.

$$\frac{\left\{ ^{+}\mathcal{T}_1 \vee \mathcal{T}_2\,, \ldots \right\}}{\left\{ ^{+}\mathcal{T}_1\,,\ ^{+}\mathcal{T}_2\,, \ldots \right\}}\ \vee_{\mathrm{E}}$$

It is important to distinguish inference rules that only preserve satisfiability from those that preserve validity. We will use solid inference lines for validity-preserving inference rules, and dashed lines (as used above) to indicate satisfiability-preserving inference rules.[4] We will

[4] Even if the rule is validity-preserving, we might use a dashed line to emphasise that the rule is being used in a consistency-preserving calculus.

use double inference lines to emphasise, where appropriate, that the rule is *reductive*. That is, it removes the premise clauses from the search space, and adds the conclusion clauses. This notation will be recalled when it is used.

2.6.2 Satallax

Satallax implements a higher-order tableau calculus (Backes and Brown, 2011). It generates successively refined propositional abstractions of the input (higher-order) formulas, and relies on a SAT solver to check the resulting propositional clauses. If the set of clauses is satisfiable, and it is possible to carry out inference over the corresponding higher-order formulas, then additional propositional clauses are generated (corresponding to the higher-order formulas derived by the inference), and the SAT solver is invoked again. This method is a distinctive feature of Satallax's design. An example of its execution will be given in Section 3.2.

This method is repeated until a tableau branch cannot be closed, which indicates that the original higher-order order formulas must be consistent, or all branches are closed, in which case the higher-order formulas must be inconsistent. (Since HOL is not decidable, this process could continue forever, or until a resource bound is exceeded.)

Another distinctive feature of Satallax is its representation of refutations, developed by Teucke (2011). This involves reconstructing a refutation found by Satallax, and representing it in an intermediate datatype from which a Coq (Bertot and Castéran, 2004) script can be output for independent proof checking. This is a form of proof exporter (described in Section 2.3.2). It will be described further in Section 3.2.1.

2.6.3 Isabelle/HOL

Isabelle (Paulson, 1994) is a programmatic framework for building interactive reasoning tools. It provides a *meta-logic* and an associated API to facilitate encoding and reasoning with various logics. Such logics are called *object-logics*. Formulas in the meta-logic are called *meta-formulas*, while those in an object-logic are called *object-formulas*. Essentially, by using the meta-logic we can reason *about* an object-logic, and by using the object-logic we can reason *within* the object-logic. The role of a meta-language is discussed by Harrison (1995), and the development of Isabelle's meta-logic is detailed by Paulson (1989). Isabelle itself has another meta-language in turn: Standard ML (Milner et al., 1997).

Isabelle is a relative of the LCF system (Gordon et al., 1979), from which it reused several ideas—not least the encapsulation of its core inferences in an abstract datatype. In LCF, meta-theorems are encoded as functions of Standard ML that transform LCF theorems. In Isabelle, meta-theorems are terms *within* Isabelle. This is convenient for inspecting their values—in general one cannot directly inspect the value of an ML function! Meta-theorems play an important role in fine-grained proof translation (described in Section 2.1) since inference rules are meta-theorems.

Among LCF and its descendents, Isabelle is unique in its intention to be a *generic* proof assistant: it is designed to facilitate reasoning in different logics. Isabelle/HOL (Nipkow et al., 2002) is the extension of Isabelle to reason in classical, higher-order predicate logic. Isabelle/HOL is based on the formalisation used in the HOL system (Gordon, 1985; Gordon and Melham, 1993), which in turn is based on simple type theory (Church, 1941). This logic was described earlier in Section 2.4. Both the HOL system and Isabelle improve Church's

system through the support for parametric polymorphism.[5] Isabelle additionally supports ad hoc polymorphism via type classes (Wenzel, 1997). The source provers we use in this dissertation (Satallax, Leo-II, and E) do not support polymorphism natively in the input language (indeed, E's calculus is untyped). From here on, by "HOL" we mean (monomorphic) higher-order logic, as described by Benzmüller et al. (2004).

Isabelle's formalism is based on the simply-typed λ-calculus (Church, 1940), and it uses a linear notation for rules. For instance the rule

$$\frac{A \wedge B}{B}$$

is encoded in Isabelle as $\mathsf{Trueprop}(A \wedge B) \Longrightarrow \mathsf{Trueprop}(B)$. Note that $A \wedge B$ and B are terms in the object language (say, propositional logic), whereas these are terms in the meta-language: $\Longrightarrow$, $\mathsf{Trueprop}(A \wedge B) \Longrightarrow \mathsf{Trueprop}(B)$, $\mathsf{Trueprop}(A \wedge B)$ and $\mathsf{Trueprop}(B)$. In the formalisation of HOL in Isabelle, the constant symbol $\mathsf{Trueprop}$ serves as a truth judgement in the meta-language: it maps propositions in the object-language (HOL) to propositions in the meta-language (Isabelle). In what follows we will not explicitly include Trueprop, to avoid clutter. In HOL, propositions take Boolean values. Isabelle is a minimal logic, and its propositions can be interpreted as witnesses to the inhabitation of simple types (Berghofer, 2003).

Isabelle interprets three primitive constants: the symbol $\equiv$ denotes identity of propositions up to $\alpha\beta\eta$-equivalence, $\Longrightarrow$ is an implication between propositions, and $\bigwedge$ is a universal quantifier. For example, the implication-introduction rule of HOL

$$\frac{\begin{array}{c}[P] \\ \vdots \\ Q\end{array}}{P \longrightarrow Q}$$

is formalised as an Isabelle/HOL meta-formula in Isabelle as the sentence

$$\bigwedge P : o, Q.\ \Big(P \Longrightarrow Q\Big) \Longrightarrow P \longrightarrow Q\,.$$

The highlighted subformulas are those written in Isabelle/HOL. Note that the symbols P and Q, both typed o, are bound at the meta-level. The modus ponens rule of HOL is formalised in Isabelle as

$$\bigwedge P : o, Q.\ P \Longrightarrow P \longrightarrow Q \Longrightarrow Q\,.$$

Thus inference rules are *linearised* into λ-calculus terms. For more compact notation, nested $\Longrightarrow$ applications are collected and written as follows. An inference rule such as

$$\frac{\mathcal{T}_1 \qquad \mathcal{T}_2}{\mathcal{T}_1 \wedge \mathcal{T}_2}$$

[5] Isabelle/HOL inherits this, and other, features from Isabelle, its meta-language.

is encoded as

$$\mathcal{T}_1 \Longrightarrow \mathcal{T}_2 \Longrightarrow \mathcal{T}_1 \wedge \mathcal{T}_2$$

but written more compactly as

$$\left[\mathcal{T}_1, \mathcal{T}_2\right] \Longrightarrow \mathcal{T}_1 \wedge \mathcal{T}_2$$

Any $\bigwedge$-bound variables that are in scope across the entire meta-formula are written in the formula's prefix, as shown next.

$$\bigwedge \vec{x}. \left[\mathcal{T}_1, \ldots, \mathcal{T}_n\right] \Longrightarrow \mathcal{T}$$

If the object logic is HOL, then this meta-formula is semantically equal to

$$\forall \vec{x}. (\mathcal{T}_1 \wedge \ \ldots \ \wedge \mathcal{T}_n) \longrightarrow \mathcal{T}.$$

Isabelle's notation is similar to that used for non-classical sequents (since there can only be a single formula in the sequent's consequent). Isabelle's notation is more general since meta-formulas can be nested. For example, the following rule formalises conjunction elimination in Isabelle/HOL's calculus:

$$\left[\mathcal{T}_1 \ \wedge \ \mathcal{T}_2, \ \left[\mathcal{T}_1, \mathcal{T}_2\right] \Longrightarrow \mathcal{T}_3\right] \Longrightarrow \mathcal{T}_3$$

Theorems are proved in Isabelle by transforming a state, itself consisting of a theorem. This process works by backward chaining, and it will be outlined next; it is documented in more detail by Paulson (1994) and Wenzel (2002). Proof of a formula ϕ in Isabelle starts at the initial state (theorem) consisting of $\phi \Longrightarrow \phi$, which is obviously valid. This state is transformed by applying a chain of rules, or meta-theorems, $\psi_1 \Longrightarrow \phi$, $\ldots$, $\psi_{n+1} \Longrightarrow \psi_n$, yielding the state $\psi_{n+1} \Longrightarrow \phi$. If ψ_{n+1} is an axiom, then it is logically equivalent to the formula $\mathsf{True} \Longrightarrow \psi_{n+1}$. Applying this rule gives us the final state $\mathsf{True} \Longrightarrow \phi$, which is equivalent to the formula ϕ. Thus ϕ is valid since we started from the tautology $\phi \Longrightarrow \phi$ and made a series of validity-preserving inferences.

Conclusion

This chapter outlined the basic concepts used in this dissertation, and described the languages and tools that will be used in the remainder of the dissertation. A more detailed survey of related work is given in Chapter 6.

A proof is but a piece of data, and like any other data it is amenable to processing. In the next chapter we will focus on proof search and the generation of proofs, including the translation of proofs at source. Following that we will describe ideas for how to translate these proofs once they have left the ATP system.

Chapter 3

Proof generation

A proof-finding tool explores a space of formulas and their consequences. If the tool succeeds in finding a proof, it could output a trace consisting of a trajectory of inferences between the input formulas and the proof's conclusion. The description of the proof in this way is not always detailed enough to facilitate its translation. Emulating the syntactical transformations done during an inference step can be non-trivial. In this chapter we look at how the proof generation of Leo-II was modified to facilitate proof translation, and how the proof generation of Satallax was modified to produce Isar scripts, which Isabelle/HOL can check directly. We also look at how Leo-II's cooperation with first-order provers was improved.

Should we modify an Automated Theorem Prover (henceforth abbreviated to *ATP system*) to facilitate interfacing with it, or should we treat it instead as a black-box and try to engineer our way around problems of interfacing?

Different answers to this question were explored in part by Joe Hurd. Hurd (1999) built GANDALF_TAC, a tactic that interfaced the Gandalf ATP system with the HOL system (Gordon and Melham, 1993). The tactic first translated first-order HOL goals into the input language of Gandalf. If Gandalf succeeded in proving the goal, then the tactic translated Gandalf's proof to produce a theorem in the HOL system. Hurd avoided influencing Gandalf's design in any way, writing that "it would have been easy to contact the author of Gandalf and ask him to put enough detail into the proofs to completely disambiguate them. However, we decided to remain faithful to the spirit of the project by treating Gandalf as a black-box, and looked instead for a solution with GANDALF_TAC." Later, Hurd (2003) built the Metis ATP system with the intention to interface it with the HOL system. This was somewhat similar in spirit to the tableau prover written by Paulson (1999), interfaced with Isabelle. Naturally, using Metis afforded Hurd much greater flexibility in ensuring a smooth exchange of information between the HOL system and the ATP system compared to when he used Gandalf.

Initially I resisted modifying the provers, but then realised that modification not only facilitated the implementation of proof translation, but in the case of Leo-II it also saved a lot of time during proof reconstruction, by avoiding search. As a result, unlike Hurd in his work on GANDALF_TAC, in this disseration I have been more liberal about modifying ATP systems. GANDALF_TAC was intended to serve as an off-the-shelf, black-box tool in the Prosper system (Dennis et al., 2003), instantiating its generic plug-in interface. Notwithstanding being bound by normative constraints—related to well-formed TPTP proofs, for instance—I did not have to treat the provers as black-boxes. In addition to modifying Leo-II to produce more detailed proofs, I also explored strengthening proof search in Leo-II by im-

proving its interface with first-order provers. I also modified Satallax to produce Isar scripts. These changes will be motivated and described next.

3.1 LEO-II

3.1.1 Refining the calculus

The purpose of this section is to describe shortcomings in LEO-II's proof output, for the purpose of reconstructing those proofs, and how these shortcomings were addressed. In this section we only look at specific LEO-II inferences, before we proceed to study the reconstruction of entire proofs in the next chapters.

Initially I was reluctant to modify LEO-II, and sought to process its proofs directly. However, two inference rules made this rather difficult: extcnf_combined and ext_uni. These are *compound* rules, consisting of applications of primitive clausification and unification rules respectively. For ease of reference, LEO-II's proof calculus is described in Appendix A.1. Using extcnf_combined and ext_uni inferences results in shorter proofs since the details of intermediate inferences are elided. Unfortunately, using compound rules also loses information that can be very expensive to recompute. In this section we see how modifying LEO-II to produce more detailed proofs was preferable to recomputing the missing information during reconstruction.

Böhme and Weber (2011) discuss ideal qualities of proof output, to facilitate reconstructing and understanding proofs. The work described in this section improves LEO-II's proof output with regards to these qualities. In particular, we will be concerned with two problems described by Böhme and Weber. We will first deal with the problem of compound inferences:

> "Proofs should not contain surprises. The number of different inference rules in a proof format will likely depend on the automatic prover [...] But no matter what the number of rules is, each one should have simple and straightforward semantics. [...] Small and focused inference rules are to be preferred, and complex rules should be broken into several smaller ones." (Böhme and Weber, 2011)

To address this problem, LEO-II was modified to accept two command-line switches, `-nux` and `--expand_extuni`, to expand the compound inferences extcnf_combined and ext_uni into primitive inferences.

The rule extcnf_combined is used to summarise a chain of applications of clausification rules into a single inference. Since this is a normalising process, applied to a single clause, one could imagine emulating this inference by normalising both the premise and conclusion clauses, and then showing these to be identical.

This is not completely straightforward, and search might be required. We shall use Skolemisation to give an example of this problem. Consider the following to be an application of extcnf_combined. It takes a unit clause containing a negative unit literal F, in the context $\vec{x}$ of clause-level variables. It produces a clause containing several literals, in the context $\vec{x'}$. We pick out two literals in the conclusion: F_1 and F_2 that have polarities p_1 and p_2 respectively.

$$\frac{\vec{x}\left\{{}^{-}\boxed{F}\right\}}{\vec{x'}\left\{\mathbf{C}, {}^{p_1}\boxed{F_1[\mathsf{sK}_1\ \vec{y}]}, {}^{p_2}\boxed{F_2[\mathsf{sK}_2\ \vec{z}]}\right\}}$$

The terms $\mathsf{sK}_1\ \vec{y}$ and $\mathsf{sK}_2\ \vec{z}$ are *Skolem terms*, introduced by Skolemisation. Recall that the constant at the head of a Skolem term is called a *Skolem constant*. sK_1 and sK_2 are such constants. A fresh Skolem constant is introduced at each application of Skolemisation. The type of this constant depends on the context of clause-level variables. The Skolem term is applied to such variables when it is introduced—this is necessary for soundness (Dowek, 2009). Thus $\vec{y}$ and $\vec{z}$ indicate which were the clause-level variables at the time when sK_1 and sK_2 were introduced, respectively.

Now imagine we are reconstructing this extcnf_combined inference, starting from the singleton clause $\vec{x}\left\{{}^{-}\boxed{F}\right\}$. If we come to carry out a Skolemisation step, how do we know which Skolem constant to use? We can easily find out the difference in constants between the conclusion and premises clauses. But this difference in constants does not exclusively contain Skolem constants—some of these constants might be theory constants that were introduced by instantiating quantifiers. In the example above, we have indicated the Skolem constants sK_1 and sK_2, to simplify matters; let us assume that we only need to choose between these two constants. We cannot assume a fixed ordering in the application of these constants, since Leo-II might have selected a different sequence of literals to normalise. We could use the constants' types to discriminate between them; if all the constants have different types then we can easily discriminate which constant to use. If, however, there are constants with the same type, then we have a factorial number of choices (in the number of constants). Furthermore, the order of terms that are applied to a Skolem constant also needs to be determined—this will be discussed further below. How can we cut down the number of choices? Tracking the shape of a formula will not work in general, since this may change: variables may be instantiated, and the formula may be broken up during normalisation.

A simple solution is to have Leo-II expand the extcnf_combined inferences into the individual primitive inferences. This eliminates the need for search during reconstruction: we no longer need to discover the order in which Skolem constants are introduced. But we are not yet out of the woods. The proof could still lack the information needed for immediate translation. As observed by Böhme and Weber,

> "Some proof formats are little more than quick hacks: Prover developers tend to include information in [proofs] only if it can readily be extracted from the prover's internal data structures. This sometimes leads to proof formats that contain redundant information, which could easily be reconstructed in the checker. More often, however, and much more gravely, it leads to proofs that contain an insufficient amount of detail." (Böhme and Weber, 2011)

Consider the inference below from Leo-II's proof of PUZ107^5. The inference removes a subformula, but Leo-II also reorders the prefix of clause-level quantifications. In this inference, both the hypothesis and conclusion clauses consist of three literals. Note that each SV_i variable has type ι, and that the scope of each SV_i variable extends across an entire clause. In this example we depart from using the notation of Leo-II inferences, to emphasise the focus on reconstructing these inferences in Isabelle/HOL.

$$\frac{\forall SV_8\ SV_6\ SV_4\ SV_2\ SV_3\ SV_1 \left(\begin{array}{l}((SV_1 \neq SV_3 \vee SV_2 \neq SV_4) = \mathsf{False} \vee \\ \quad \mathsf{sK1_{SX0}}\ SV_1\ SV_2\ SV_6\ SV_8 = \mathsf{False}) \vee \\ \mathsf{sK1_{SX0}}\ SV_3\ SV_4\ SV_6\ SV_8 = \mathsf{False}\end{array}\right)}{\forall SV_4\ SV_8\ SV_6\ SV_2\ SV_3\ SV_1 \left(\begin{array}{l}((SV_1 \neq SV_3) = \mathsf{False} \vee \\ \quad \mathsf{sK1_{SX0}}\ SV_1\ SV_2\ SV_6\ SV_8 = \mathsf{False}) \vee \\ \mathsf{sK1_{SX0}}\ SV_3\ SV_4\ SV_6\ SV_8 = \mathsf{False}\end{array}\right)}$$

In Isabelle/HOL, we could start reconstructing this inference by first applying the $\forall$-introduction rule for six times, then applying $\forall$-elimination to instantiate the quantified variables in the hypothesis. To make use of existing automation within Isabelle, we need not specify how the quantified variables are to be instantiated. Instead we would rely on Isabelle's matcher to find suitable instantiations. But there are 6^6, or 46656, possible instantiations for this inference. It takes disproportionately long to explore this space, so a more feasible solution is needed. I settled for a pragmatic solution: inspecting the variable names, and matching like with like. That is, instantiating SV_8 to SV_8, SV_6 to SV_6, and so on. For completeness, there is also a failover tactic that explores the whole space of instantiations.

Rather than relying on Isabelle's matcher, a dedicated matcher was written to handle such cases, to avoid introducing *logical variables*. These are familiar from Prolog, and consist of special placeholder variables that can be instantiated by Isabelle using unification. There is a performance gain from the increased discrimination of the purpose-built matcher: while Isabelle's matcher would consider any symbols of suitable type to be possible candidates, the dedicated matcher will focus solely on suitable candidates—such as the clause-level quantified variables.

As mentioned earlier, we also need to determine the order of terms to which a Skolem constant is applied to yield a Skolem term. Referring to a Skolem term from an earlier example, in $\mathsf{sK}_1\ \vec{y}$ we need to determine $\vec{y}$. The arguments to Skolem terms consist of clause-level quantified variables, but, as remarked earlier, if they have the same type then it is not immediately clear in what order to place the arguments. Once again, we can inspect the variable names. For example, take this application of extcnf_forall_neg from Leo-II's proof of GEG007^1:

$$\frac{\forall SV_{13}\ SV_6 \left(\begin{array}{l}(\forall SX_2.\neg \mathsf{c}\ SX_2\ \mathsf{spain} \vee \\ \quad \mathsf{c}\ SX_2\ \mathsf{catalunya}) = \mathsf{False} \vee \\ \mathsf{a}\ SV_6\ SV_{13} = \mathsf{False}\end{array}\right)}{\forall SV_6\ SV_{13} \left(\begin{array}{l}(\neg\ \mathsf{c}\ (\mathsf{sK7_{SX2}}\ SV_{13}\ SV_6)\ \mathsf{spain} \vee \\ \quad \mathsf{c}\ (\mathsf{sK7_{SX2}}\ SV_{13}\ SV_6)\ \mathsf{catalunya}) = \mathsf{False} \vee \\ \mathsf{a}\ SV_6\ SV_{13} = \mathsf{False}\end{array}\right)}$$

We need to apply the Skolem constant $\mathsf{sK7_{SX2}}$ to the arguments SV_{13} and SV_6 in some order. SV_{13} and SV_6 have the same type—had this not been the case, it would be easy to order them. As before, trying all possible instantiations of the arguments given to the Skolem constant can take factorial time. Instead, one could look in the conclusion for Skolem terms, noting which Skolem constant is at the head, then looking at the names of its arguments in order. This order is then used when emulating the inference. In this case, we use the Skolem term $\mathsf{sK7_{SX2}}\ SV_{13}\ SV_6$.

The other compound rule in Leo-II's implementation is ext_uni, which abbreviates chains of inferences related to higher-order unification (Huet, 1975). In Leo-II, as in its predecessor LEO (Benzmüller, 1999; Benzmüller, 2005), unification forms part of the proof calculus

itself. As can be seen in Leo-II's calculus in Section A.1, the uni_triv and uni_dec rules are straightforward to emulate, but the uni_flex_rigid rule less so. Rather than reimplement Leo-II's unification engine, which is quite a complex function (Benzmüller and Sultana, 2013), and essentially duplicate the code, I opted to expand the ext_uni inference into its primitive steps, and emulate those instead.

As far as the existing proof calculus is concerned, the changes applied to Leo-II were restricted to expanding the compound inferences. For example, there are cases where the implementation does not seem to agree with the specification—for example, consider this application of this clause_copy inference, from Leo-II's proof of GEG007^1:

$$\frac{\forall XY.\ \mathsf{c}\,X\,Y \longrightarrow \mathsf{c}\,Y\,X}{(\forall XY.\ \mathsf{c}\,X\,Y \longrightarrow \mathsf{c}\,Y\,X) = \mathsf{True}}$$

Clearly, the inference does not only copy the clause, it also polarises its sole literal. Such behaviours were not changed in Leo-II: instead the reconstruction code was adapted to deal with them. The handling of the clause_copy inference is detailed in Section 5.5.1.1.

3.1.2 Strengthening the prover

The work described in this section does not specifically affect proof generation, as in the previous section. Rather, the purpose of this section is to describe improvements to Leo-II's proof-finding ability. Since considerable effort was devoted to building an automatic import facility of Leo-II proofs into Isabelle/HOL (described in Chapter 5), the usefulness of this facility is clearly increased if Leo-II's proof-finding ability is increased.

In earlier joint work with Jasmin Blanchette and Larry Paulson, various ATP systems were evaluated using Sledgehammer's problem-translation features (Sultana et al., 2012). During those experiments we compared how the provers responded when given a large number of facts. We also evaluated how they handled higher-order problems compared to their first-order encodings. The first-order encoding was produced by Sledgehammer using methods described by Blanchette (2012). During this earlier work some problems with Leo-II were observed:

- Leo-II's performance declined noticeably when it was given more input clauses, as can be seen in Figure 3.1. In that figure we count the number of *facts* (i.e., free-form formulas) rather than clauses, but the outcome would have been similar had we counted clauses. Leo-II's performance seemed to peak when it was given 40 facts.[1] We observed that

  > "Given that Leo-II is based on a highly optimised first-order prover and includes its own relevance filter, we could have expected it to scale better. We suspect that Leo-II's inefficient encoding of higher-order types in the untyped format supported by E is at cause."
  > (Sultana et al., 2012)

  Here the mentioned first-order prover is E, with which Leo-II cooperates to prove theorems. Leo-II's sensitivity to the number of clauses was surprising since Leo-II used term indexing and sharing to deal with formulas efficiently (Theiss and Benzmüller,

[1] Monomorphisation (instantiating Isabelle's polymorphic terms to form monomorphic terms) might have produced a 100 additional facts, thus the total number of facts might be 140.

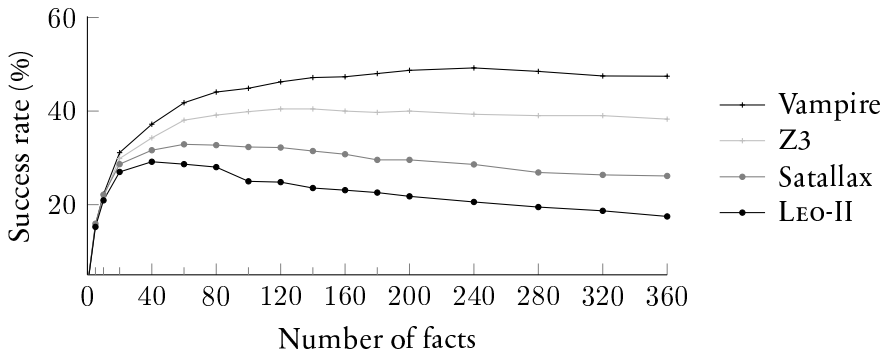


Figure 3.1: How different ATP systems coped when given more facts. Ideally, having more facts should make it easier to prove a theorem. In practice, it might overcome a prover's ability to store and process those facts. This graph is taken from previous experiments (Sultana et al., 2012), and was drawn with the help of Daniel Wand.

2006). It was later found that these features are currently underused (Benzmüller and Sultana, 2013): inference is not done using the efficient term representations, and indexing/sharing does not apply at the level of literals and clauses.

- Experiments showed that Leo-II appeared to perform better on first-order encodings (done by Sledgehammer) of higher-order problems, rather than on the original higher-order problems. Leo-II's translation to FOL was suspected to be the culprit. Reference was made above to "Leo-II's inefficient encoding of higher-order types"; this is elaborated further later in the article:

  > "Although Leo-II is based on E, it performs some reasoning of its own and relies on a [verbose] translation of types to first-order logic."
  > (Sultana et al., 2012)

  Examples of the encoding used by Leo-II, and its shortcomings, are discussed below.

Since it is central to Leo-II's operation, it was important to improve the first-order encoding used by Leo-II to communicate with E. We look at this in more detail next.

3.1.2.1 Improved interface with other provers

Parts of this section previously appeared in published work (Benzmüller and Sultana, 2013) co-authored with Christoph Benzmüller, but the underlying work and writing of this section are mostly mine.

Most of the proofs generated by Leo-II reveal its reliance on the first-order prover with which it collaborates—usually E (Schulz, 2002b). Leo-II attempts to reduce the conjecture and axioms into a set of first-order clauses which are then given to the first-order prover for consistency-checking. Figure 3.2 shows such a proof, in which E is given 11 facts. It is only rarely that Leo-II completes a proof without calling E; Figure 3.3 shows such an example.

Figure 3.3 is the first annotated example of a Leo-II proof so far. Let us step through the inferences made in the proof. We start with the conjecture, which is negated during the first step. This yields a clause containing a single negative literal. The second step unfolds definitions (via the inference unfold_def), and we can see that this has no effect. The polarity_switch inference simply negates a literal and flips its polarity. Then extcnf_combined

pushes the negation in, and Skolemises the formula.[2] Note the resulting Skolem term $\mathsf{sK1}_{\mathrm{V}}\ U$, where $\mathsf{sK1}_{\mathrm{V}}$ is a (fresh) Skolem constant. Then the clause_copy inference is applied; it has no logical effect. The extcnf_forall_pos inference then lifts the universal quantification of U, within the sole literal, to the clause level, and renames U to SV_1. Then the extcnf_not_pos inference simply removes an application of the negation constant at head position from a literal, and flips the literal's polarity. Finally the unification algorithm succeeds in unifying $SV_1(\mathsf{sK1}_{\mathrm{V}}\ U) = \mathsf{cA}$. The so-called *imitation* binding (Huet, 1975) is a possible unifier:

$$SV_1 \mapsto \lambda\_.\mathsf{cA}.$$

This results in a singleton clause containing the literal $(\mathsf{cA} = \mathsf{cA}) = \mathsf{False}$, which is clearly absurd. An application of Leo-II's uni_triv rule (described on page 140) to this clause yields the empty clause.

As acknowledged by Leo-II's authors, cooperation is indispensable to Leo-II's success (Benzmüller et al., 2008c, §3). This snippet is from an early paper on Leo-II, but still holds true today:

> "We have a particular interest in optimizing the reasoning in Leo-II towards quick transformation of essentially higher-order clauses into essentially first-order or propositional ones [...]. Then, [...] we want to cooperate with specialist reasoners for efficient refutation of these subsets of clauses. In summary, we are rather interested in the strengths of the cooperative approach than in isolated completeness of Leo-II and both aspects may even turn out to be in conflict with each other."
> (Benzmüller et al., 2007, §3)

Naturally, the number of clauses given to the first-order prover varies from problem to problem. For instance, in the proof of NUM668^1, Leo-II gives E a single fact to prove inconsistent:

$$\forall SV_1 : \mathsf{nat}.\ (\mathsf{pl}\ \mathsf{x}\ \mathsf{y}\ =\ \mathsf{pl}\ \mathsf{x}\ SV_1) = \mathsf{False}.$$

This is easy to refute: simply instantiate SV_1 to y, and we obtain an absurd statement. (In this case, Leo-II should try and refute this clause itself, rather than pay the performance penalty of calling E for such an easy refutation.) Many proofs involve sending E many more facts—for instance, in the proof of SEV058^5, E is given 393 clauses. By modern standards, this is not a large number of clauses to give to an ATP system. More mature ATP systems are fairly robust to larger number of clauses compared to newer prototypes; this can be seen in Figure 3.1.

E's indispensibility to Leo-II underscores the importance of the encoding used by Leo-II to communicate problems to E. During earlier work this encoding was found to have the following shortcomings (Sultana et al., 2012; Benzmüller and Sultana, 2013):

- As previously observed (Sultana et al., 2012), the old translation was verbose, and its use resulted in first-order formulas that were fairly large because of how type information was encoded. This verbosity caused additional overhead to ATP systems. It contributed to ATP systems missing their timeout to find a refutation. I worked on a re-implementation of the translation to FOL. The new translation is complete, in

[2] In this instance Leo-II was not run with the -nux parameter to expand extcnf_combined into primitive inferences, as described in Section 3.1.1.

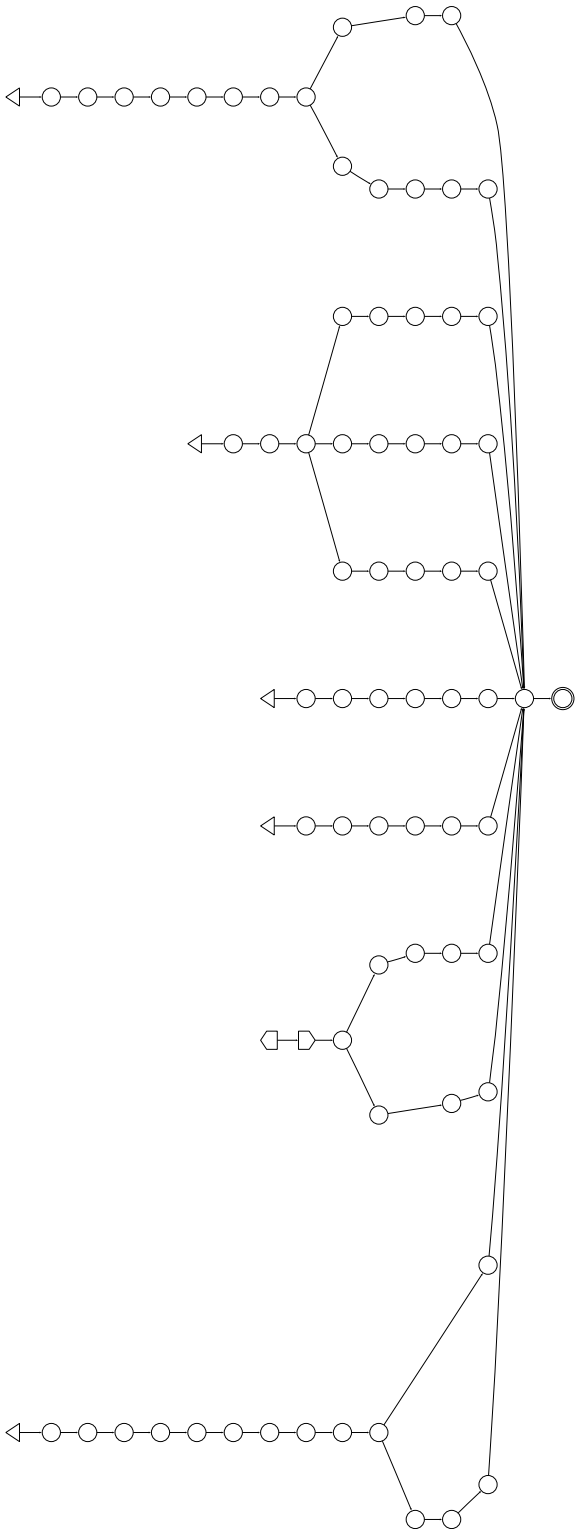


Figure 3.2: Graph for Leo-II's proof of NUM667^1. Formulas and inference names are not shown here since their precise values are not important. The graph is implicitly directed from left to right. The double-circled node on the right is the final clause—the derivation of falsity. The node connected to it is the contradiction derived by E from the 11 clauses to which it is connected. The diagram uses the presentation cues used by Trac et al. (2006) in their IDV system. The left-pointing pentagonal node is the conjecture, the right-pointing pentagon is the negated conjecture, triangles are axioms, and circular nodes are derived clauses.

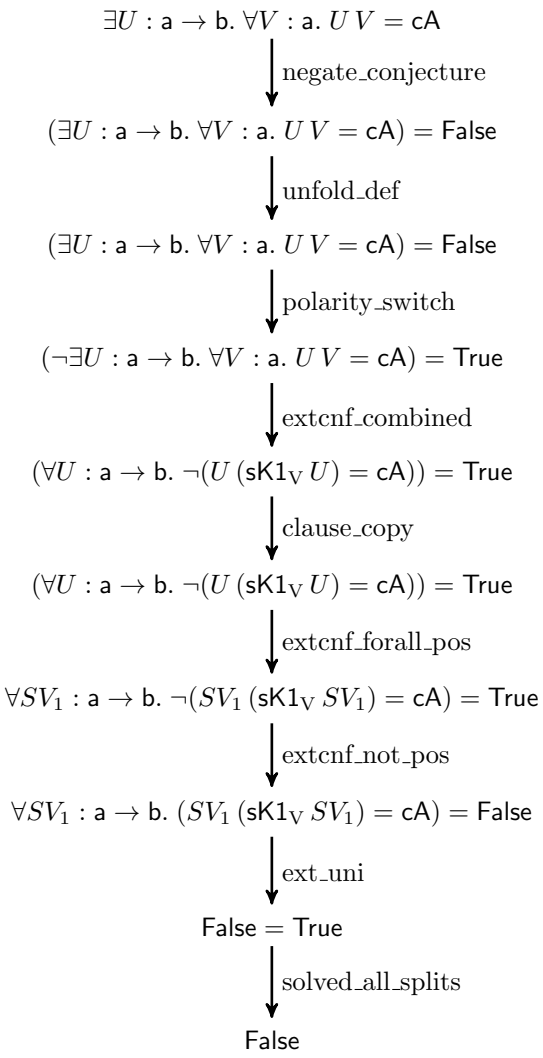


Figure 3.3: Graph for Leo-II's proof of SET045^5. This proof is notable for its lack of reliance on an external ATP system.

the sense that λ-terms are fully λ-lifted, and all required type information is encoded. Since the new translation tends to include more information than the old translation used in Leo-II, it may well be larger. To address this problem, a more compact encoding of type information was implemented, closely following Claessen et al. (2011) to implement their monotonicity analysis of types. A type is *monotonic* if its domain can be extended with new elements. Claessen et al. show that it is safe to erase the type information of monotonic types, leading to smaller encodings of λ-terms, and describe a method to check whether a type is monotonic. Recently Blanchette and Popescu (2013) formalised and verified the correctness proofs of Claessen et al. (2011) using Isabelle/HOL.

- The old translation was incomplete, since it did not properly encode λ-terms, nor did it encode term-level *proxies* of logical constants. This is described in more detail next.

Using the new translation, when Leo-II is asked to prove

```
thf(goal, conjecture, ((=) = (=))).
```

Leo-II produces the clause `~($true)`, probably by using the uni_triv rule on the negated conjecture, and gives it to E to refute. (Unfortunately Leo-II does not recognise that its work is complete.) Using the old translation, it would send E the single clause:

```
fof(7,axiom,((~ leoLit(leoTi(true,o))))).
```

Here, the constant symbol `true` is a *proxy*: a term-level constant which should have the same semantics as a (formula-level) logical constant. Proxies are used since in FOL we must maintain a distinction between formulas and terms. Proxies help us get around this distinction for terms with propositional semantics. Unfortunately, in this case Leo-II did not include an axiom to give semantics to `true`. It should have included an axiom such as the following:

```
fof(true, axiom, leoLit(leoTi(true,o))).
```

Had such an axiom been included, the FOL ATP system would have been able to find a refutation.

The other special constants seen above will be described next. In both the old and new translations used by Leo-II, `leoTi` is used to assign types to terms—here it is saying that the term `true` is of type `o`, where 'o' itself is a constant in the language. This constant is special to the translation: it is used to encode the Boolean type. Thus, the translation encodes types as first-order terms. This kind of type encoding (using a constant such as `leoTi`) is called *tagging* in Sledgehammer (Blanchette, 2012), and is credited by Meng and Paulson (2008) to Hurd (2003). The constant `leoLit` is used to lift propositional terms (that is, terms that are typed 'o') into formulas.

Given the same THF problem, the new translation sends the following output to the ATP system:

```
fof(7, axiom, ~($true)).
```

That is, the translation notices that instead of using `leoLit` to encode True, it can directly use the FOF constant with that denotation. When it becomes necessary to use proxy terms, such as `true`, then the semantics of each proxy are specified using additional axioms, such as `fof(prox_true1, plain, ($true <=> leoLit(leoTi(true, o))))`.

The old translation was also incomplete in a different sense, since it did not fully translate λ-terms into first-order form. The previous translation simply created fresh constants for λ-terms, and did not characterise these constants further. For example, while trying to prove the THF problem

```
thf(conj_0,conjecture,(
    ? [F: $o > $o] :
    ! [P: $o > $o,Q: $o] :
      ( ~ ( P @ ( => @ ( F @ $true ) @ Q ) )
      | ~ ( F @ ( => @ Q @ ( F @ $false ) ) )
      | ( F @ Q ) ) )).
```

the output of the previous translation includes axioms such as

```
1 fof(44,axiom,(
2     ~ leoLit(leoTi(leoAt(leoTi(sK2_SY3,leoFt(leoFt(o,o),o)),
3                          leoTi(abstrSX0SX0,leoFt(o,o))),o)) )).
```

The encoded type `leoFt(o,o)` on line 3 indicates that the constant `abstrSX0SX0` is of type $o \to o$. The name `abstrSX0SX0` is derived from serialising the term $\lambda SX_0.SX_0$, but no further definition of this constant is given by the translation. The new translation carries out full λ-lifting (Hughes, 1982) on such terms. This involves creating fresh custom combinators (called *super-combinators*) to eliminate abstractions. In this example, λ-lifting yields a pair of axioms, one of which characterises the newly-introduced constant `ll1` as needed.

```
fof(ll1,axiom,(
    ! [SX0] :
      ( leoLit(leoTi(leoAt(leoTi(ll1,leoFt(o,o)),
                           leoTi(SX0,o)),o))
    <=> leoLit(leoTi(SX0,o)) ) )).

fof(44,axiom,(
    ~ leoLit(leoTi(leoAt(leoTi(csK2_SY3,leoFt(leoFt(o,o),o)),
                         leoTi(ll1,leoFt(o,o))),o)) )).
```

λ-lifting was developed for compiling functional programs, but has also been used in translations of HOL to FOL (Meng and Paulson, 2008).

Evaluation. These improvements were implemented in Leo-II version 1.5. Table 3.1 shows evaluation results related to this work (Benzmüller and Sultana, 2013). These results suggest that the performance improvement gained from improving the translation to FOL is small (at least on this benchmark). The translation forms the last stage of an inference process, and occurs before handing off to an external ATP system, therefore it is vulnerable to any upstream inefficiencies.

In Table 3.1, the `fully-typed` translation refers to Leo-II's existing translation. The `fof_full` translation is the improved (complete) translation without erasure of monotonic types (Claessen et al., 2011). Finally, `fof_experiment` is the complete translation which involves erasure of monotonic types. The experiments used E version 1.6 as the backend ATP system, and were run on a 2GHz AMD Opteron with 4GB RAM.

3.2 Satallax

Satallax did rather well in the evaluation mentioned earlier (Sultana et al., 2012), but the additional proved theorems surpassed Isabelle's ability to reconstruct them:

> "Satallax's unique contribution of 0.6%, or 8 goals, puts it in respectable company with E (0.8%), SPASS (0.6%), and Vampire (0.8%) and raises the overall success rate from 71.8% to 72.4%. The goals uniquely solved by Satallax all involve reasoning about λs. [...] Unfortunately, reconstruction tends to fail precisely for the goals that are solved by only Satallax, leaving the user with a short list of facts but no actual proof."
> (Sultana et al., 2012)

This section describes a possible solution to this problem. It consists of an extension of Satallax to produce proofs that can be directly checked and reused in Isabelle. In contrast, the reconstruction method used in earlier work (Sultana et al., 2012) involved attempting to re-find Satallax's proof using first-order provers.

Before looking at Satallax's internal proof representation, and how it can be mapped to a form that Isabelle can check, let us start by looking at how Satallax builds a proof for an example problem.

Satallax tests the satisfiability of a set of higher-order formulas by computing propositional approximations. Each higher-order formula in the initial set is then mapped to a propositional unit clause, and a SAT solver—usually MiniSat (Eén and Sörensson, 2003)—is used to test the satisfiability of this set. This initial set forms the initial branch in Satallax's tableau proof. Borrowing an example given by Brown (2013), let us see how Satallax would initialise the tableau. Let x be a Boolean constant. Let the input set of formulas, which we want to check for consistency, be $\{\mathsf{x},\ \forall Y.\, \mathsf{x} \longrightarrow Y\}$. Let $\ulcorner\cdot\urcorner$ be a bijective mapping from higher-order formulas to propositional letters. This mapping will be used to produce propositional encodings of tableau branches. The initial branch would be encoded as a set of two unit clauses:

$1:\ \{\ulcorner \mathsf{x} \urcorner\}$

$2:\ \{\ulcorner \forall Y : o.\, \mathsf{x} \longrightarrow Y \urcorner\}\,.$

This set of propositional formulas can be satisfied, but it can also be extended further by using the rules making up Satallax's calculus. We will see how to do this next.

SZS Status	`fully-typed`	`fof_full`	`fof_experiment`
Thm	64.8	64.9	65.3
All	60.9	61	61.3

Table 3.1: Comparing FOL encodings in Leo-II 1.5 (30s timeout). The first row of the table shows how many THF theorems could be proved using each encoding. The difference between `fully-typed` (64.8%) and `fof_experiment` (65.3%) consists of 13 problems. Using the lighter encoding clearly seems useful. Further tests (with longer timeout) and inspection of the translated problem could give a better idea of performance. The second row shows how Leo-II's verdict (be it that the problem is a theorem, satisfiable, countersatisfiable, etc), matches the problem's SZS classification.

If the propositional encoding is unsatisfiable, then the original set of higher-order formulas must be unsatisfiable. Alternatively, if a satisfying assignment is found, then Satallax uses its tableau calculus to extend open branches. The rules used to extend the open branches are themselves abstracted into non-unit propositional clauses, and unified with the earlier set of propositional clauses. Continuing the example used earlier, Satallax will instantiate the universal quantifier by applying its *specialisation* rule. Satallax maintains a set of terms, called the *term universe*. Specialisation involves producing every instance (of the appropriate type) using that universe. Since Y has type o, and initially the term universe for o is $\{\mathsf{True}, \mathsf{False}\}$, this means that $\forall Y : o.\ \mathsf{x} \longrightarrow Y$ must be instantiated to both $\mathsf{x} \longrightarrow \mathsf{True}$ and $\mathsf{x} \longrightarrow \mathsf{False}$. Satallax propositionally encodes the application of the specialisation rule by adding two clauses to the propositional abstraction of the problem. Note that, unlike the initial branch, the clauses are not unit. As is standard, a bar drawn above a literal indicates the negation of that literal.

$3 : \{\overline{\ulcorner \forall Y.\ \mathsf{x} \longrightarrow Y \urcorner}, \ulcorner \mathsf{x} \longrightarrow \mathsf{True} \urcorner\}$

$4 : \{\overline{\ulcorner \forall Y.\ \mathsf{x} \longrightarrow Y \urcorner}, \ulcorner \mathsf{x} \longrightarrow \mathsf{False} \urcorner\}.$

Satallax identifies $\ulcorner \mathsf{x} \longrightarrow \mathsf{False} \urcorner$ with $\overline{\ulcorner \mathsf{x} \urcorner}$, therefore clause 4 could be written as

$4 : \{\overline{\ulcorner \forall Y.\ \mathsf{x} \longrightarrow Y \urcorner}, \overline{\ulcorner \mathsf{x} \urcorner}\}.$

After Satallax has encoded the application of one of its tableau rules, the SAT solver is run again on the clause set. A branch is closed when the SAT solver finds the propositional abstractions of the higher-order formulas to be unsatisfiable. If a branch cannot be closed, and can be extended further, then the branch is extended by encoding a rule application as we have seen. If a branch cannot be closed, and cannot be extended further, then Satallax has found a satisfying assignment to the original formulas. Continuing the example, the SAT solver resolves clauses 2 and 4, yielding

$5 : \{\overline{\ulcorner \mathsf{x} \urcorner}\}$

which is then found to be inconsistent with clause 1. Thus Satallax closes the branch. There are no other branches in this example, therefore there is no more work to be done: Satallax has found the initial set of formulas to be inconsistent.

Satallax can output TPTP proofs describing the refutations it finds. In later chapters we describe the interpretation of Leo-II's TPTP proofs in Isabelle/HOL. We could also try to interpret the TPTP proofs produced by Satallax, but we have an opportunity to try a different approach: modifying Satallax's proof generation code to produce *proof scripts* to be checked directly by Isabelle. That is, instead of *importing* a TPTP proof, produced by Satallax, into Isabelle/HOL, we would *export* a proof script from Satallax in Isabelle's syntax. This approach involves studying the work of Teucke (2011), who implemented the part of Satallax that generates *refutation values*. Satallax's TPTP and Coq output are produced from a refutation value; this is described further below. We will also see how Satallax was modified to produce proofs encoded in Isar (Wenzel, 2002) syntax. Isar facilitates the writing of forward-style proofs, which fits the existing proof-generation code in Satallax.

3.2.1 Teucke's translation

Teucke's method starts by reconstructing the refutation found by Satallax. It uses three pieces of information: the initial set of clauses that Satallax sought to refute, the UNSAT core, and

the term universe. The *term universe* consists of the finite set of terms that Satallax used in instantiations during proof search. The *UNSAT core* consists of a set of inconsistent formulas, forming a subset of the original set of formulas given to Satallax. If available, the PicoMUS tool is used to process the UNSAT core to compute a *minimal* UNSAT core. PicoMUS is distributed with PicoSAT (Biere, 2008).

Using the UNSAT core produced by Satallax, and possibly minimised using PicoMUS, Teucke's method builds a tableau proof in a custom *finite* calculus. The calculus is finite in the sense that not the full inductively-generated set of inferences are required. This means that the calculus is not generally complete, but it is complete enough to refute the negated conjecture. Since the calculus is finite, we are assured that re-finding the proof is a terminating process.

The reconstructed proof is encoded as a value of a datatype called *refutation*. The definition of this datatype follows that of a tableau calculus. Part this definition is shown below, written in OCaml.

```
1  type refutation =
2   (* Conflict(m1,m2), where [m1] = [~m2] *)
3     Conflict of trm * trm
4   [...]
5   (* Implication(h,m1,m2,R1,R2), where [h]=s->t, [~m1]=s, [m2]=t *)
6   | Implication of trm * trm * trm * refutation * refutation
7   [...]
8   (* All(h,s,R,a,n,t), where [h] = forall_a.p , s = [pt]  *)
9   | All of trm * trm * refutation * stp * trm *trm
10  [...]
```

Using this datatype, a *conflict* inference (line 3) can be encoded when we have previously derived a formula and its negation; an *implication* inference (line 6) results in a fork of the tableau, with the negated antecedent on the first branch and the consequent on the second branch; and an *instantiation* inference (line 9) results in an extension of the current branch with the instance formula, as will be shown below.

The refutation value for the example used earlier on page 50, is

$$\texttt{All}(\overbrace{\forall Y.\, \mathsf{x} \longrightarrow Y}^{1},\ \overbrace{\mathsf{x} \longrightarrow \mathsf{False}}^{2},\ \overbrace{\texttt{Conflict}(\mathsf{x}, \mathsf{x} \longrightarrow \mathsf{False})}^{3},\ \_,\ \_,\ \overbrace{\mathsf{False}}^{4}).$$

Unimportant values are replaced with underscores. The numbered parameters above are described next:

1. The rule's premise: the formula to which the rule is applied.

2. The rule's conclusion: the instance formula.

3. The continuation of the refutation. In this case, this consists of noticing the contradiction between two formulas in the branch.

4. The term used to instantiate the universal formula.

3.2.2 Generation of proof scripts

Once a refutation value has been obtained, it is pretty-printed into a target format: either a Coq or TPTP script. Satallax was extended to print out refutations as Isar scripts. This modified version of Satallax has been made available online.[3]

It was not difficult to retarget an implementation of Teucke's method to produce proofs in a new logic, but it was not easy to retarget it to produce proofs that can be processed fully automatically. That is, if the target language is not sufficiently expressive then we might have to rely on the user modifying the proof to allow it to be checked. Coq's $\mathcal{L}_{tac}$ macros are more expressive than Isabelle's Isar syntax since $\mathcal{L}_{tac}$ allows expressing simple combinations of tactics.

If the resulting proof cannot be immediately checked—as is the case sometimes with our translation into Isar—then the proof modifications the user might need to make are usually minor. For example:

- Choosing among alternative tactics. This can easily be specified in an $\mathcal{L}_{tac}$ macro, but cannot be expressed directly in Isar.
- Replacing the blast tactic with the more powerful auto tactic.
- Carrying out small term transformations based on equivalences, such as $(F \longrightarrow \mathsf{False}) = \neg F$.

Examples of the need of such modifications can be seen in the Isar proof output from Satallax for problem PUZ081^1.

There are two solutions to this:

- If it exists, one could use a language similar to Coq's based on $\mathcal{L}_{tac}$. Very recently, one such language has been developed for Isabelle (Matichuk et al., 2014).
- Write a driver, using lower-level tactic programming in ML, to automate the kinds of proof-modifications needed by automatically choosing among alternative tactics, or attempting term-transformation tactics.

The Isar proof for the example from the previous section is shown in Figure 3.4. In that script, line 1 simply names the current Isabelle session, and line 2 refers to a theory containing Satallax-related definitions, such as inference rules used by Satallax and validated in Isabelle/HOL. We will revisit this further below. Line 4 specifies the signature, which consists of the constant x having Boolean type. Lines 6-9 formulate the theorem, and names the formulas involved. The proof starts at line 10, and is carried out by contradiction. The conjecture, in this problem, is the formula labelled `claim`, asserting False. The negation of this conjecture, labelled `H0` in the proof, cannot help us derive a contradiction! So the remainder of the proof focuses on showing the two assumptions (labelled `a1` and `a2`) to be inconsistent. The resulting proof is generated from the refutation value shown above: the intermediate formula labelled `H1` consists of specialising `a2` by instantiating its quantified variable to False. This is expressed as `a2[THEN spec, of "False"]`. *THEN* and *of* are theorem transformers (Nipkow et al., 2002, §5.15). The combinator *THEN* applies a rule to the goal—in this case, the rule is *spec*, or specialisation. The *of* combinator is then used to indicate the precise term to instantiate with. Thus the value of `H1` is $\mathsf{x} \longrightarrow \mathsf{False}$. As in

[3] `http://www.github.com/niksu/satallax2.7-isar/`

```
1  theory SatallaxProblem
2  imports Satallax
3  begin
4  consts x :: "o"
5
6  lemma
7  assumes a1 : "x"
8  assumes a2 : "(! X1::o. (x --> X1))"
9  shows claim : "False"
10 proof (rule ccontr)
11   assume H0 : "(~(False))"
12   show False
13     proof (rule ccontr)
14     note H1 = a2[THEN spec, of "False"]
15     from a1 H1 show ?thesis by blast
16     qed
17   qed
18 end
```

Figure 3.4: Isar script based on the refutation example from page 50. The refutation value from which this proof was produced was given on page 52.

the refutation value seen earlier, formulas `a1` (which simply asserted `x`) and `H1` are found to be inconsistent. The *blast* tactic is used to glue together inferences, or to finish off easy refutations. This tactic is described in Section 6.3.

We now briefly review how Satallax produces Coq output, before showing how this was adapted to produce Isar scripts. The Coq output relies on a formalisation of HOL in Coq, and a shallowly-embedded tableau calculus. $\mathcal{L}_{tac}$ macros (Delahaye, 2000) are used to encode tactic macros. These are invoked in the output Coq script.

Figure 3.5 shows the first part of a Coq script generated by Satallax for the TPTP problem NUM663^1. Lines 1-5 specify the signature, consisting of the type `nat`, the constants `x`, `y`, and `z` (all typed `nat`), and the constant `less`, denoting a binary relation over `nat`. Lines 6-11 formulate the theorem, using the constants declared earlier. The formulas making up the premises and theorem are named, to enable referring to them later in the proof. The proof starts at line 15. As with any tableau proof, it is a refutation proof. It largely consists of a chain of macro applications—the `tab_*` tokens are $\mathcal{L}_{tac}$ macros. It is a *forward-style* proof: new and intermediate facts are derived from existing facts, and are used to derive further facts. This is in contrast to the backward-chaining approach described in Section 2.6.3. The newly-generated facts are named according to the last argument given to each macro: for instance, "`tab_start H0.`" kicks off the proof by asserting the negated conjecture, and names this fact H0. The generation of facts is threaded through the proof. For instance, line 16 instantiates the `eq_sym` lemma (which states that equality is a symmetric relation) for the type `nat`, naming this new fact H1. Lines 17 and 18 then successively instantiate the universal quantifiers in H1 using the constants `x` and `y` respectively, resulting in the intermediate fact H2, and finally in the fact H3. Despite not being stated explicitly in the proof, we can work out that fact H3 encodes the formula $\mathsf{x} = \mathsf{y} \longrightarrow \mathsf{y} = \mathsf{x}$.

The generation of Isar output follows the approach used to generate Coq and TPTP

```
Variable nat:SType.
Variable x : nat.
Variable y : nat.
Variable z : nat.
Variable less : nat --> nat --> o.
Hypothesis l : ~ less x y -> x = y.
Hypothesis k : less y z.
Hypothesis et : forall (Xa:o), ~ (~ Xa) -> Xa.
Hypothesis satz15 : forall (Xx:nat) (Xy:nat) (Xz:nat),
  less Xx Xy -> less Xy Xz -> less Xx Xz.
Theorem satz16a : less x z.
% SZS output start Proof
% Coq Proof Script

tab_start H0.
set (H1 := (@eq_sym nat)).
tab_all H1 (x) H2.
tab_all H2 (y) H3.
tab_all satz15 (x) H4.
tab_all H4 (y) H5.
tab_all H5 (z) H6.
tab_imp l H7.
 tab_imp H6 H8.
  tab_conflict H8 H7.
[...]
```

Figure 3.5: Part of a Coq script produced by Satallax for NUM663^1

```
typedecl nat
consts x :: "nat"
consts y :: "nat"
consts z :: "nat"
consts less :: "(nat=>(nat=>o))"

lemma
assumes l : "((~(((less x) y))) --> (x = y))"
assumes k : "((less y) z)"
assumes et : "(! X1::o. ((~((~(X1)))) --> X1))"
assumes satz15 : "(! X1::nat. (! X2::nat. (! X3::nat.
  (((less X1) X2) --> (((less X2) X3) --> ((less X1) X3))))))"
shows satz16a : "((less x) z)"
(*% SZS output start Proof*)
(*% Isar Proof Script*)

proof (rule ccontr)
  assume H0 : "(~(((less x) z)))"
  show False
    proof (rule ccontr)
    note H1 = eq_sym[where 'ty = nat]
    note H2 = H1[THEN spec, of "x"]
    note H3 = H2[THEN spec, of "y"]
    note H4 = satz15[THEN spec, of "x"]
    note H5 = H4[THEN spec, of "y"]
    note H6 = H5[THEN spec, of "z"]
    note H7 = l[THEN TImp[THEN mp]](*tab_imp*)
    from H7 have False
      proof
        assume H7 : "(~((~(((less x) y)))))"
        note H8 = H6[THEN TImp[THEN mp]](*tab_imp*)
        from H8 have False
          proof
            assume H8 : "(~(((less x) y)))"
            from H8 H7 show ?thesis by blast
[...]
```

Figure 3.6: Part of an Isar script produced by Satallax for NUM663^1

output: the refutation value is pretty-printed into an Isar script. Figure 3.6 shows the first part of an Isar script generated by Satallax for the TPTP problem NUM663^1. Perhaps the legibility of the output proof might be improved using methods such as those described by Smolka and Blanchette (2013).

There is a clear correspondence between segments of the proof in Figure 3.5 and others in Figure 3.6. This is unsurprising, since both were generated from the same refutation value. For instance, lines 21-23 in the Isar proof shown in Figure 3.6 are the analogues of lines 16-18 described earlier from Figure 3.5.

Instead of macros, the Isar output uses theorem transformers such as *THEN* and *of*, and the *blast* tactic to carry out some proof search, described earlier. It also uses a formalisation of the axioms used by Satallax, such as eq_sym and TImp—these will be described further below. Note that these are not regarded as being axioms in Isabelle/HOL. Rather, they have been derived in Isabelle/HOL, and contained in the Satallax theory for Isabelle/HOL. This will be explained further below, in the context of the proof we just saw.

For example, on line 21 we find the expression

```
note H1 = eq_sym[where 'ty = nat]
```

which derives a fact from the pre-existing fact eq_sym, by specialising `'ty` to `nat`. The fact eq_sym is formalised as follows, and `'ty` is simply the schematic type of the values concerned:

$$\forall x,\, y.\, (x :: {'ty}) = y \longrightarrow y = x.$$

Clearly, the fact eq_sym is valid in Isabelle/HOL. Similarly, on line line 27 we find the expression `note H7 = l[THEN TImp[THEN mp]]` which takes the fact labelled `l`, and applies to it the rule `TImp[THEN mp]`. This rule is itself a derived formula. The name `mp` refers to modus ponens, while TImp refers to the following formula:

$$(X \longrightarrow Y) \longrightarrow \neg X \vee Y.$$

This too is valid in Isabelle/HOL. The value of `TImp[THEN mp]` is

$$(X \longrightarrow Y) \Longrightarrow \neg X \vee Y.$$

That is, the HOL implication was lifted to an inference rule. Finally therefore, the value of the expression `l[THEN TImp[THEN mp]]` is

$$\neg(\mathsf{less}\ \mathsf{x}\ \mathsf{y}) \ \vee\ \mathsf{x} = \mathsf{y}.$$

As in the Coq output, the formulas corresponding to intermediate facts are not always clear from the Isar script, but they are displayed by the proof assistant when the script is checked.

3.2.3 Evaluation

The previous section described an Isar proof generator implemented in Satallax. The generator relies on an embedding of Satallax's inference rules in Isabelle/HOL, which assures soundness by checking each generated Isar proof.

In principle, the proof generation is complete: that is, every Satallax refutation should result in an Isar proof. In practice, the implementation has syntactic shortcomings that impede the fully automatic checking of proofs, as described in the previous section. This

problem can be overcome by using a more flexible target language: either a macro language, or a more low-level programmable interface.

To obtain an idea of the effectiveness of the Isar-producing modification to Satallax, I ran it on all 3172 THF problems in TPTP version 6.0.0, using a 10-second timeout.[4] Satallax generated refutation proofs for 53% of these problems. Isabelle/HOL could check 77.3% of these refutations fully automatically. This value should increase if either of the improvements described in the previous section are made.

Conclusion

This chapter described the arguments for and against modifying an ATP system. Modifying Leo-II to expand compound inferences allowed us to make performance gains during reconstruction. We also saw that this alone was insufficient, since there were inferences (in the expanded calculus) which could take exponential time to reconstruct. A purpose-built matcher, sensitive to variable names, was used to keep the time complexity linear. Finally, we also looked at proof generation in Satallax, and how this was modified to produce Isar scripts, which can be checked using Isabelle/HOL. In the next chapter we will look at a proof transformation that can be used to embed refutation proofs produced by an ATP system into Isabelle/HOL.

[4]The code and data used in this experiment are available online at `http://www.cl.cam.ac.uk/~ns441/files/satallax_isar_eval.tgz`

Chapter 4

Proof embeddings

This chapter describes two approaches to transform logical inferences. These transformations on inferences can be lifted to the level of proofs, yielding proof translations. These translations will be used on proofs encoded in a consistency-preserving calculus to produce proofs encoded in a validity-preserving calculus. The proof translations are conceptually simple, but they work for a broad class of consistency-preserving calculi.

4.1 Proof by contradiction

Proof by contradiction is a popular proof method in mathematics. It involves refuting the negation of the conjecture. That is, if P is the statement being conjectured, then a proof by contradiction involves proving $\neg\neg P$. An argument by contradiction is only acceptable if the logic validates

$$\neg\neg P \longrightarrow P \tag{4.1}$$

for that specific P. This proof method is often used in classical logics, since such logics validate formula 4.1 for all propositions.

Proof systems whose main proof method is proof by contradiction are called *refutation* systems. They work by refuting the negated conjecture. Refutation involves concluding an absurd statement—one which denotes falsity, such as the atom False. Such a statement is derivable when a proposition is shown to be both true and false—a contradiction.

Most ATP systems implement some form of refutation calculus. These calculi are usually reductive, in the sense that they work by iteratively simplifying a collection of formulas. This iteration has an important invariant: the satisfiability (or consistence) of these formulas is preserved across iterations. This means that a set of formulas at iteration $n \in \mathbb{N}$ is inconsistent if, and only if, the set of formulas at iteration $0 \leq m < n$ is inconsistent, for all m. Because of this property, checking a formula for validity can be reduced to checking its negation for satisfiability: if the negated conjecture is false, then it has no models, which must mean that its negation (the original conjecture) is satisfied by all models. Thus a conjecture is determined to be a theorem.

A calculus is said to be *refutation complete* if it can be used to falsify any false statement. In classical logic, a refutation complete calculus qualifies as being *complete*, by virtue of formula 4.1. That is, if the negated conjecture $\neg P$ is found to be inconsistent with the theory axioms, then we have proved that $\neg P \longrightarrow \mathsf{False}$ is valid—which is logically equivalent to $\neg\neg P$. In classical logic, $\neg\neg P$ is logically equivalent to P.

4.2 Consistency-preserving rules

This section recounts known results, and it is intended to provide context to the original work described in the sections that follow. In this section we will look at examples of rules that occur in a refutation calculus. Specifically, we will look at rules for clausification, Skolemisation and splitting. The soundness criterion of such rules is *consistency-preservation*: if there exists a model that satisfies a rule's premises, then there also exists a model that satisfies its conclusion. We will implicitly overload this term to mean that *inconsistency* is also preserved by such inferences. That is, if there does not exist a model that satisfies the premises, then there does not exist a model that satisfies the conclusion.

Where useful, these rules will be contrasted with *validity-preserving* rules. These have a different soundness criterion: if a rule's premises are satisfied by all models, then its conclusion is satisfied by all models.

4.2.1 Tseitin clausification

Many ATP systems do not operate directly on free-form logical statements. Instead, they first bring free-form statements into some kind of normal form. (Preserving the normalised status of formulas should therefore also be an invariant of an ATP system's main loop.)

One such normal form is *conjunction normal form* (CNF). In CNF, formulas consist of conjunctions of disjunctions of literals—where literals consist of either atoms or negated atoms. Quantifiers could be *prenexed* (pulled outside to leave a quantifier-free formula body), or *miniscoped* (pushed in to lessen the quantifiers' scope). A prenex-based clausification was described by Bundy (1983), while an approach based on miniscoping was described by Quaife (1992).

We can bring arbitrary formulas into CNF by using logical identities. This process preserves validity, but in the worst-case it is exponential in space. Since the ATP system works by consistency-preservation anyway, why use a validity-preserving preprocessing step to normalise formulas? Tseitin (1968) described a consistency-preserving transformation into CNF that has polynomial time complexity. It is sufficient for an ATP system to use this procedure as a preprocessing step without compromising its soundness and completeness.

Essentially, Tseitin's procedure abbreviates formulas by using definitions. For example, given a formula $\mathcal{T}_1 \wedge \mathcal{T}_2$, where $\mathcal{T}_i$ are arbitrary subformulas, Tseitin's transformation will produce names, unless this has been done already, for $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_1 \wedge \mathcal{T}_2$. Let us use the quoted symbols $\ulcorner \mathcal{T}_1 \urcorner$, $\ulcorner \mathcal{T}_2 \urcorner$, and $\ulcorner \mathcal{T}_1 \wedge \mathcal{T}_2 \urcorner$ as names for $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_1 \wedge \mathcal{T}_2$ respectively. The names are then related via a Boolean equation defining $\ulcorner \mathcal{T}_1 \wedge \mathcal{T}_2 \urcorner$ in terms of $\ulcorner \mathcal{T}_1 \urcorner$ and $\ulcorner \mathcal{T}_2 \urcorner$:

$$\ulcorner \mathcal{T}_1 \wedge \mathcal{T}_2 \urcorner = \ulcorner \mathcal{T}_1 \urcorner \wedge \ulcorner \mathcal{T}_2 \urcorner.$$

This equation is then conjoined with the recursive Tseitin clausification of $\mathcal{T}_1$ and $\mathcal{T}_2$. The resulting conjoined equations are then expanded into conjunctions (of disjunctions of literals), yielding a CNF formula that is equisatisfiable with $\mathcal{T}_1 \wedge \mathcal{T}_2$. A concrete example of this will be given below.

Tseitin's clausification can be described in terms of rules as follows. Let R be the set of (sub)formulas to be clausified. Initially R could be a singleton set, containing the formula that is to be transformed into CNF. This formula usually consists of the negated conjecture. R may also contain axioms related to the theory within which we are reasoning. Let T be

the set of subformulas transformed so far. Initially, T is the singleton set containing only the fresh name for the full formula, asserting it positively.

We will now look at how negation is handled by Tseitin's method. We will use a notation that looks like (and in some respect it does constitute) logical inference, but that mainly conveys the operational meaning of Tseitin's method. The method iterates over a state consisting of two sets, T and R, which we write "$\mathsf{T} \mid R$". At each iteration, the method processes the contents of R, and adds clauses to T. We use a dashed inference line to emphasise that the method preserves satisfiability, not validity.

$$\frac{\mathsf{T} \mid \{\neg A\} \cup R}{\mathsf{T} \cup \left\{ \begin{array}{c} (\neg\ulcorner\neg A\urcorner \vee \neg\ulcorner A\urcorner), \\ (\ulcorner\neg A\urcorner \vee \ulcorner A\urcorner) \end{array} \right\} \mid \{A\} \cup R}$$

This rule expresses that the negated formula $\neg A$ is named $\ulcorner\neg A\urcorner$ (unless it has been named already at an earlier occurrence in the formula), a Boolean equation is added to T, and $\neg A$ is replaced with A in R. Note that the set

$$\{(\neg\ulcorner\neg A\urcorner \vee \neg\ulcorner A\urcorner)\,, (\ulcorner\neg A\urcorner \vee \ulcorner A\urcorner)\}$$

is logically interpreted to be equivalent to $\ulcorner\neg A\urcorner = \neg\ulcorner A\urcorner$, the Boolean equation relating the names of the formulas involved.

An invariant of this process is that all formulas in T consist of disjunctions of literals. The formulas of T are implicitly conjoined together. This invariant guarantees that when R is empty at the end of the procedure, T can be interpreted to be the CNF of the initial formula.

Conjunction is handled as follows:

$$\frac{\mathsf{T} \mid \{A \wedge B\} \cup R}{\mathsf{T} \cup \left\{ \begin{array}{c} (\ulcorner A \wedge B\urcorner \vee \neg\ulcorner A\urcorner \vee \neg\ulcorner B\urcorner), \\ (\neg\ulcorner A \wedge B\urcorner \vee \ulcorner A\urcorner), \\ (\neg\ulcorner A \wedge B\urcorner \vee \ulcorner B\urcorner) \end{array} \right\} \mid \{A, B\} \cup R}$$

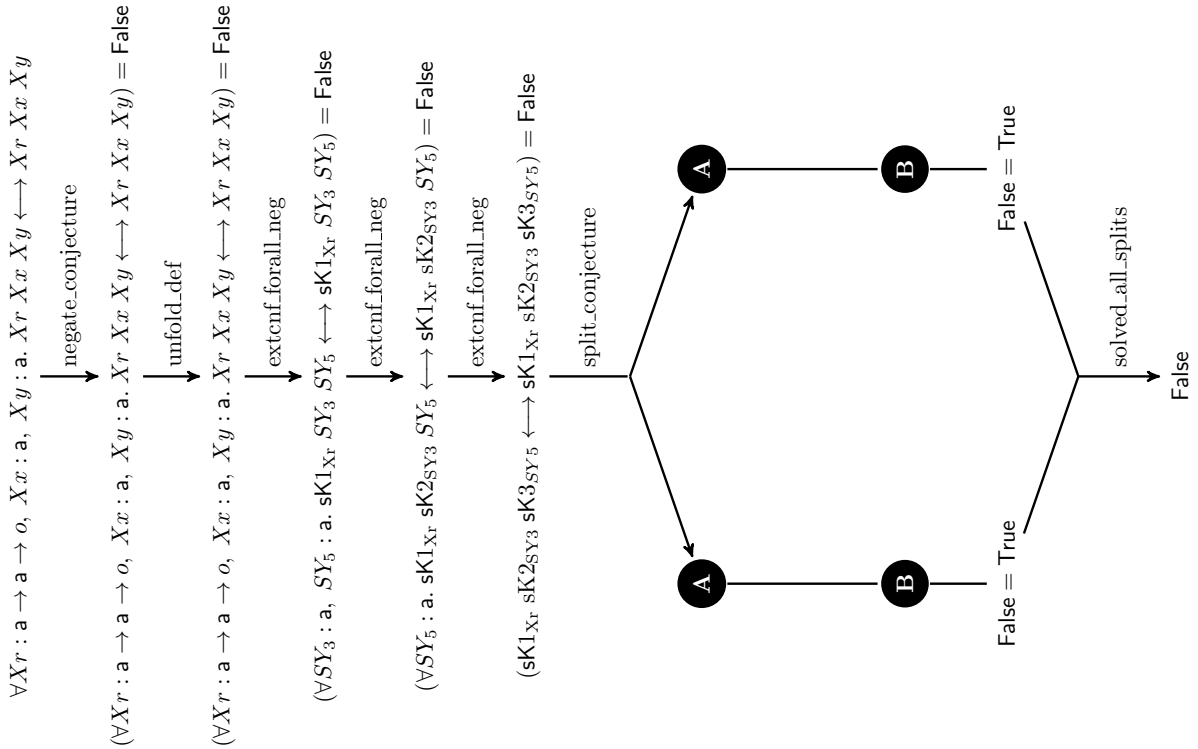
Other connectives are handled in the obvious analogue ways.

4.2.2 Skolemisation

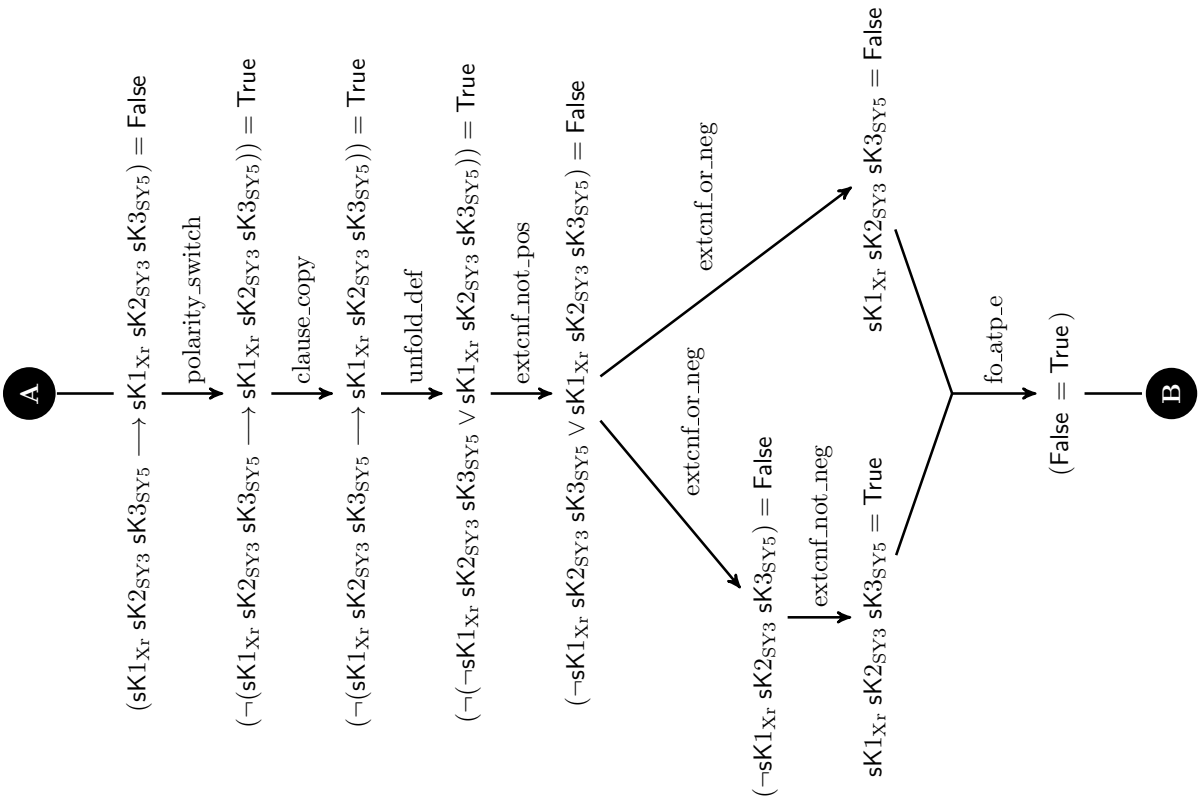
Skolemisation is formalised by the following rule:

$$\frac{\exists x.\, F(x)}{F(\mathsf{c})}\ \mathsf{c} \text{ is a fresh constant}$$

This is consistency-preserving: if $\exists x.\, F(x)$ is true in a model, then we can pin down one such x satisfying F in that model, denote that element using a fresh symbol c, and the application of F to c should result in the same truth value. But this inference is not validity-preserving: if $\exists x.\, F(x)$ is true in all models, it does not mean that $F(\mathsf{c})$, for arbitrary fresh c, is valid too, since we can always find a countermodel for a given constant. For instance, suppose that $\models \exists x.\, F(x)$, and that we want to use the constant c to derive $\models F(\mathsf{c})$ according to the above rule. But the set of models contains the model $\mathcal{M} = (\{\mathsf{c}, \mathsf{d}\}, \{F(\mathsf{d})\})$, where $\models \exists x.\, F(x)$ but $\not\models F(\mathsf{c})$. Consequently we cannot use the rule soundly in a validity-preserving calculus.



(a) The beginning and end segments of Leo-II's proof for SEV161^5. The splitting-related sub-proofs happen to be identical. This sub-proof is shown below.



(b) The splitting sub-proof. We obtain the full proof of SEV161^5 by inlining this sub-proof in (a).

Figure 4.1: Visualisation of Leo-II's proof of problem SEV161^5, showing an occurrence of splitting.

4.2.3 Splitting

Intuitively, splitting consists of an elimination by cases. It allows us to reason using smaller formulas. This is desirable in practice, since ATP systems often do not cope well with large formulas. For simplicity we will only consider splitting a clause into two clauses, but splitting can be generalised to be n-ary.

Recall that the *Herbrand universe* is the set of ground (i.e., variable-free) instances of a set of formulas. Refutation proofs often involve finding inconsistent ground instances of formulas. Schulz (2002a) gives an example to convey the appreciable benefit of using splitting when the Herbrand universe is finite. Let us recount this example. Consider a formula $C \vee D$, where C and D are (free-)variable disjoint. Let the expression $\mathrm{Free}(C)$ denote the free variables in C. Then if the size of the Herbrand universe is $|U|$, the total number of instantiations σ we might need to try before finding a refutation—that is, showing that $(C \vee D)\sigma = \mathsf{False}$—is $|\mathrm{Free}(C) \cup \mathrm{Free}(D)|^{|U|}$. Since C and D are variable-disjoint, we can split this formula and search for instantiations σ_1 and σ_2, which make $C\sigma_1 = \mathsf{False}$ and $D\sigma_2 = \mathsf{False}$ separately. The total space of instantiations is now potentially much smaller: $|\mathrm{Free}(C)|^{|U|} + |\mathrm{Free}(D)|^{|U|}$. When the Herbrand universe is not finite, ATP systems benefit from working with smaller clauses. We now look at two common ways of implementing splitting.

Splitting through search-space partitioning. Splitting involves partitioning the search space according to the disjuncts being split, and looking for a model of one of the disjuncts. Once a model is found, the search terminates. If a contradiction is found, then a model using the next disjunct is sought. If no models are found, it means that a contradiction was derived for either part of the split, and thus for the original formula. (Either part of the split corresponds to a partitioning of the search space.) Splits can be nested—this is how an n-ary split can be carried out using a sequence of applications of a binary splitting rule. A stack can be used to implement splitting, to manage nested splits and the associated book keeping. An element on the stack indicates whether the prover is currently processing the left or right disjunct, together with other information.

This style of splitting is described by Weidenbach (2001) and Fietzke and Weidenbach (2009), and it is the style of splitting implemented in Leo-II (Sultana and Benzmüller, 2012). Figure 4.1 shows an example of a Leo-II proof that features splitting. As an inference rule, this form of splitting could be described as follows:

$$\dfrac{\begin{array}{ccc}\overset{\vdots}{C} & & \overset{\vdots}{D} \\ \mathsf{False} & & \mathsf{False}\end{array}}{\mathsf{False}}\ \mathrm{Free}(C) \cap \mathrm{Free}(D) = \emptyset \qquad \text{from } C \vee D$$

Splitting through new symbols. The trouble with the previous technique is that implementing it in an existing theorem prover can require a significant amount of re-engineering. We need to add the stack-based state management to the prover's main loop, and make sure that all the datastructures are being used to their full potential. This motivated the development of ways to simulate splitting in existing provers, without changing their internals (de Nivelle, 2001; Riazanov and Voronkov, 2001; Schulz, 2002a). This approach does carry a performance cost however (Fietzke and Weidenbach, 2009; Bridge and Paulson,

2013). In this approach, splitting consists of an inference rule on the same level as other inference rules, and does not require management of the search space via a stack. This is the style of splitting implemented in E. As an inference rule, this form of splitting could be described as follows:

$$\frac{C \vee D}{(C \vee \mathsf{d}) \;\&\; (D \vee \neg \mathsf{d})} \;\; \mathrm{Free}(C) \cap \mathrm{Free}(D) = \emptyset, \; \mathsf{d} \text{ fresh}$$

Recall that the dashed line indicates that the inference preserves satisfiability, and the double line indicates that, operationally, the rule deletes the premise from the search space, and adds the conclusions to the search space. The '&' operator here indicates that the two clauses are conjoined with each other. That is, the two clauses need to be satisfied *together*. Operationally, this means that the premise clause $(C \vee D)$ is rewritten into the two conclusion clauses ($C \vee \mathsf{d}$ and $D \vee \neg \mathsf{d}$).

In E, the above rule is called *introduce definition*. Structurally, it is like a converse of resolution, since it introduces complementary atoms in the clauses that are produced by the split. Note that d is a fresh propositional letter that is used to relate the two parts of the split. It is a *definition symbol*, and it consists of a fresh constant. Recall that the search space has not been partitioned. Therefore care must be taken to avoid resolving the two splits before they are reduced to singletons, containing d and $\neg \mathsf{d}$ respectively. That is, if both C and D turn out to be unsatisfiable, we will be left with the singleton clauses $\{\mathsf{d}\}$ and $\{\neg \mathsf{d}\}$, to which unit resolution can be applied.

The *introduce definition* rule is consistency-preserving since:

- If there is no model satisfying $C \vee D$, then there is no Boolean valuation of d that can satisfy both $C \vee \mathsf{d}$ and $D \vee \neg \mathsf{d}$.
- If there is a model satisfying $C \vee D$, then
  - If C is satisfied by the model, then the conclusion is satisfiable in an extended model in which $\mathsf{d} = \mathsf{False}$.
  - If D is satisfied by the model, then the conclusion is satisfiable in an extended model in which $\mathsf{d} = \mathsf{True}$.

If both C and D are satisfied by the model, then the valuation of d does not matter—any valuation of d would satisfy both $C \vee \mathsf{d}$ and $D \vee \neg \mathsf{d}$.

E has another rule called *apply definition*, which is formalised in the following way:

$$\frac{\sigma(C \vee \mathsf{d}) \;\&\; (C \vee D)}{\sigma(C \vee \mathsf{d}) \;\&\; (D \vee \neg \mathsf{d})} \;\; \mathrm{Free}(C) \cap \mathrm{Free}(D) = \emptyset$$

where d is a (previously-introduced) definition symbol, σ is a renaming of variables, and C does not contain other definition symbols. Naturally, since σ is a renaming of variables, it does not affect definition symbols. Intuitively, this rule reduces the size of a clause without affecting the satisfiability or unsatisfiability of the clause set. Operationally, this rule rewrites the clause $C \vee D$ into the smaller clause $D \vee \neg \mathsf{d}$. Note that we could recover the original clause, modulo renaming of free variables, by resolving $D \vee \neg \mathsf{d}$ with $\sigma(C \vee \mathsf{d})$.

The *apply definition* rule is consistency-preserving since:

- If there is no model satisfying both $\sigma(C \vee \mathsf{d})$ and $C \vee D$, then it must mean that in all models, $C = \mathsf{False}$. It also means that in all models, $\mathsf{d} = \mathsf{False}$ or $D = \mathsf{False}$. It cannot be that $\mathsf{d} = \mathsf{False}$ in all models, therefore it must be that $D = \mathsf{False}$. Then, irrespective of σ, there is no valuation of d that can satisfy both $\sigma(C \vee \mathsf{d})$ and $(D \vee \neg\mathsf{d})$, since at least one of them is not satisfied.

- If there is a model satisfying both $\sigma(C \vee \mathsf{d})$ and $C \vee D$, then

  - If C is satisfied by the model, then the conclusion is satisfiable in a model in which $\mathsf{d} = \mathsf{False}$.
  - If D is satisfied by the model, then the conclusion is satisfiable in a model in which $\mathsf{d} = \mathsf{True}$.

As before, if both C and D are satisfied by the model, then the valuation of d does not matter—any valuation of d would satisfy both $\sigma(C \vee \mathsf{d})$ and $D \vee \neg\mathsf{d}$.

4.2.4 Other rules

Some rules that appear in consistency-preserving calculi also happen to be validity-preserving. Examples include resolution, factoring and primitive substitution in higher-order resolution calculi (Benzmüller, 1999), and the α, β, and γ rules in tableau calculi (Fitting, 1996). These rules can readily be used in validity-preserving calculi, without any transformation.

We will be interested in preserving the validity-preserving quality of such rules. That is, we will study transformations that map consistency-preserving rules into validity-preserving ones, and map validity-preserving rules into validity-preserving ones.

4.3 Embedding refutation proofs in Isabelle

Up to now this chapter has presented prior work in automated theorem proving. We will now start probing the problem of embedding refutation proofs in Isabelle, to find a way of translating proofs generated by an ATP system into Isabelle/HOL. The purpose of this section is to explore some ideas for embedding proofs, before developing one of them in the next section.

An ATP system implements an inference pipeline consisting of a preprocessing stage followed by the main loop stage. Both of these stages contain consistency-preserving inferences. We would like to translate the output proof of this pipeline into Isabelle/HOL, which implements a validity-preserving calculus. Ultimately, the proof produced by the ATP system should be mapped into a theorem of Isabelle/HOL. How can we go about this?

4.3.1 Ideas for embedding

In this section we consider some ways of embedding E's calculus in Isabelle. Ultimately we want to allow a proof found by the former to be validated by the latter.

Compared to E's calculus, Leo-II's proof system is simpler in the sense that it does not distinguish between preprocessing and main-loop stages. This is because clausification can occur at arbitrary points during proof search in HOL, unlike in FOL where it only takes place

at the start. Both E's preprocessing and main-loop stages contain consistency-preserving inferences. In the inference system of the preprocessing stage we have Tseitin-style clausification and Skolemisation, while in the main-loop's inference system we have splitting. E's clausification procedure is based on that described by Nonnengart and Weidenbach (2001).

We will therefore focus on tackling E in this section, since it is the more challenging of the two. Let us concentrate on handling clausification in Isabelle/HOL to emulate E's clausification. We could approach this by first carrying out a validity-preserving clausification in Isabelle/HOL and then proving that the result is equivalent to the Tseitin-style result found by E, as described next. Let F be the original formula, $\mathsf{T}(\neg F)$ be the Tseitin-clausified form of the negated conjecture, and $\mathsf{V}(\neg F)$ be the iff-clausified form. As described in Section 4.2.1, computing $\mathsf{V}(\neg F)$ is much more expensive in general than computing $\mathsf{T}(\neg F)$. However, it should be the case that:

$$\begin{array}{ll} & F \text{ is valid} \\ \text{iff} & \neg\mathsf{V}(\neg F) \text{ is valid} \\ \text{iff} & \neg\mathsf{T}(\neg F) \text{ is valid.} \end{array}$$

Our goal is to prove F in Isabelle/HOL, starting with a proof of $\neg\mathsf{T}(\neg F)$ provided by E. Since $\mathsf{V}(\cdot)$ is validity-preserving, we can prove that $\neg\mathsf{V}(\neg F) = \neg\neg F$ in Isabelle/HOL since we are using logical equivalences. Recall that $F = \neg\neg F$ in E and Isabelle/HOL, and E's proof is done by refutation, therefore we will seek to prove $\neg\neg F$ in order to prove F. Using this fact, in order to prove F, it remains to show that $\neg\mathsf{V}(\neg F)$ using the proof of $\neg\mathsf{T}(\neg F)$ provided by E's main loop.

Since we are focussed on proving the equivalence between $\neg\mathsf{T}(\neg F)$ and $\neg\mathsf{V}(\neg F)$, let us assume that E's proof of $\neg\mathsf{T}(\neg F)$ can be reconstructed in Isabelle/HOL. Then it remains to show, in Isabelle/HOL, that $\mathsf{T}(\neg F) = \mathsf{V}(\neg F)$. Since from E's reconstructed proof we know that $\neg\mathsf{T}(\neg F)$ is valid, it remains to show that $\neg\mathsf{V}(\neg F)$ is valid. It is not immediately clear how to build this proof from the available information. Using proof search would add more complexity on top of the iff-clausification, and this is undesirable. This approach would not be very appealing.

It turns out that we do not need to rely on $\mathsf{V}(\cdot)$, and we can solely rely on the proof of $\neg\mathsf{T}(\neg F)$ to prove F via a proof-theoretical embedding. Before studying this in more detail, let us briefly look at alternative approaches.

Deeply embedding E in Isabelle/HOL. This would involve embedding E's proof system in Isabelle/HOL, similar to the work done by Ridge and Margetson (2005) to embed and verify a proof procedure for FOL in Isabelle/HOL. The proof system of E consists of the rules related to preprocessing (clausification and Skolemisation) as well as those used by the main loop. Using this embedding, we could then interpret E's proofs in Isabelle/HOL, but not as native theorems of Isabelle/HOL. That is, the resulting theorems would be formal statements in Isabelle/HOL saying that $\vdash \ulcorner F \urcorner$ is an E-theorem rather than $\vdash F$. To get from the former to the latter we could then use *reflection*. Harrison (1995) discusses this approach in detail. This would involve validating the following inference rule in Isabelle/HOL

$$\frac{\ulcorner F \urcorner \text{ is an E-theorem}}{F}$$

(possibly for specific instances of F). Alternatively, we could seek to prove the theorem $\vdash \ulcorner F \urcorner$ is an E-theorem $\longrightarrow F$, then use modus ponens to obtain $\vdash F$. While certainly

interesting, either of these approaches is likely be rather difficult because of the subtle formal manipulations involved. Instead of embedding E, we could use a simpler or existing embedded prover (Ridge and Margetson, 2005), but that is no different from attempting to re-find the proof, unless E's inferences can be related to the prover's inferences.

Extending Isabelle with E. Instead of embedding E in Isabelle/HOL, we could extend Isabelle (Isabelle/HOL's metalanguage) to reason using E's calculus, giving us Isabelle/E. This could serve as a checker for E's proofs. We could then add metatheoretical functions to transfer Isabelle/E theorems into Isabelle/HOL. It is not immediately clear how to define these metatheoretical functions. An idea would involve describing them as theorem embeddings. But then we could eschew creating Isabelle/E, and map the proofs from E directly to Isabelle/HOL, where they would be checked anyway.

Proof-theoretical embedding. This is the approach that we will develop. It is conceptually clean, and involves defining an abstract mapping that allows us to embed a logic that relies on an *is-satisfiable* judgement, into a logic whose judgement is *is-valid*. This would be a proof transformation applied *uniformly* across the proof, in the sense that the same transformation is applied to each inference step. This makes it much easier to define and argue about. It would need metatheoretical justification, but should allow for an easier implementation than the previous ideas. The implementation could target Isabelle/HOL directly, rather than first going to Isabelle/E and then mapping to Isabelle/HOL. This idea will be developed next.

4.4 Tableau embedding

This section describes how to embed a class of consistency-preserving proof calculi into a class of validity-preserving proof calculi. This is one of the contributions of this chapter. A prototype implementation is described at the end of the section.

In this section we will use Isabelle-style notation, described in Section 2.6.3, to make meta-inference and meta-binding explicit. We will extend this notation with a meta-level existential quantifier, denoted by the symbol $\bigvee$. This will be used in consistency-preserving calculi (such as those of E and Leo-II) but not in validity-preserving calculi (such as that of Isabelle/HOL). This notation will be used only to make meta-level reasoning more precise when reasoning about consistency-preserving systems. That is, the symbol $\bigvee$ will not be interpreted by Isabelle, nor does it appear explicitly in the proofs produced by ATP systems. We will never use the symbols $\bigwedge$ and $\bigvee$ in the same formula. Instead of giving the semantics of $\bigvee$ via inference rules, as done by Paulson (1989) for $\bigwedge$, we will give an intuitive definition of $\bigvee$ here, and will defend its use in every rule in which it appears. This is sufficient for our purposes; there is no need to define a specific calculus, or explore a consistency-preserving counterpart to Isabelle's validity-preserving calculus.

The rough meaning of a formula $\bigvee(x : \tau).\, F(x)$ is: there is an element within the denotation of τ in the universe of discourse, and this element can be denoted by x, such that $F(x) = \mathsf{True}$. Recall that the meaning of a formula $\bigwedge(x : \tau).\, F(x)$ is: for all x typed τ, we have that $F(x) = \mathsf{True}$. As before, we will not represent or refer to type information when this is not important.

Definition 4.4.1 (closed, closed modulo, scoped constants)**.** *A formula is* closed *if all its variables are object-level bound. That is, they are not bound by* $\bigwedge$ *or* $\bigvee$*. A formula is* closed modulo *a* signature extension *if the object-level free variables are bound by* $\bigwedge$ *or* $\bigvee$ *(at the meta-level). The set of* $\bigwedge$*-bound or* $\bigvee$*-bound symbols that are in scope in a subterm is called the* signature extension *relative to which that subterm is interpreted. The variables that are bound at the meta-level are called* scoped constants*.*

For example, $\exists x.x$ is closed, whereas $\bigvee f.\, \exists x.fx$ is closed modulo (and the signature extension consists of f). The formula $\exists x.\ \bigvee f.fx$ is malformed: we cannot have object-level quantifiers whose scope includes meta-level binding occurrences.

Notation 4.4.1. *We write* $\mathcal{T}_1[\mathcal{T}_2]$*, where* $\mathcal{T}_{i\in\{1,2\}}$ *range over (object-logic) formulas, to indicate that* $\mathcal{T}_2$ *occurs in* $\mathcal{T}_1$*. If* $\mathcal{T}_2$ *is an object-level variable then it means that* $\mathcal{T}_1[\mathcal{T}_2]$ *is not closed.*

From here on whenever we speak of a rule

$$\frac{A_1 \quad \dots \quad A_n}{A_{n+1}}$$

in a consistency-preserving calculus, we implicitly include the following variable-closure constraints:

$$\bigvee \vec{\kappa}\ \frac{A_1 \quad \dots \quad A_n}{\bigvee \vec{\kappa'}.\ A_{n+1}}$$

That is, $A_1, \dots, A_{n+1}$ are in the scope of $\bigvee \vec{\kappa}$, and A_{n+1} is in the scope of $\bigvee \vec{\kappa}, \vec{\kappa'}$. We will also consider more general consistency-preserving rules, having multiple conclusions.

Using this notation, the Skolemisation rule described in Section 4.2.2 is written:

$$\frac{\exists x.\, F[x]}{\bigvee c.\, F[c]}$$

We have simply moved the existential quantifier into the meta language. The meaning of the inference rule is exactly as when it was written without using $\bigvee$. Using the $\bigvee$ symbol allows us to be more precise about the scope of $\bigvee$-bound symbols, and this will be crucial in the transformation described in Definition 4.4.2.

You might have noticed that the side-condition of the Skolemisation rule, requiring c to be fresh, was omitted. The side-condition is essential for sound reasoning during proof *search*, but proof translation does not check for soundness. Any unsound reasoning will be caught at the *checking* stage, when the translated proof is processed by a different tool. In this case, if c were not a fresh symbol then when we apply the $\bigvee$-binding we might have captured free variables, leading to unsound reasoning. This will be caught during proof checking. We will prove that the proof transformation we will use will preserve the refutation meaning of the proof—both if the proof is correct, and also if it is incorrect.

The $\bigvee$-enriched rules are applied in the same fashion as the original rules: the rules are applied by an ATP system until a contradiction is encountered, or the clause set is saturated. Each time we apply a rule having a $\bigvee$ in the conclusion, we switch to searching for a model relative to an extended signature. When the scope of $\bigvee$ extends across the whole rule, it means that we are searching within the space of models extending the same signature. We will see an example of both next.

We now translate the stack-less splitting rules from Section 4.2.3 into this notation. This involves using different binding locations—indicating a different scope for the bound symbols. The *introduce definition* rule is written:

$$\frac{C \vee D}{\bigvee d.\ (C \vee d)\ \&\ (D \vee \neg d)}$$

and the *apply definition* rule is written:

$$\bigvee d\ \frac{\sigma(d \vee C)\ \&\ (C \vee D)}{\sigma(d \vee C)\ \&\ (\neg d \vee D)}$$

Note that the scope of $\bigvee d$ extends only across the rule's conclusion in *introduce definition*, whereas it extends across the entire rule in *apply definition*. As mentioned earlier, in the first case we are making precise that, in the conclusion line, we are looking in a space of models where d is interpreted. In the second case, we are *continuing* to look in a space of models where d is interpreted—moreover, the interpretation of d is not changing. Despite the italic script, within the object-formulas the symbol d behaves as a constant. Using the constant notation d would seem inconsistent, since we do not usually bind constants. As mentioned earlier, we can drop mention of side-conditions when stating rules in the context of *translating* proofs—but not when *building* proofs.

You might recall that in the original formulation of the above rules we used two lines to separate the premises from the conclusion, but in the above version we only used a single line. A double line (cf. page 35) indicates that, operationally, the rule deletes the premises and replaces them with the conclusion. Since we have already argued that the rules preserve consistency (cf. page 64), we can be assured that changing the rule's operational behaviour (to not delete the premises) does not affect soundness or completeness. It does affect performance during proof search, but we are assuming that a proof has already been found, and our task is to map it from one calculus to another. We will now describe such a mapping.

Recall that refutation calculi prove results of the form $\models \phi \longrightarrow \mathsf{False}$, and that rules in such calculi are consistency-preserving but not generally validity-preserving. In the argument that follows, the precise rules of a calculus do not always matter—we study properties of a calculus as a whole, by relying on its soundness and completeness to assure us that any inconsistent set of formulas can be refuted.

In what follows let the symbol $\mathfrak{C}$ (the Schwabacher letter "C") represent an arbitrary sound and complete refutation calculus whose inference rules can be adequately formalised as Isabelle terms.[1] Let $\mathfrak{V}$ (the Schwabacher letter "V") represent a calculus whose rules preserve validity, and which can be formalised as Isabelle terms; Isabelle/HOL is such as calculus. We take soundness and completeness to be relative to extensional Henkin models (cf. Section 2.4), but several of the properties described below hold in a broad class of classical logics.

We start with a basic lemma that establishes a correspondence between two families of calculi.

Lemma 4.4.1 (Refutation correspondence). *For all sets of sentences* Φ, $\Phi \vdash_{\mathfrak{C}} \Box$ *iff* $\vdash_{\mathfrak{V}} \Phi \longrightarrow \mathsf{False}$.

Proof sketch: The semantic expansions of $\Phi \vdash_{\mathfrak{C}} \Box$ and $\vdash_{\mathfrak{V}} \Phi \longrightarrow \mathsf{False}$ are identical. (Proof in Section C.1.2) ◁

To translate proofs from Leo-II or E, into proofs in Isabelle/HOL, we would only use the "only if" direction of Lemma 4.4.1. But the lemma is not directly usable, since it does not show us how to translate a proof from $\vdash_{\mathfrak{C}}$ to $\vdash_{\mathfrak{V}}$. We now define a translation that maps

[1] This constraint may exclude esoteric formalisations, but it is weak enough to include calculi such as those of Leo-II and E.

inferences from $\vdash_{\mathfrak{C}}$ to $\vdash_{\mathfrak{V}}$, and that will be used to map *proofs* between the two systems. Intuitively, the translation formalises the consistency-preservation property of a $\vdash_{\mathfrak{C}}$-rule, which we represent using the general notation:

$$\frac{A_1 \quad \ldots \quad A_n}{B_1 \quad \ldots \quad B_m}$$

Note that the premises $A_1 \ \ldots \ A_n$ are implicitly conjoined, while the conclusions $B_1 \ \ldots \ B_m$ are implicitly disjoined. That is, all the premises must hold in a model for it to be necessary that (at least) one of the conclusions holds in a model. The symbols A_i and B_j, for each $1 \leq i \leq n$ and $1 \leq j \leq m$, range over general clauses, as described in Section 2.6.1.

Our transformation will produce another rule which says: if there is no model satisfying at least one of the following sets of clauses

$$\begin{gathered}\{A_1, \ldots, A_n, B_1\} \\ \{A_1, \ldots, A_n, B_2\} \\ \vdots \\ \{A_1, \ldots, A_n, B_m\}\end{gathered}$$

then there is no model satisfying $A_1, \ldots, A_n$. This follows how a refutation-based ATP system operates, extending a set of clauses with their derivations in a consistency-preserving way. The transformed rule is admissible in $\vdash_{\mathfrak{V}}$ (i.e., it is validity-preserving). This is also the informal argument of Lemma 4.4.2, which validates this transformation.

Definition 4.4.2 (Contranegation)**.** *Given a rule r:*

$$\bigvee \vec{\kappa} \ \frac{A_1 \quad \ldots \quad A_n}{\bigvee \vec{\kappa_1}.B_1 \quad \ldots \quad \bigvee \vec{\kappa_m}.B_m}$$

where each $\bigvee \vec{\kappa}, \vec{\kappa_i}.B_i$ is closed modulo, we transform it into $r^{\perp}$:

$$\bigwedge \vec{\kappa} \ \frac{A_1 \quad \ldots \quad A_n \qquad \bigwedge \vec{\kappa}, \vec{\kappa_1}. \begin{bmatrix} A_1, \ldots, A_n, B_1 \\ \vdots \\ \mathsf{False} \end{bmatrix} \quad \ldots \quad \bigwedge \vec{\kappa}, \vec{\kappa_m}. \begin{bmatrix} A_1, \ldots, A_n, B_m \\ \vdots \\ \mathsf{False} \end{bmatrix}}{\mathsf{False}}$$

In Definition 4.4.2, the result $r^{\perp}$ is closed modulo. Also, since '$\bigwedge$' is a binder, nested bindings can always be renamed to avoid variable capture. Let Σ be the set of constants in the signature. We can always ensure that $\Sigma \cap \kappa = \emptyset$, $\Sigma \cap \kappa_i = \emptyset$, and $\kappa \cap \kappa_i = \emptyset$.

Lemma 4.4.2. *Assuming $\vdash_{\mathfrak{C}}$ to be sound, and $\vdash_{\mathfrak{V}}$ to be complete, then for each inference $r \in \vdash_{\mathfrak{C}}$, r is consistency-preserving iff $r^{\perp}$ is validity-preserving.*

Proof sketch: Using semantic reasoning, we can show r and $r^{\perp}$ to be equivalent within a refutation setting, provided that r is sound. (Proof in Section C.1.3) $\triangleleft$

Let the symbol π range over proofs. We write $\pi \in \vdash_{\mathfrak{C}}$ if π is encoded in a consistency-preserving calculus. Proofs consist of finite chains of instances of inference rules $(r_1, \ldots, r_n)$ in that calculus. We write $\Phi \vdash_{\mathcal{C},\pi} F$ to mean that from the set of sentences Φ we can derive sentence F using calculus $\vdash_{\mathcal{C}}$, and that π is such a derivation. Note that, in our setting, $\mathcal{C} \in \{\mathfrak{C}, \mathfrak{V}\}$.

We can now prove a constructive version of Lemma 4.4.1. A proof π can be mapped into $\pi^{\perp}$ by mapping each rule r element-wise to $r^{\perp}$.

Lemma 4.4.3 (Refutation correspondence – constructive). *For every proof* $\pi \in \vdash_{\mathfrak{C}}$*, and for every set of sentences* Φ*, we have* $\Phi \vdash_{\mathfrak{C},\pi} \Box$ *iff* $\vdash_{\mathfrak{V},\pi^{\perp}} \Phi \longrightarrow \mathsf{False}$.

Proof sketch: Induction on derivations π. (Proof in Section C.1.4) ◁

Structurally, Definition 4.4.2 maps $\vdash_{\mathfrak{C}}$-style systems into left-handed tableau systems, the inferences of which can be admitted in $\vdash_{\mathfrak{V}}$-style systems. In our setting, the tableau system is shallowly embedded in Isabelle/HOL. The reconstructed fragment of the search space forms a tableau branch; this branches further when disjunctions are processed, with each disjunct forming a new branch. As is normal with tableau systems, the preceding elements of the branch are shared by all subsequent branches.

When we reconstruct a $\vdash_{\mathfrak{C}}$-refutation into a $\vdash_{\mathfrak{V}}$-proof via Definition 4.4.2 and Lemma 4.4.3, we are being less efficient than when using a tableau calculus directly, because the latter deletes redundant formulas from the branches. Specifically, when an α-rule is applied, the original conjunction is deleted. (Note that we only delete a formula when an equivalent set of formulas is added to the branch; anything weaker would generally introduce incompleteness.) It should be possible to modify Definition 4.4.2, and subsequently Lemma 4.4.3, to allow this deletion. This is an optimisation which makes the reconstruction behave like a tableau proof, but we do not pursue this here.

The space complexity of this method is quadratic in the size of the proof. If $|\Phi|$ is the size of the initial set of formulas that were proved to be inconsistent by the ATP system, and π is the proof produced by the ATP system, then the translated proof will have size

$$\sum_{i=1}^{n} \left(|\Phi| + \sum_{j=1}^{i} |\pi_j| \right)$$

where n is the number of inferences in π, and $|\pi_j|$ is the size of inference j in π (i.e., the size of its hypothesis and conclusion formulas). We get this space behaviour since the translation copies the entire branch (up to that point in the proof) at each step in the proof. This can be seen from Definition 4.4.2, and also from the example implementation in Section 4.4.2. The time complexity for translation is also quadratic; the algorithm within the proof of Lemma 4.4.3 only stitches together the translated inferences. This can also be seen in the translated examples in Section 4.4.2. Note that these are the complexity estimates for *translating* a proof; *checking* a proof can be subject to different bounds if search or unification are involved.

4.4.1 Isabelle encoding of tableau embeddings

This section focuses on specific instances of Lemma 4.4.3, to argue that proofs from both Leo-II and E can be embedded in Isabelle/HOL. As part of this argument, we will also look at examples of consistency-preserving rules from the calculi of Leo-II and E, and explain how they are transformed by Definition 4.4.2. A prototype implementation of this transformation is described in Section 4.4.2, including example translations of E proofs.

4.4.1.1 Leo-II

In principle we should be able to map any proof encoded in Leo-II's calculus into that of Isabelle/HOL.

Corollary 4.4.1. *If Leo-II is sound and Isabelle/HOL is complete (with respect to Henkin models), then every theorem proved by Leo-II is also a theorem of Isabelle/HOL.*

Proof sketch: The proof follows from Lemma 4.4.3, the content of which describes how to map a Leo-II proof into an Isabelle/HOL proof. ◁

Lemma 4.4.3 is parametric in the proof calculi used. We can also verify the lemma's use by checking that, for each of the rules in Leo-II's calculus, the transformation yields Isabelle/HOL-admissible rules. This is done in Section C.1.5.

We now look at some rules from Leo-II's calculus, and the formalisation of their tableau encoding in Isabelle/HOL as meta-theorems. Meta-theorems are essentially admissible rules. We have checked that the contranegation $r^{\perp}$ of each Leo-II rule r is valid in Isabelle/HOL. This is described in Lemma C.1.3, and in an Isabelle/HOL script that is included in Section B.1.2. Furthermore, Section B provides an example proof that was translated using this method.

Resolution. Let $\mathbf{C}, \mathbf{D}$ be meta-variables ranging over Leo-II subclauses, and A, B be meta-variables ranging over formulas. A *subclause* is simply a collection of formulas that can form a clause. Leo-II's resolution rule

$$\frac{\left\{\mathbf{C}, {}^{+}A\right\} \qquad \left\{\mathbf{D}, {}^{-}B\right\}}{\left\{\mathbf{C}, \mathbf{D}, {}^{-}A = B\right\}}$$

whose transformed (Definition 4.4.2) form is

$$\dfrac{\dfrac{\mathbf{C} \vee {}^{+}A \qquad \mathbf{D} \vee {}^{-}B \qquad \mathbf{C} \vee \mathbf{D} \vee {}^{-}A = B}{\vdots\ \mathsf{False}} \qquad \mathbf{C} \vee {}^{+}A \qquad \mathbf{D} \vee {}^{-}B}{\mathsf{False}}$$

can be encoded in Isabelle/HOL as

$$\left[\begin{array}{l} \mathbf{C} \vee {}^{+}A, \\ \mathbf{D} \vee {}^{-}B, \\ \left[\begin{array}{l} \mathbf{C} \vee {}^{+}A, \\ \mathbf{D} \vee {}^{-}B, \\ \mathbf{C} \vee \mathbf{D} \vee {}^{-}A = B \end{array}\right] \Longrightarrow \mathsf{False} \end{array}\right] \Longrightarrow \mathsf{False}.$$

As before, the Leo-II literal ${}^{+}A$ is encoded in Isabelle/HOL as $A = \mathsf{True}$, and the literal ${}^{-}A$ is encoded in Isabelle/HOL as $A = \mathsf{False}$. Leo-II (sub)clauses are encoded as disjunctive HOL formulas.

Note that the rule can be simplified to remove redundant premises, to give

$$\left[\begin{array}{l} \mathbf{C} \vee {}^{+}A, \\ \mathbf{D} \vee {}^{-}B, \\ \left[\ \mathbf{C} \vee \mathbf{D} \vee {}^{-}A = B\ \right] \Longrightarrow \mathsf{False} \end{array}\right] \Longrightarrow \mathsf{False}.$$

In what follows, we will not generally carry out this simplification step. Simplification is possible unless there is a $\bigwedge$-binding occurring inside the rule. An example of this is given further down, on page 74. Note that the lack of simplification also affects the space complexity of the translation, as described in the previous section.

Skolemisation. The Leo-II Skolemisation rule

$$\frac{\left\{\mathbf{C},\ {}^{-}\,\forall A\right\}}{\left\{\mathbf{C},\ {}^{-}\,A\,\mathsf{sk}\right\}}\ \mathsf{sk}\text{ fresh constant}$$

and which can be rewritten using our $\bigvee$ notation as[2]

$$\frac{\left\{\mathbf{C},\ {}^{-}\,\forall A\right\}}{\bigvee sk.\ \left\{\mathbf{C},\ {}^{-}\,A\,sk\right\}}$$

and whose transformed (Definition 4.4.2) form is

$$\frac{\mathbf{C} \vee {}^{-}\,\forall A \qquad \bigwedge sk.\ \left[\begin{array}{c} \mathbf{C} \vee {}^{-}\,\forall A,\quad \mathbf{C} \vee {}^{-}\,A\,sk \\ \vdots \\ \mathsf{False} \end{array}\right]}{\mathsf{False}}$$

can be encoded in Isabelle/HOL as

$$\left[\begin{array}{l} \mathbf{C} \vee {}^{-}\,\forall A, \\ \bigwedge sk.\ \left[\begin{array}{l} \mathbf{C} \vee {}^{-}\,\forall A, \\ \mathbf{C} \vee {}^{-}\,A\,sk \end{array}\right] \Longrightarrow \mathsf{False} \end{array}\right] \Longrightarrow \mathsf{False}.$$

The $\bigvee$-bound sk in the $\vdash_{\mathfrak{C}}$-style calculus becomes $\bigwedge$-bound when mapped into the $\vdash_{\mathfrak{V}}$ calculus. This is detailed in the proof of Lemma 4.4.2. Intuitively, the $\bigvee$-bound sk can denote any element (of the domain) that can satisfy the clause set, whereas the $\bigwedge$-bound sk is used in a setting where *any* denotation of sk must ultimately result in a refutation.

Splitting. Recall that Leo-II implements splitting through search-space partitioning, described in Section 4.2.3. The splitting rule has two conclusions, both of which are ultimately expected to derive False, thus refuting the original disjunction. Having a rule with multiple conclusions is not a problem since Definition 4.4.2 handles such rules. When transformed, this rule becomes:

$$\frac{{}^{+}\,A \vee B \qquad \left[\begin{array}{c} {}^{+}\,A \vee B,\quad {}^{+}\,A \\ \vdots \\ \mathsf{False} \end{array}\right] \qquad \left[\begin{array}{c} {}^{+}\,A \vee B,\quad {}^{+}\,B \\ \vdots \\ \mathsf{False} \end{array}\right]}{\mathsf{False}}$$

[2] As mentioned earlier on page 68, we can drop mention of side-conditions when stating rules in the context of *translating* proofs.

and which can be encoded in Isabelle as

$$\left[\begin{array}{l} {}^{+}\boxed{A \vee B}, \\ \left[{}^{+}\boxed{A \vee B}, {}^{+}\boxed{A}\right] \Longrightarrow \mathsf{False}, \\ \left[{}^{+}\boxed{A \vee B}, {}^{+}\boxed{B}\right] \Longrightarrow \mathsf{False} \end{array}\right] \Longrightarrow \mathsf{False}.$$

As observed before, we can simplify this formula to remove the redundant ${}^{+}\boxed{A \vee B}$ in the inner rules. This will yield a specialisation of the disjunction-elimination rule from the Natural Deduction calculus, but specialised to have the conclusion formula False. This will be revisited in Section 4.5.2.

4.4.1.2 E

In principle we should be able to map any proof encoded in E's calculus into that of Isabelle/HOL.

Corollary 4.4.2. *If E is sound and Isabelle/HOL is complete (with respect to Henkin models), then every theorem proved by E is also a theorem of Isabelle/HOL.*

Proof sketch: The proof follows from Lemma 4.4.3, the content of which describes how to map an E proof into an Isabelle/HOL proof. ◁

As with Leo-II in the previous section, we now look at some rules from E's calculus, and the formalisation of their tableau encoding in Isabelle/HOL as meta-theorems. Ordering criteria are a key feature of proof search in superposition calculi, such as that implemented by E. This is not a concern in the style of proof translation studied here, since we are focussing on *proofs*, not on *proof search*, therefore we can ignore the side-conditions relating to ordering.

Splitting. E splits a disjunction by introducing new symbols, as described in Section 4.2.3. The transformed rule is encoded in Isabelle as

$$\left[\begin{array}{l} \mathbf{C} \vee \mathbf{D}, \\ \bigwedge d. \left[\begin{array}{l} \mathbf{C} \vee \mathbf{D}, \\ \mathbf{C} \vee d, \\ \mathbf{D} \vee \neg d \end{array}\right] \Longrightarrow \mathsf{False} \end{array}\right] \Longrightarrow \mathsf{False}.$$

The *apply definition* rule is transformed as follows:

$$\bigwedge d. \left[\begin{array}{l} d \vee \mathbf{C}, \\ \mathbf{C} \vee \mathbf{D}, \\ \bigwedge d. \left[\begin{array}{l} d \vee \mathbf{C}, \\ \mathbf{C} \vee \mathbf{D}, \\ \neg d \vee \mathbf{D} \end{array}\right] \Longrightarrow \mathsf{False} \end{array}\right] \Longrightarrow \mathsf{False}.$$

Both schemes were verified in Isabelle/HOL to be validity-preserving. Note that we could simplify the formula by removing the redundant $\mathbf{C} \vee \mathbf{D}$ from the inner rule, but we cannot remove either of the clauses mentioning d in the inner rule, since the symbol d is being bound by that inner rule.

Clausification. E's clausification, including Skolemisation, happens during preprocessing (Schulz, 2013, §3.2). This is unlike in Leo-II, where clausification happens during the main loop. The reason for this is that in HOL, a normal (i.e., clausified) formula like $x : o \vee \neg\, y : o$ can become non-normal: for instance, by instantiating x to $\forall z.\ z \wedge y$. This kind of instantiation cannot take place in FOL.

E's clausification procedure is based on that described by Nonnengart and Weidenbach (2001), but here we target the simpler procedure described in Section 4.2.1, and handle Skolemisation as in Leo-II above.

Schematically, the transformation described in Section 4.2.1 takes a formula F and returns a conjunction $\mathsf{T}(F)$, using a collection $\vec{x}$ of fresh names:

$$\overline{\bigvee \vec{x}.\ \overline{F}\ \overline{\mathsf{T}(F)}}$$

Schematically, this is transformed by Definition 4.4.2 as follows.

$$\left[\begin{array}{l} F, \\ \bigwedge \vec{x}. \left[\begin{array}{l} F, \\ \mathsf{T}(F) \end{array}\right] \Longrightarrow \mathsf{False} \end{array}\right] \Longrightarrow \mathsf{False}$$

Let us use $A \vee ((B \longrightarrow C) \wedge D)$ as an example formula. Conjoining the contents of the set $\mathsf{T}(A \vee ((B \longrightarrow C) \wedge D))$ gives

$$x_1 \wedge (x_1 = (A \vee x_2)) \wedge (x_2 = (x_3 \wedge D)) \wedge (x_3 = (B \longrightarrow C))$$

For the sake of readability, we will not expand this into a conjunction of disjunctions, but the explanation in Section 4.2.1 shows how to do this. Applying Definition 4.4.2 we get:

$$\left[\begin{array}{l} A \vee ((B \longrightarrow C) \wedge D), \\ \bigwedge x_1\, x_2\, x_3. \left[\begin{array}{l} A \vee ((B \longrightarrow C) \wedge D), \\ x_1 \wedge \\ \quad (x_1 = (A \vee x_2)) \wedge \\ \quad (x_2 = (x_3 \wedge D)) \wedge \\ \quad (x_3 = (B \longrightarrow C)) \end{array}\right] \Longrightarrow \mathsf{False} \end{array}\right] \Longrightarrow \mathsf{False}$$

which can be shown to be valid.

4.4.2 Prototype implementation

In addition to space complexity, there is some implementational complexity incurred when implementing the previous technique in Isabelle. Normally, ATP systems use data structures allowing them to address specific clauses; but Isabelle natively only allows the addressing of subgoals—not hypotheses. This means that picking the clauses to which we want to apply an inference is not a simple matter: we might need to re-order the clauses making up the hypotheses, since we cannot point to the relevant clauses directly. Working through the second example in Section B.1.4 makes this difficulty appreciable. Having fewer items in the set of hypotheses makes it easier to manage that set; making this set smaller has been suggested above as a possible improvement.

Fortunately, Isabelle also supports *forward proof* —that is, proofs built by combining existing facts, rather than by reducing formulas into axioms. This style of proof allows hypothesis addressing. Proofs can be written in this style using the Isar language (Wenzel, 2002), or the underlying API. This style of proof lends itself more to implementing the method described in this section, compared to the backchaining style of proof that was used in experiments. An example of such an experiment is shown in Section B.1.

The proof mapping described in this section was implemented as a prototype for mapping proofs produced by E into Isar scripts. Isabelle/HOL is then used to import and check these proofs, after which they can be used in Isabelle/HOL developments. The prototype has been made available online.[3]

Note that in E's calculus, proofs do not branch, since splitting does not partition the search space like in Leo-II. (Both splitting approaches were described in Section 4.2.3.) Since proofs do not branch, an E proof can be regarded as a trace through a growing set of clauses, derived from the original set of clauses (consisting of the axioms and the conjecture), such that consistency is preserved. Since E's proofs are linear in this respect, the translation will yield a linear series of inferences in Isabelle/HOL. We will look at examples of this below. The prototype is implemented as a quick check on this method. Currently it has not been tested on a large scale, and does not handle $\bigwedge$-bindings.

We now look at two example proofs translated by the prototype. The TPTP problem SYN929+1 consists of the conjecture

$$\begin{array}{l}\neg(\exists Y.\ \mathsf{p}\,Y) \longrightarrow \\ \quad \forall Y.\ (\exists X.\ \mathsf{p}\,X) \longrightarrow \\ \quad\quad \mathsf{p}\,Y\end{array}$$

E's proof for this conjecture is shown in Figure 4.2. The resulting Isar script from translation by the prototype is shown in Figure 4.3.

We now analyse the translated proof in Figure 4.3, to relate it to the proof mapping described earlier. The translated proof consists of four parts:

1. Problem specification, and start of proof by contradiction: lines 1-6. Proofs always start in the same manner, which involves stating the conjecture, stating that the proof will proceed by contradiction, and naming the negated conjecture formula for later reference.

2. Specification of inference conclusions: lines 8-17. This involves using a handy macro-definition device to shorten the lines making up the next step.

3. Translated proof: lines 18-32. This consists of the original proof's inferences translated in the manner described earlier (Definition 4.4.2). These inferences are chained together to form the translated proof. Note that we are using the *blast* tactic to validate each step. This tactic implements a tableau-based proof search, and its development is described in Section 6.3.

   Note how the first fact, `?r1`, on line 18, is a tautology. Essentially it is subsumed by the tautology $\mathsf{False} \Longrightarrow \mathsf{False}$; one of its hypotheses, `?c_0_9`, is an alias of False. All proofs translated using this method start with such a tautology, and proceed through a series of validity-preserving steps to derive a contradiction from the negated conjecture.

4. Derivation of False from the negated conjecture: line 33.

```
fof(c_0_0, conjecture, ((~(?[X1]:p(X1))=>![X1]:(?[X2]:p(X2)=>p(X1)))),
  file('~/TPTP-v5.4.0/Problems/SYN/SYN929+1.p', prove_this)).
fof(c_0_1, negated_conjecture, (~((~(?[X1]:p(X1))=>
                              ![X1]:(?[X2]:p(X2)=>p(X1))))),
  inference(assume_negation,[status(cth)],[c_0_0])).
fof(c_0_2, negated_conjecture, (![X3]:(~p(X3)&(p(esk1_0)&~p(esk2_0)))),
  inference(shift_quantors,[status(thm)],
            [inference(skolemize,[status(esa)],
             [inference(variable_rename,[status(thm)],
              [inference(shift_quantors,[status(thm)],
               [inference(fof_nnf,[status(thm)],[c_0_1])])])])])).
cnf(c_0_3,negated_conjecture,(p(esk1_0)),
  inference(split_conjunct,[status(thm)],[c_0_2])).
cnf(c_0_4,negated_conjecture,(~p(X1)),
  inference(split_conjunct,[status(thm)],[c_0_2])).
cnf(c_0_5,negated_conjecture,(p(esk1_0)), c_0_3).
cnf(c_0_6,negated_conjecture,(~p(X1)), c_0_4).
cnf(c_0_7,negated_conjecture,(p(esk1_0)), c_0_5).
cnf(c_0_8,negated_conjecture,(~p(X1)), c_0_6).
cnf(c_0_9,negated_conjecture,($false),
  inference(sr,[status(thm)],[c_0_7, c_0_8, theory(equality)]),
            ['proof']).
```

Figure 4.2: TPTP-encoded proof produced by E, for the TPTP problem SYN929+1.

The observation in point 3 above is central to how this method works. All the proofs translated using this method will have tautologies at the leaves of the proof tree, and will gradually proceed by validity-preserving steps from those tautologies to derive a contradiction from the negated conjecture.

We now look at a slightly more complex problem. The TPTP problem SWC138+1 is specified as follows. Note that the problem's original formulation also includes various axioms, but none of them is needed to prove the conjecture.

```
fof(co1,conjecture,
    ( ! [U] :
        ( ssList(U)
       => ! [V] :
            ( ssList(V)
           => ! [W] :
                ( ssList(W)
               => ! [X] :
                    ( ssList(X)
                   => ( V != X
                      | U != W
                      | ~ neq(V,nil)
                      | neq(X,nil) ) ) ) ) ) )).
```

Part of its translated proof is shown in Figure 4.4, which has a similar structure to the pre-

[3] http://www.github.com/niksu/pceil

```
lemma
shows "((~((? X1 . p(X1)))-->(! X1 . ((? X2 . p(X2))-->p(X1)))))"
proof (rule ccontr)
  assume c_0_0 : "~(((~((? X1 . p(X1)))-->
                      (! X1 . ((? X2 . p(X2))-->p(X1))))))"
  show "False"
  proof -
    let ?c_0_1 = "(~((~((? X1 . p(X1)))-->
                      (! X1 . ((? X2 . p(X2))-->p(X1))))))"
    let ?c_0_2 = "((! X3 . (~p(X3)&(p(esk1_0)&~p(esk2_0)))))"
    let ?c_0_3 = "(p(esk1_0))"
    let ?c_0_4 = "(~p(X1))"
    let ?c_0_5 = "(p(esk1_0))"
    let ?c_0_6 = "(~p(X1))"
    let ?c_0_7 = "(p(esk1_0))"
    let ?c_0_8 = "(~p(X1))"
    let ?c_0_9 = "(False)"
    have r1 : "[|?c_0_1; ?c_0_2; ?c_0_3; ?c_0_4; ?c_0_5; ?c_0_6; ?c_0_7;
                ?c_0_8; ?c_0_9|] ==> False" by blast
    from r1 have r2 : "[|?c_0_1; ?c_0_2; ?c_0_3; ?c_0_4; ?c_0_5; ?c_0_6;
                        ?c_0_7; ?c_0_8|] ==> False" by blast
    from r2 have r3 : "[|?c_0_1; ?c_0_2; ?c_0_3; ?c_0_4; ?c_0_5; ?c_0_6;
                         ?c_0_7|] ==> False" by blast
    from r3 have r4 : "[|?c_0_1; ?c_0_2; ?c_0_3; ?c_0_4; ?c_0_5;
                        ?c_0_6|] ==> False" by blast
    from r4 have r5 : "[|?c_0_1; ?c_0_2; ?c_0_3; ?c_0_4;
                        ?c_0_5|] ==> False" by blast
    from r5 have r6 : "[|?c_0_1; ?c_0_2; ?c_0_3; ?c_0_4|] ==> False"
      by blast
    from r6 have r7 : "[|?c_0_1; ?c_0_2; ?c_0_3|] ==> False" by blast
    from r7 have r8 : "[|?c_0_1; ?c_0_2|] ==> False" by blast
    from r8 have r9 : "[|?c_0_1|] ==> False" by blast
    from c_0_0 show ?thesis by (rule r9)
  qed
qed
```

Figure 4.3: Translation into Isar of the TPTP proof from Figure 4.2.

```
lemma
shows "((! X1 . ((ssList X1)-->(! X2 . ((ssList X2)-->
 (! X3 . ((ssList X3)-->
  (! X4 . ((ssList X4)-->
   (((X2~=X4|X1~=X3)|~((neq X2 nil)))|(neq X4 nil)))))))))))"
proof (rule ccontr)
  assume c_0_0 : "~(((! X1 . ((ssList X1)-->(! X2 . ((ssList X2)-->
    (! X3 . ((ssList X3)-->(! X4 . ((ssList X4)-->
     (((X2~=X4|X1~=X3)|~((neq X2 nil)))|(neq X4 nil)))))))))))"
  show "False"
  proof -
    let ?c_0_1 = "(~((! X1 . ((ssList X1)-->(! X2 . ((ssList X2)-->
      (! X3 . ((ssList X3)-->(! X4 . ((ssList X4)-->
       (((X2~=X4|X1~=X3)|~(neq X2 nil))|(neq X4 nil)))))))))))"
    let ?c_0_2 = "(((ssList esk48_0)&((ssList esk49_0)&((ssList esk50_0)&
      ((ssList esk51_0)&(((esk49_0=esk51_0&esk48_0=esk50_0)&
        (neq esk49_0 nil))&
       ~(neq esk51_0 nil)))))))"
    let ?c_0_3 = "(esk49_0=esk51_0)"
    let ?c_0_4 = "(~(neq esk51_0 nil))"
    let ?c_0_5 = "((neq esk49_0 nil))"
    let ?c_0_6 = "(esk51_0=esk49_0)"
    let ?c_0_7 = "(~(neq esk51_0 nil))"
    let ?c_0_8 = "((neq esk49_0 nil))"
    let ?c_0_9 = "(esk49_0=esk51_0)"
    let ?c_0_10 = "(~(neq esk51_0 nil))"
    let ?c_0_11 = "(False)"
    [...]
    from r10 have r11 : "[|?c_0_1|] ==> False" by blast
    from c_0_0 show ?thesis by (rule r11)
  qed
qed
```

Figure 4.4: Part of the translation into Isar of E's TPTP proof for SWC138+1. The translated inferences have been elided under the symbol [...].

vious proof we analysed. The abbreviation `?c_0_11` in Figure 4.4 is used to produce the tautology at the start of the validity-preserving sequence of inferences.

4.5 Implicative embedding

The embedding described in the previous section is appealing since it involves applying a single transformation uniformly to all inferences. This facilitates its implementation. On the other hand, the resulting translation is not efficient in its current form. It suffers in practice because it does not delete subsumed formulas. In this section we look at a different approach that works best in calculi having mostly validity-preserving inferences. Inspecting Leo-II's calculus reveals that only two rules are not validity preserving: Skolemisation and splitting. Most of the rules in E's calculus are validity-preserving too.

If an inference $r \in \vdash_{\mathfrak{C}}$ is validity-preserving, then it should be admissible in $\vdash_{\mathfrak{V}}$. We should be able to encode every such rule directly in Isabelle/HOL, and prove it to be a meta-theorem. Perhaps such an arrangement is not sufficiently worthy of the name 'embedding', but for the sake of classification we can consider it as such. Let us call this embedding *implicative* because it retains the implicative nature of most of the inference rules.

This style of embedding offers appealing features:

- We can formalise each (validity-preserving) r as a scheme, validate it in Isabelle/HOL (proving it to be a meta-theorem), then instantiate it to obtain the precise instance of r used by Leo-II. Clauses in Leo-II are ground, therefore we do not have free universal variables at the level of inferences.

- Since we do not have free universal variables, we can avoid using logical variables when chaining reconstructed inferences. This is desirable since handling such variables requires us to rely on unification, which is not easy to control or tune. Moreover, the scope of such variables spans across branches, creating dependencies.

- All inference rules can be reconstructed separately in isolation, then back-chained together. Indeed, the reconstruction of separate inferences can be parallelised, since (as mentioned above) they are independent from one another. Because we are reconstructing each inference separately, we get a local reasoning scope for each rule, and can write specialised code to target each rule, without fearing interference from ongoing reconstruction in other parts of the proof.

- This backchaining-based approach lends itself well for implementation over Isabelle's core inference engine, which uses resolution. In principle Isabelle's calculus is a constructive sequent calculus, but its reasoning engine, for efficiency reasons, mainly relies on functions that implement derived rules, rather than on the primitive rules of Isabelle's calculus (Paulson, 1989).

Despite its practical appeal, this method is clearly not immediately and universally applicable to all of Leo-II's rules. We now look at how to handle Skolemisation and splitting in this setting.

4.5.1 Implicative Skolemisation

Instead of assigning a fresh constant to a witness, *Hilbert's rule* describes the witness using the information at hand: that such a witness, if it exists, must satisfy a specific predicate. A special combinator, denoted by the symbol ε known as the *indefinite description* combinator, is used to express this inference:

$$\frac{\exists P}{P(\varepsilon P)}$$

Hilbert's rule can be used to simulate Skolemisation by validating the following scheme for a faux Skolem constant c. That is, c is simply a theory-scoped constant, rather than a proof-scoped constant. Skolem constants are normally regarded to be proof-scoped, since they are not required or expected to generally have a particular significance in the background theory. A concrete example showing theory-scoped Skolem constants is given in Section 5.1.

$$\frac{\exists P \qquad \mathsf{c} = \varepsilon P}{P\,\mathsf{c}}$$

To use this inference rule, however, we would need to have Skolem equations (such as $\mathsf{c} = \varepsilon P$ above). Leo-II does not provide these definitions, but we can discover them during a pre-processing phase of the translation. Note that if we unfold Skolem equations (replacing c with εP), this would lead to an exponential growth in the size of the conclusion formula. This danger does not apply here, since Leo-II never unfolds Skolem equations, and we do not need to do this during reconstruction either.

Since we are emulating Skolemisation, the constants must be kept in the context of clause-level universally-quantified variables in order to ensure soundness (Dowek, 2009). That is, Skolemisation produces *Skolem terms*, not only *Skolem constants*. Indeed, simulating Leo-II's inference requires us to handle Skolem terms, and not only Skolem constants; if this were not the case than one would question the soundness of Leo-II's inference.

4.5.2 Combining embeddings for Leo-II proof reconstruction

In the previous section we saw how we can directly interpret all but one of Leo-II's inferences in Isabelle/HOL, allowing us to use an implicative embedding (Section 4.5) for those inferences.

The remaining rule is splitting. We use a tableau-based embedding (Section 4.4) for splitting, and simplify it into the rule shown below—that turns out to be the disjunction elimination rule:

$$\left[\begin{array}{l} \mathbf{C} \vee \mathbf{D}, \\ \mathbf{C} \Longrightarrow \mathsf{False}, \\ \mathbf{D} \Longrightarrow \mathsf{False} \end{array}\right] \Longrightarrow \mathsf{False}$$

By using the validity-preserving counterparts to the Leo-II rules, we can backward-chain through the inferences. This is a style of reasoning which Isabelle can handle very well.

Adequacy. Soundness is assured since (i) the rules that are implicatively embedded are validity-preserving, and (ii) disjunction-elimination, the analogue rule to splitting, is sound. Completeness is assured since every Leo-II inference—be it validity-preserving or not—can be interpreted as an Isabelle/HOL meta-theorem, via the embeddings described above. Moreover, inferences can be combined into proofs of the target logic by following the structure of the source proof.

Complexity. Given a refutation of length n for a set for formulas Φ, using this method it should take time in $\mathcal{O}(n)$ to translate:

- The pre-processing step to extract the Skolem equations (Section 4.5.1) involves scanning the whole proof for Skolemisation steps, taking time in $\mathcal{O}(n)$.
- Each validity-preserving inference (i.e., all of Leo-II's inference rules except for splitting) takes $\mathcal{O}(1)$ to translate, then an m-chain of validity-preserving inferences takes time $\mathcal{O}(m)$.
- Given two splitting-related subproofs, of lengths m_1 and m_2, then applying the splitting counterpart should take constant time, yielding a total cost of $\mathcal{O}(m_1 + m_2)$.

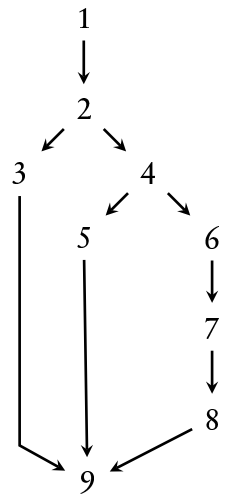


Figure 4.5: An example abstract proof graph.

The space complexity is less straightforward to calculate. Intuitively, it is the most costly fringe of the proof graph, and this is dominated by $\mathcal{O}\left(\Sigma_{\phi\in\Phi}|\phi|\right)$: the sum of sizes of formulas in Φ. A *fringe* is a maximal set of vertices such that none is the ancestor of the others.

For example, in Figure 4.5, node 9 cannot be in a fringe with other nodes, since every other node is its ancestor. Similarly, nodes 1 and 2 cannot be in a fringe with other nodes. The set $\{3, 4, 7\}$ is not a fringe, since 4 is an ancestor of 7. The set $\{3, 5\}$ is not a fringe, since it is not maximal. The fringes of this graph are $\{1\}$, $\{2\}$, $\{3, 4\}$, $\{3, 5, 6\}$, $\{3, 5, 7\}$, $\{3, 5, 8\}$, $\{9\}$. Each vertex in a fringe represents a formula. The size of a formula is the number of atoms and connectives it contains. The most *costly* fringe is the fringe containing the largest sum of vertex sizes. Additionally, the cost could also include a coefficient multiplied by the number of vertices, representing the overhead to store each strand of the proof.

More formally, let $\mathrm{Fr} \subset \mathcal{P}(V)$, where $\mathcal{P}$ is the powerset function, be the set of fringes in a graph (V, E). Let $F \in \mathrm{Fr}$ be a fringe, then the cost of F is $\sum_{v\in F}|v|$, where v is a formula at the conclusion of an inference in the proof, and $|v|$ is the size of that formula. Let $|F|$ be the cardinality of F, and c represent the overhead incurred when storing a branch of the proof. Finally, the estimate for worst-case space requirement of the method is the maximum of the fringe costs:

$$\max\left(\bigcup_{F\in\mathrm{Fr}}\left(\mathsf{c}\cdot|F| + \sum_{v\in F}|v|\right)\right)$$

4.6 Evaluation

Both methods described in this chapter (tableau embedding in Section 4.4, and implicative embedding in Section 4.5) are sound and complete for a large class of calculi. These methods were described abstractly to underscore their wide applicability. The methods' complexity was evaluated too.

A prototypical implementation of the tableau embedding was described, but it was too small and incomplete to warrant a full evaluation. Rather, we will evaluate the implicative embedding as part of the framework described in the next chapter.

Conclusion

In this chapter we studied two methods for mapping proofs between calculi: tableau embedding (4.4) and implicative embedding (4.5). These two methods map from calculi whose inferences are consistency-preserving into calcui whose inferences are validity-preserving.

Many proof checkers such as Isabelle/HOL use validity-preserving calculi, while automatic proof-finding tools, such as Leo-II and E, use consistency-preserving calculi. We looked at examples of rules encountered in the calculi of proof-finding tools, and how these can be mapped into Isabelle/HOL.

The first of these approaches was implemented as a crude prototype, mapping E proofs into Isabelle/HOL. The second approach was implemented completely, as an extension to Isabelle/HOL, to import proofs found by Leo-II. The next chapter will describe the framework on which this import feature is based, including its implementation.

Chapter 5

Reconstruction framework

This chapter describes a framework for proof translation. It builds on the previous chapter, and presents a more abstract interface to proof translation, modelled as cut machines. *I argue that this model can be used for translating* hybrid *proofs: that is, proofs containing subproofs encoded in different logics.*
This chapter also describes an implementation of the framework that can translate Leo-II *proofs into Isabelle/HOL. Every step of the translation is described in detail.*

5.1 Translation pipeline

This section presents a high-level description of how the translator works. In what follows, the *source* logic is the logic being translated, and the *target* logic is the language to which the source logic is translated. For instance, in the implementation described in this chapter, Leo-II is the source logic and Isabelle/HOL is the target logic.

A translator consists of a pipeline of steps applied to some representation of a proof. Recall that there are different kinds of translators. Using the classification in Section 2.3.2, this chapter describes an implementation of an *importer*.

Machine-found proofs, when printed directly, tend to look like intimidating walls of ASCII characters. This underscores the importance of having automatic and reliable means of translating them, since such proofs are not meant for human consumption in their original form. For an example problem, let us take the TPTP problem SEU553^2. In this problem, we use individuals ι to model sets of elements. The powerset function has the type $\iota \to \iota$. The problem conjectures a very simple property: that if two sets are equal, then their powersets are equal too. This is formalised in TPTP as follows:

```
thf(powerset,type,(
    powerset: $i > $i )).

thf(powerset__Cong,conjecture,(
    ! [A: $i,B: $i] :
      ( ( A = B )
     => ( ( powerset @ A )
        = ( powerset @ B ) ) ) )).
```

The proof produced by Leo-II, encoded in the TPTP syntax, is shown in Figure 5.1.

```
thf(tp_powerset,type,(powerset: ($i>$i))).
thf(tp_sK1_A,type,(sK1_A: $i)).
thf(tp_sK2_SY2,type,(sK2_SY2: $i)).
thf(1,conjecture,(![A:$i,B:$i]: ((A = B) => ((powerset@A) = (powerset@B)))),
  file('SEU553^2.p',powerset__Cog)).
thf(2,negated_conjecture,((![A:$i,B:$i]: ((A = B) => ((powerset@A) = (powerset@B))))=$false)),
  inference(negate_conjecture,[status(cth)],[1])).
thf(3,plain,((![A:$i,B:$i]: ((A = B) => ((powerset@A) = (powerset@B))))=$false)),
  inference(unfold_def,[status(thm)],[2])).
thf(4,plain,((![SY2:$i]: ((sK1_A = SY2) => ((powerset@sK1_A) = (powerset@SY2))))=$false)),
  inference(extcnf_forall_neg,[status(esa)],[3])).
thf(5,plain,(((sK1_A = sK2_SY2) => ((powerset@sK1_A) = (powerset@sK2_SY2)))=$false)),
  inference(extcnf_forall_neg,[status(esa)],[4])).
thf(6,plain,(((sK1_A = sK2_SY2)=$true)),
  inference(standard_cnf,[status(thm)],[5])).
thf(7,plain,(((powerset@sK1_A) = (powerset@sK2_SY2))=$false)),
  inference(standard_cnf,[status(thm)],[5])).
thf(8,plain,(((~ ((powerset@sK1_A) = (powerset@sK2_SY2)))=$true)),
  inference(polarity_switch,[status(thm)],[7])).
thf(9,plain,(((sK1_A = sK2_SY2)=$true)),
  inference(clause_copy,[status(thm)],[6])).
thf(10,plain,(((~ ((powerset@sK1_A) = (powerset@sK2_SY2)))=$true)),
  inference(clause_copy,[status(thm)],[8])).
thf(11,plain,(((powerset@sK1_A) = (powerset@sK2_SY2))=$false)),
  inference(extcnf_not_pos,[status(thm)],[10])).
thf(12,plain,((($false)=$true)),
  inference(fo_atp_e,[status(thm)],[9,11])).
thf(13,plain,($false),
  inference(solved_all_splits,[solved_all_splits(join,[])],[12])).
```

Figure 5.1: Leo-II's proof of SEU553^2

We now step through how this proof is handled by the translator. First, the proof is parsed. The logical signature—consisting of a set of types, and a set of constants—is extracted. The signature is described in lines 1-3 of Figure 5.1. The signature is interpreted in the target logic—in this case, `$i` is mapped to the ι type in Isabelle/HOL, and `$i>$i` is mapped to the set of functions $\iota \to \iota$. We also specify the constants powerset, $\mathsf{sK1_A}$, $\mathsf{sK2_{SY2}}$, following the signature described in the proof. The constants $\mathsf{sK1_A}$ and $\mathsf{sK2_{SY2}}$ do not appear in the problem's formulation shown earlier. This is because they are Skolem constants (cf. Section 3.1.1), and they are scoped in the proof, not in the background theory. This was remarked earlier in Section 4.5.1.

After interpreting the signature, the formulas contained in inferences (lines 4-29) are interpreted in the target logic relative to this signature. For example, the formula on line 6 is mapped to the Isabelle/HOL formula

$$(\forall A, B.\ A = B \longrightarrow \mathsf{powerset}\ A = \mathsf{powerset}\ B) = \mathsf{False}.$$

The parsing and interpretation step is described in more detail in Section 5.1.1.

Next, some preprocessing and transformation of inferences is carried out. The inferences form a proof graph: the vertices are formulas, and the edges are the inference steps. The graph is directed, and it is technically a hypergraph, since edges may connect more than two nodes. The proof from Figure 5.1 is shown as a graph in Figure 5.2. The analysis and transformation of the inferences might be done to simplify the proof, or to facilitate further analyses or transformations. These will be described in more detail in Section 5.2.

So far we have processed the logical signature, and the structure of the proof. We now turn our attention to the inferences themselves. The inferences are interpreted relative to the target calculus, to yield meta-theorems. Note that this is different from the interpretation of *formulas* from the source logic. To avoid confusion, we use the word *emulation* to mean the interpretation of *inferences*. Emulation might be implemented using schemes or tactics, or could rely on re-finding instead. For instance, the inference described in line 26 of Figure 5.1 can be emulated by the Isabelle/HOL meta-theorem

$$\frac{(\mathsf{sK1_A} = \mathsf{sK2_{SY2}}) = \mathsf{True} \ \wedge\ (\mathsf{powerset}\ \mathsf{sK1_A} = \mathsf{powerset}\ \mathsf{sK2_{SY2}}) = \mathsf{False}}{\mathsf{False} = \mathsf{True}}$$

where, as specified in the TPTP encoding of the proof, the first premise is contributed from the conclusion of the inference labelled 9 (occurring in line 20), and the second premise from the inference labelled 11 (line 24). As shown above, we conjoin the premises to form a single premise. This ensures that all inferences have a simple form, consisting of a single premise and a single conclusion. The proof text also indicates that this inference was made using E. The resulting meta-theorem in Isabelle/HOL is labelled 12, consistent with the name used in the TPTP encoding of the proof. Emulation is described in more detail in Section 5.5.

So far we have imported all of the constituent information from the source logic into the target logic. We now need to combine the information to reconstruct the theorem in the target logic. The proof graph is traversed, to produce a trace of the (possibly transformed) proof. Following this trace should allow us to reconstruct the proof. As will be described later, following this trace does not always lead to a successful reconstruction. Perhaps this trace could be treated somewhat as a proof plan (Bundy, 1996). Proof planning is the application of planning to proof-search. *Planning* involves finding a solution in a space of

candidates, given constraints described in terms of a set of operators. In proof planning, operators are called *methods*. Methods contain tactics, which in turn use low-level logical inferences to build proofs. In this setting, traversing the proof graph yields a plan of how to build the proof in the target system. The plan is almost always unfailing, and it seems more accurate to refer to it as a *program*. For the example proof above, the program consists of 18 instructions:

```
1 [Step "13", Step "12", Unconjoin,
2  Step "11", Step "10", Step "8", Step "7",
3    Step "5", Step "4", Step "3", Assumed,
4  Step "9", Step "6",
5    Step "5", Step "4", Step "3", Assumed,
6  Caboose]
```

This program will be interpreted next. At the start of the program, the conjecture is implicitly double negated. That is, in Isabelle notation the program in the above example sets out to prove

$$(\forall A, B.\ A = B \longrightarrow \mathsf{powerset}\ A = \mathsf{powerset}\ B) = \mathsf{False} \Longrightarrow \mathsf{False}.$$

This is our proof *goal*, and it corresponds to the negated conjecture. We always start with a single goal, and refine it into subgoals as the proof progresses.[1] As we will see below, the number of goals can vary during the proof. The proof is completed when all goals have been discharged. Note that the conjectured formula can be easily found in the TPTP proof, since it is labelled with the *conjecture* role (cf. Section 2.5.1). In our example, the conjecture is shown on line 4.

We now describe the semantics of the commands appearing in the program shown earlier:

- '*Step* ID' applies the emulated inference labelled with ID. For example, we saw the inference for step 12 above, which relied on the conclusions of steps 9 and 11. The emulation of inferences will be described in Section 5.5.

- *Unconjoin* simply applies conjunction elimination. This is needed since the proof graph is not a tree in general—it recombines. For example, this command is applied to step 12, formalised in Isabelle/HOL earlier, to break up the conjoined premises into two premises. This step consumes one goal, and produces two goals. Each goal relates to a path to the root of the graph. The root of the graph consists of the conjecture formula.

- *Assumed* simply applies an instance of the assumption scheme $(A \Longrightarrow A)$ to discharge a goal. For example, in the program shown above, both uses of this command apply the tautology:

$$\begin{aligned}&(\forall A, B.\ A = B \longrightarrow \mathsf{powerset}\ A = \mathsf{powerset}\ B) = \mathsf{False} \Longrightarrow \\ &\quad (\forall A, B.\ A = B \longrightarrow \mathsf{powerset}\ A = \mathsf{powerset}\ B) = \mathsf{False}.\end{aligned}$$

- *Caboose* terminates the proof. It succeeds if there are no more subgoals. A caboose is the last carriage in a train, and here it should be the last instruction in a program.

[1] We will use the terms *goal* and *subgoal* interchangeably.

Note that in the program shown earlier, lines 3 and 5 are duplicates. We could extract a lemma that fuses together the contents of line 3 into another admissable rule, then replace line 5 with an application of this lemma. This, and the model we use for such programs, is described in Section 5.4.

Finally, we execute this program on the double-negated conjecture. If all emulation steps are successful, then the execution should yield an Isabelle/HOL theorem corresponding to the theorem proved by Leo-II.

5.1.1 Parsing and interpretation

The implementation of this framework uses the Isabelle-integrated TPTP parser and interpreter developed during previous work (Sultana et al., 2012). It is convenient that both TPTP-encoded problems *and* proofs share the same syntax; in TPTP the two kinds of formal objects differ mainly in the annotations given to formulas. These annotations contain information related to the inference made by a theorem prover.

The parser closely follows the grammar specification of the TPTP language, to produce sequences of parsed Annotated Formulas (AFs were described in Section 2.5.1). Recall that an AF can contain additional information to an encoding of a formula. Annotation information is very useful when the TPTP file encodes a proof, but it is hardly ever present when the TPTP file encodes a problem. Thus when we are importing a TPTP problem in Isabelle/HOL, we only care about interpreting formulas. When importing proofs we additionally interpret *inferences*—this will be described in Section 5.5. Recall that we will use the word *emulation* to refer to the interpretation of inferences, and we will reserve the word *interpretation* to mean the interpretation of formulas.

There is a core signature that is already interpreted in Isabelle/HOL, consisting of the base types ι, o, denoting individuals and Boolean propositions respectively, and the type of functions $\tau \to \tau'$ for all types τ, τ'. The signature also interprets the usual constant symbols $\mathsf{True}, \mathsf{False}, \neg, \wedge, \vee, \longrightarrow$, and $\forall_\tau, \exists_\tau, =_\tau, \varepsilon_\tau$ for all types τ.

This core signature is extended with the signature information from the TPTP file, describing the types and constants used in the encoded problem or proof. Type information is not included in untyped TPTP languages, such as FOF and CNF. When available, we use the type information, otherwise we type all atom-level terms as propositions. Sub-atom level n-ary constants, where $n \geq 0$, are given the n-ary function type over ι. That is, they are given the type ι if $n = 0$, the type $\iota \to \iota$ if $n = 1$, $\iota \to \iota \to \iota$ if $n = 2$, and so on. Since the untyped languages are first-order, all constant functions appear fully-applied.

5.2 Proof transformations

In this section we look at the analysis and transformation of proofs, focussing on proofs produced by Leo-II, to facilitate translating the proofs into Isabelle/HOL. There are three proof transformations that were found to be useful for processing Leo-II proofs:

1. *Eliminating redundant parts of the proof.* Occasionally Leo-II includes redundant chains of inferences that do not contribute to the refutation.

   Figure 5.3 shows an example of a proof with redundant inferences. In that graph, note that there are two clauses produced by the standard_cnf inference on the formula

   $$(\forall Xp : \iota \to o.\ Xp\ \mathsf{y} \longrightarrow \forall Xx.\ \mathsf{cR}\ Xx\ \wedge\ \mathsf{cQ}\ \mathsf{y}) = \mathsf{False}$$

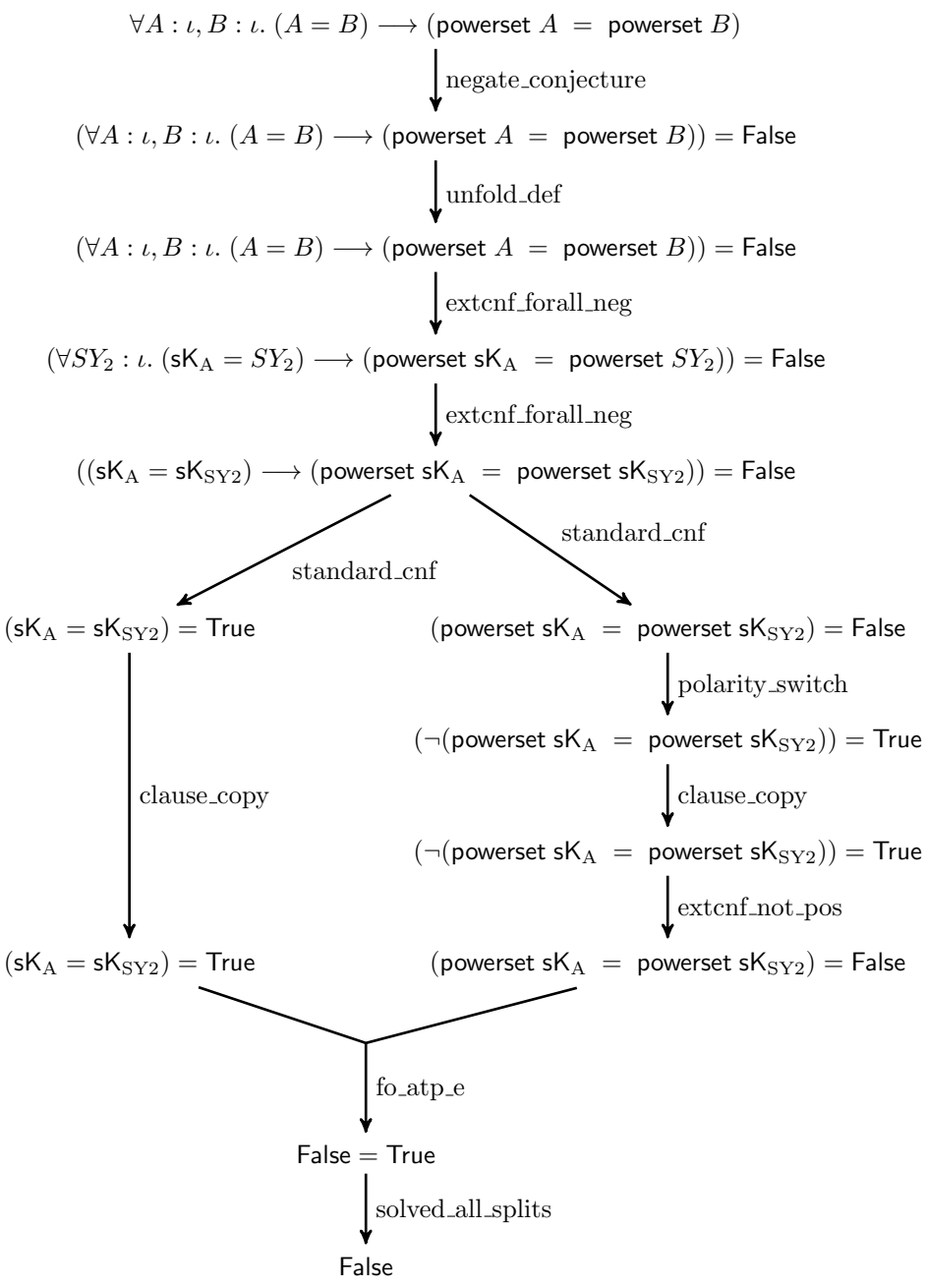


Figure 5.2: Graph for Leo-II's proof of SEU553^2

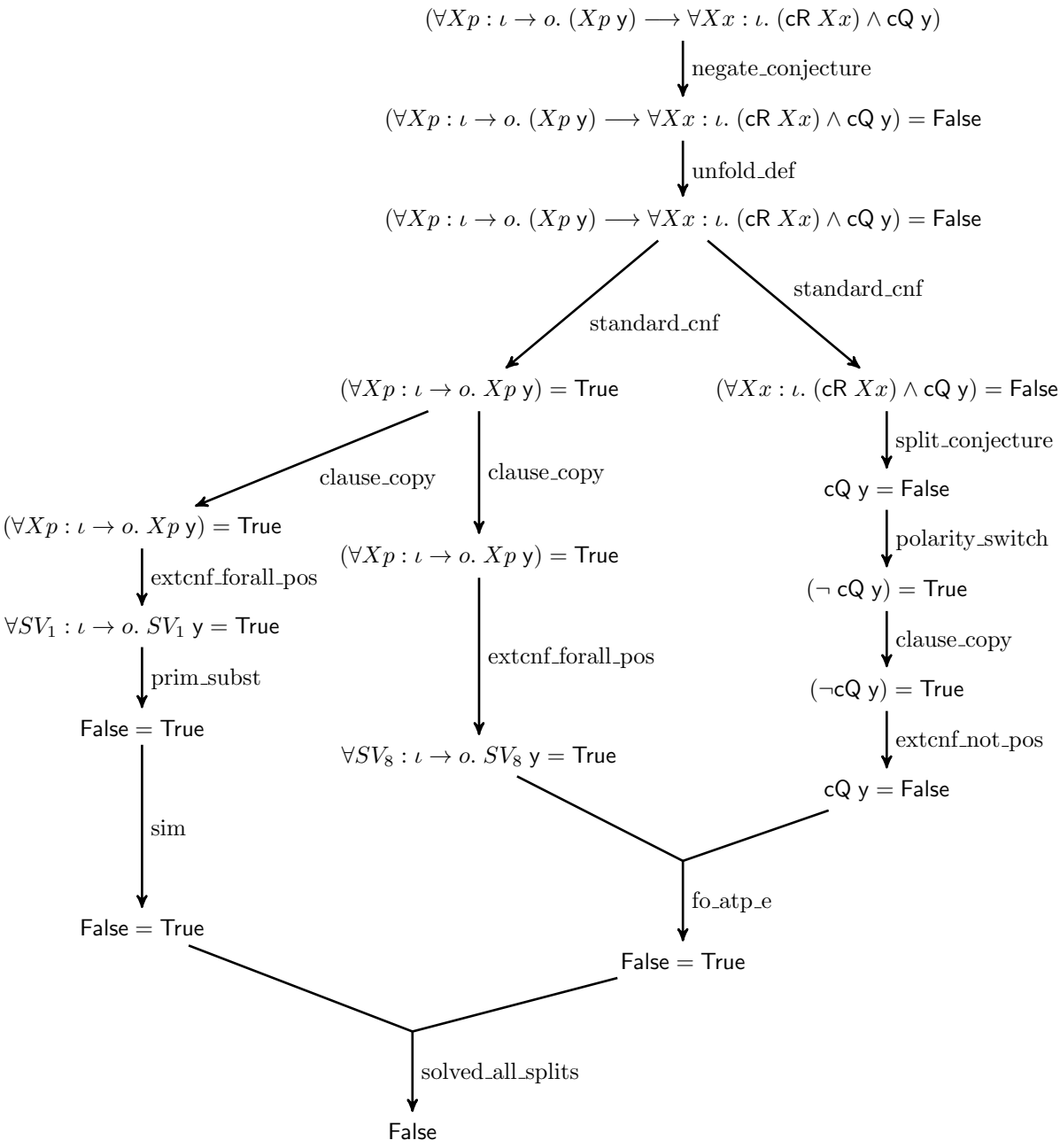


Figure 5.3: Graph for Leo-II's proof of SYO285^5

One of its derivatives, $(\forall Xp : \iota \to o.\ Xp\ \mathsf{y}) = \mathsf{True}$, is copied twice. Its left-side derivative reaches a contradiction independently of its other derivative. Moreover, its right-side derivative relies on another branch of inferences, but this is redundant since we can directly refute its derivative, $\forall SV_8 : \iota \to o.\ SV_8\ \mathsf{y} = \mathsf{True}$ without using the clauses contributed by the rightmost branch. We would simply use the binding

$$SV_8 \mapsto \lambda\_.\mathsf{False}.$$

We can find and eliminate redundant parts of proofs by finding the shortest path from the conclusion of the proof to the conjecture. This must heed splitting—it must keep all paths that derive from a split, since each of them must be refuted independently. In the example above, we keep the leftmost path of the graph, discarding the other two paths.

2. *Extracting subproofs related to splitting.* As described in Section 4.2.3, each subproof of a split yields a refutation. The set of nodes generated during a subproof of a split is disjoint from the nodes from other splits. Each subproof is used to construct a lemma, that produces a premise to the disjunction-elimination rule. An example of such a proof can be seen in Figure 4.1.

3. *Separating instantiation from other inferences.* Leo-II sometimes overloads inferences with instantiation of clause-level (universal) variables. This makes it harder to emulate an inference.

   We could strengthen the emulation of each inference rule to be wary of any additional instantiation which might have taken place. Instead of complicating the behaviour of emulation, we can use a global transformation to separate out instantiation from inferences. This would allow us to handle the two kinds of inference separately. (The two kinds of inference referred to here are the inferences that solely carry out instantiation, and all other inferences.) This transformation is used in our implementation.

   Separating instantiation from other inferences creates additional nodes in the proof graph—the new nodes correspond to the instantiation steps. The instantiation steps do not appear explicitly as inferences in Leo-II's implemented calculus, so we added a new, internal inference rule to encode this kind of inference. This rule is called *bind*, to match the annotation used by Leo-II to indicate such instantiations.

A useful analysis to carry out on Leo-II proofs involves discovering Skolemisation equations (described in Section 4.5.1). This involves analysing the syntax of Skolemisation steps to produce Skolemisation equations, and adding these as axioms to the theory. One might feel justifiably squeamish about adding axioms to the theory, but the axioms concerned here are definitional axioms for Skolem constants. Note that Leo-II proofs generally contain the declarations of Skolem constants (as can be seen in Figure 5.1), but not their definitions. These definitions are implicit in the proof, so this analysis step extracts these definitions to complete the specification of the theory in which the proof is being carried out.

The list of operations on proofs described in this section is not exhaustive—more can be done. For instance, the proof shown in Figure 4.1 involved splitting, but Leo-II used an identical sub-proof for both sides of the split. Finding identical sub-proofs, and re-using previously-constructed tactics for them, would improve the performance of reconstruction.

Additionally, more aggressive simplification of proofs could be done to filter out non-productive inferences. For instance, in Figure 5.4, the sequence of inferences unfold_def,

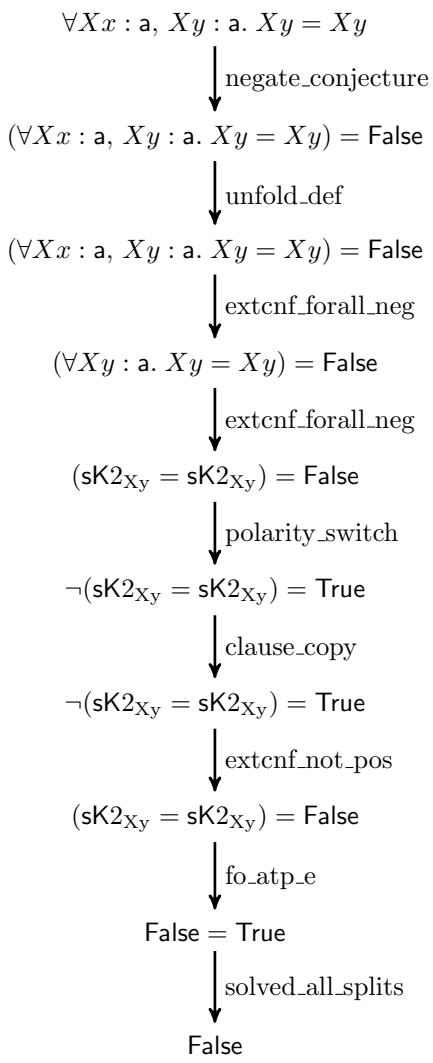


Figure 5.4: Graph for Leo-II's proof of SEV159^5

clause_copy, and extcnf_not_pos undoes the effect of polarity_switch. Also, the inference being done by solved_all_splits is trivial. The proof could be reduced from nine inferences to four.

5.3 Hybrid proofs

We can generalise translating between two proof languages, to look at translating between a *hybrid* proof language and a target proof language. The proofs of a hybrid proof language may contain subproofs encoded in different logics. This is related to *borrowing* (Cerioli and Meseguer, 1997), where the tools for a logic are used to solve problems in another logic. This is usually done for the latter to benefit from the implementation of the former. For instance, this technique is used for modal formulas by Hustadt and Schmidt (2000) by translating them into FOL.

The combination of Leo-II and E is another example of borrowing. Leo-II delegates to E the task of refuting the consistency of a set of (first-order) formulas. These formulas are

computed by Leo-II using its proof calculus, and are derived from the problem given to Leo-II. Thus proof obligations are formed by Leo-II and encoded to E's input language for proving. Let us define *relying logic* to mean the source logic borrowing other logics. For example, in the Leo-II+E combination, Leo-II's is the relying logic, and both Leo-II's and E's are the source logics. In the proof returned by Leo-II, E's contribution to the proof is acknowledged in two ways: either as a compound inference fo_atp_e (as can be seen in Figure 5.4 for example), or as an expansion of this inference into primitive rules used by E.

Translating between two proof languages (the source and target logics) is an instance of this more general approach. When translating between two proof languages, the relying logic is the only source logic. But, as we can see from Leo-II+E, there could be multiple source logics used in the source proof—thus forming a hybrid proof. Hybrid proofs could be crudely turned into single-logic proofs by eliding the subproofs of all logics except for the relying logic under a single inference. For example, Leo-II allows hiding E's inferences as a single Leo-II inference (as can be seen in Figure 5.4 for example). This makes translation more difficult, since it lacks the information contained within the subproofs, and we must resort to re-finding E's inferences. We can avoid this loss if we have an embedding of all the subproofs into the target language—for example, an embedding of Leo-II's inferences in Isabelle/HOL, *and* an embedding of E's calculus in Isabelle/HOL. We could then use the cut rule to stitch together the translated proofs. We will see how the model described in the next section allows us to translate hybrid proofs; but note that the implementation described in this chapter only handles proofs in Leo-II's calculus, not the full Leo-II+E combination. Composing reconstructed E proofs with Leo-II proofs remains further work.

5.4 Cut machines

This section contains a description of an abstract machine for mapping proofs from one logic into another. This machine will be used as an abstract model of proof translation, and will be validated by proof. A similar method was used by de Nivelle (2002) to describe the generation of proof terms that validate clausifications.

Proof search abhors the cut rule (Benzmüller et al., 2009), but proof translation benefits from it. The cut rule can be used to splice proofs together. This is particularly useful since the proofs could have been translated separately, from different logics. This process is somewhat reminiscent of *purification* in SMT solvers (Krstic and Goel, 2007), where a formula encoding a statement spanning several theories, is transformed into a series of formulas each encoding a statement in a single theory, and each of which must be satisfiable.

Put abstractly, proof translation is a function mapping proofs from one logic to another. This view is too abstract to allow us to study how to extend and compose translations.[2] Extensibility and composition is desirable for implementing more maintainable translations. It also appears that composition facilitates implementing translations of *hybrid* proofs, described in the previous section.

A *cut machine* is defined in terms of two features:

State consists of a tuple $(\rho,\ \sigma,\ F)$, where ρ is a set of assumptions (that can include axioms of a particular theory) and validated inferences, σ is a stack of subgoals, and F is the goal formula.

[2]Here *composition* does not refer to function composition, but rather to the composition of proofs from different calculi.

Instructions given to the machine may consist of the following:

- 'PROVE F' states that F is the goal formula. A goal formula may only be set once per proof.
- 'CUT r' applies the rule $r \in \rho$ to the stack of subgoals in the machine's state. This will be described in more detail below.
- 'END' asserts that a machine is in a terminal state. A terminal state is one where a goal state has been set in the past, and the subgoal stack σ is empty. An instance of END may occur only once in a program, at the end.

Let the symbol $\rhd$ represent the single-step transition relation between states. We will use '$-$' to describe an empty stack, and right-associative ':' to describe the push operation. The symbol $\cup_1$ will be used to describe the union between a set and a singleton: that is, $A \cup_1 B$ abbreviates $A \cup \{B\}$. The formal semantics of the machine's instructions are as follows:

$$\begin{array}{rll}
\textsc{Prove}\ F\text{:} & (\rho,\ -,\ \mathsf{True}) & \rhd\ (\rho,\ F,\ F) \\
\textsc{Cut}\ r\text{:} & (\rho,\ B:\sigma,\ F) & \rhd\ (\rho,\ A_1:\ldots:A_n:\sigma,\ F) \\
 & \text{where } r \in \rho \text{ and } r = \frac{A_1,\ldots,A_n}{B} & \\
\textsc{End}\text{:} & (\rho,\ -,\ F) & \rhd\ (\rho,\ -,\ F).
\end{array}$$

Definition 5.4.1. *A* cut program *consists of a finite sequence of instructions. A* well-formed *cut program consists of a single* PROVE *instruction, zero or more* CUT *instructions, and finally a single* END. *An* initial state *consists of any state of the form* $(\rho,\ -,\ \mathsf{True})$. *That is, an initial state simply asserts a very weak statement:* True. *A* terminal state *consists of any state of the form* $(\rho,\ -,\ F)$. *That is, a terminal state must have an empty subgoal-stack.*

We will use the symbol $\vdash_\rho$ to denote the finite proof system encoded in ρ. Note that a cut program without a single PROVE instruction will only prove the tautology $\vdash_\rho \mathsf{True}$. A cut machine running a well-formed cut program can get stuck in two ways:

- when executing 'CUT r' if $r \notin \rho$ or if the conclusion of r does not match the top element in σ, or
- when executing END if the machine is not in a terminal state.

Definition 5.4.2. *A cut program that does not get stuck is called* well-going.

5.4.1 Validating the model

Use of the model relies on the assumption that ρ contains all the rules needed by the cut program. The finite set ρ contains a restricted inference system, consisting of inference rules in the target logic. This set is a parameter to the model, and the generation of these rules takes place externally—such as by using the methods described in the previous chapter to map the source inferences into a form that can be validated in the target system. Provided that a suitable ρ exists, the model has the following properties.

Lemma 5.4.1. *Provided it is given a well-going cut program (cf. Definition 5.4.2), the cut machine has the following invariant: if the subgoals are valid, then the goal is valid too.*

Proof sketch: The proof proceeds by induction on the length of cut programs. (Proof in Section C.2.1) ◁

Lemma 5.4.2. *If a cut program reaches a terminal state then its proof goal is valid.*

Proof. Let (PROVE $G, \ldots$) be a terminating cut-program. G is the proof goal. A terminal state encodes the following meta-theorem:

$$\frac{\vdash_\rho \mathsf{True}}{\vdash_\rho G}$$

which is equivalent to '$\vdash_\rho G$'. This was derived from a valid initial state (which simply encoded $\vdash_\rho \mathsf{True}$), so by Lemma 5.4.1 G must be valid. □

Corollary 5.4.1. *A well-going cut program always produces a theorem in the target logic. Moreover, this can be verified by inspecting a proof in the target logic.*

Proof. A well-going cut program must eventually reach a terminal state. The invariant (described earlier in Lemma 5.4.1) ensures that validity is preserved across the machine's execution, starting with the machine's initial state (which is clearly valid). The resulting terminal state assures the validity of the goal (Lemma 5.4.2).

We now turn to how the machine reconstructs the proof in the target logic. Essentially this is described in the CUT-case of Lemma 5.4.1. The proof grows from the goal, by applying rules from ρ. □

Cut machines are inspired by how Isabelle works, which in turn is inspired by earlier work on LCF, which is based on backward chaining. Cut machines are so-called because they rely on applying instances of a Cut rule at the meta-level. This is not visible in the notation used, in the interest of reduced clutter, but we will give an expanded version below. By default, neither LCF nor Isabelle keep a representation of the proof as it is being built.[3] A proof is not stored by the cut machine defined earlier either, but it could be modified to do so (by using a graph instead of a subgoal stack in the state, for instance).

Consider, for example, the following proof in a source logic:

$$\frac{\dfrac{}{\vdash_\rho A_1} \quad \dfrac{}{\vdash_\rho A_2}}{\vdash_\rho B}$$

The high-level structure of a proof in the target logic will be identical to that in the source logic, except that, instead of primitive inferences, admissible inferences may be used in the target logic.

$$\frac{\frac{A_1\ A_2}{B} \in \rho \qquad \dfrac{A_1 \in \rho}{\vdash_\rho A_1} \qquad \dfrac{A_2 \in \rho}{\vdash_\rho A_2}}{\vdash_\rho B}$$

The role of cut is more apparent in the above representation: in order to prove B, we cut to proving A_1 and A_2, using an inference rule stored in ρ. The set ρ consists of validated instances of inference rules and axioms.

[3] cf. *semantic proofs* in Section 2.1.

5.4.2 Using the model

Among other things, this section will describe how to generate programs that have the well-going property. In order to use the model we require three functions, described below. Before proceeding, let L_1 represent a source logic, and L_2 represent a target logic. 'Logic' here is used to describe essentially the syntactical features of a logic: the syntax of its formulas, and the formation rules of its proofs. These rules are expected to have been validated, via meta-theorems, using suitable semantics. For instance, the source logic could consist of LEO-II's calculus, and the target logic that of Isabelle/HOL. We will assume that the proof calculus of L_2 consists of validity-preserving rules. To use the model we require these three functions:

1. A mapping from formulas of L_1 into formulas of L_2. To be meaningful, this must be defined such that semantics is preserved. We will rely on the interpretation of formulas for this (described in Section 5.1.1).

2. A compiler that takes proofs encoded in L_1 and produces a cut program. At the very least, the compilation must produce well-formed programs. Ideally, the translation should produce well-going programs. An example output of such a compiler was given on page 88.

   As with any compiler, this step might involve several optimisations and transformations.

   If the calculus of L_1 only consists of validity-preserving inferences, then a well-going program can be produced by carrying out a depth-first traversal of the proof graph. (If L_1 is consistency-preserving, then we could map it into a validity-preserving calculus using the methods described in Chapter 4.)

3. A mapping from inferences in the source logic into rules in the target logic. We call this mapping an *emulation*. The resulting rules are not necessarily primitive rules—they could be admissible rules. These rules make up the contents of ρ.

   This mapping could be extracted from an argument showing that any theorem of the source logic is a theorem of the target logic, if the proof proceeds by induction on the structure of (source logic) proofs. The cases of the induction argument describe the mapping we are after.

If the functions above are total and preserve semantic properties, then we are assured that any proof in the source logic can be translated into a proof in the target logic. The translation can be carried out by running the cut program on an implementation of the cut machine.

It is worthwhile to use cut machines as an abstract approach to proof translation since cut machines facilitate extension and composition of logical calculi. These ideas are explored next.

5.4.3 Extending the model

Lifting rules. A cut machine can be specialised by lifting features of the source logic to the level of the machine. This involves extending the definition of the machine and its

instruction set. The lifted feature would then be simulated at the machine level, like the CUT instruction, rather than relying on opaque derivations in ρ.

This can be useful for features such as *splitting*. To recap from Section 4.2.3, splitting is a rule scheme used in clausal calculi to make clauses smaller. We will base the description of splitting on the implementation of this concept in LEO-II. Without loss of generality, we will look at an example starting with a binary clause $\{A, B\}$ such that A and B do not share free variables. We can split this clause into singleton clauses $\{A\}$ and $\{B\}$, but separate refutations must be obtained for each element of the split—that is, $\{A\}$ cannot be used in a refutation derived from $\{B\}$, and vice-versa.

As argued in Section 4.5.2, we can recast splitting as a specialisation of disjunction-elimination.

$$\frac{\vdash_\rho A \vee B \qquad \begin{array}{c}[\vdash_\rho A] \\ \vdots \\ \vdash_\rho \mathsf{False}\end{array} \qquad \begin{array}{c}[\vdash_\rho B] \\ \vdots \\ \vdash_\rho \mathsf{False}\end{array}}{\vdash_\rho \mathsf{False}}$$

Using the current definition of the machine, such a rule could be used outside the machine to populate ρ (remember that ρ is a parameter to the model) with the rule $\frac{A \vee B}{\mathsf{False}}$. We would then use this rule via CUT as before.

Instead, we could modify the machine's definition to *lift* the rule to the machine level, to specialise the machine to support splitting.

Logically, this would be the following rule:

$$\frac{\vdash_\rho \frac{A}{\mathsf{False}} \qquad \vdash_\rho \frac{B}{\mathsf{False}} \qquad \vdash_\rho A \vee B}{\vdash_\rho \mathsf{False}}$$

Note that instead of deriving formulas, we are deriving rules. We could rewrite this meta-rule as follows:

$$\frac{\vdash_{\rho \cup_1 A} \mathsf{False} \qquad \vdash_{\rho \cup_1 B} \mathsf{False} \qquad \vdash_\rho A \vee B}{\vdash_\rho \mathsf{False}}$$

A cut machine can be extended to support this feature by modifying the model such that σ is no longer a stack of formulas, but a stack of pairs: the first element consists of a set of assumptions (which temporarily extends ρ for that subgoal), and the second element is the subgoal formula. The semantics of the new instruction SPLIT $(A \vee B)$, in terms of this model, is:

$$(\rho,\ (C, \mathsf{False}) : \sigma,\ F) \quad \rhd \quad (\rho,\ (C \wedge A, \mathsf{False}) : (C \wedge B, \mathsf{False}) : (C, A \vee B) : \sigma,\ F)$$

Such a machine has been implemented for interpreting LEO-II proofs in Isabelle/HOL. This is described further below. Arbitrary features or rules of a logic could be lifted to the machine level, but that would complicate the machine.

This extended model has less applicability than the original, because it restricts the interpretation of disjunction, and causes the machine to take on features of the target logic. The semantics of the SPLIT rule most naturally fits a classical interpretation of disjunction. This is clearly seen when validating the extended model—this involves proving Lemma 5.4.1 for this model, and including the case for SPLIT. Essentially, this involves proving the soundness of the disjunction-elimination rule in the target logic, for the special case where the conclusion is False.

Read/append cut machines. So far ρ has been treated as a read-only store. We can benefit from regarding it as a monotonically growing store, since this would allow us to progressively prove lemmas and store them in ρ, then use these lemmas to derive further results. This allows us to modularise a proof using lemmas, and could lead to shorter cut programs, shorter translation times, and smaller target proofs. This is linked to the cut-elimination theorem: eliminating Cut involves in-lining the proofs of lemmas, and this can lead to a significant increase in the size of proofs (Boolos, 1984). The machines described so far are vulnerable to additional complexity cost, since shared subproofs (that could be isolated as lemmas) need to be reconstructed each time they are used. We can avoid this by modifying the machine's specification, to allow *appending* to ρ. This can be used to optimise proof translation during compilation, by carrying out a *lemma extraction* step, and encoding each lemma as a sub-program. Note that, since ρ is a parameter to the model, lemma extraction could be carried out *outside* the machine, and the lemmas inserted in ρ. By modifying the machine to help populate ρ, we allow it to play a greater role in checking its contents. This feature has not been implemented in the language described in Section 5.4.5, but we explore it briefly here.

We will formalise this new feature by adding an instruction to store new theorems or rules in ρ. The effect of a STORE command will be to transfer the state of the machine into ρ, and reset the proof goal to True.

$$\textsc{Store:} \qquad (\rho,\ \sigma,\ F) \quad \rhd \quad (\rho \cup_1 (\sigma, F),\ -,\ \mathsf{True})$$

The definition of *well-formed programs* is modified as follows:

- Following a PROVE command, we can have zero or more CUT commands, terminated with either STORE or END.
- A STORE must be followed by another PROVE.

As before, validating this model involves re-proving Lemma 5.4.1 and including the case for STORE. The proof of this case is trivial, it does not even require use of the induction hypothesis. Another useful property can be proved about the extended model, described next.

Lemma 5.4.3. *Let p be a well-going program, ρ^0 be the value of ρ at the initial state of the machine executing p, and ρ^∞ be its value at the terminal state (i.e., after executing END). Then*

1. $\rho^0 \subseteq \rho^\infty$
2. *Let the symbol $\vdash_{\rho^0}$ represent the target proof system extended with the rules in ρ^0. For all $F \in (\rho^\infty \backslash \rho^0)$ we have $\vdash_{\rho^0} F$.*

Proof sketch: The proof proceeds by induction on the length of cut programs. (Proof in Section C.2.2) ◁

A further extension of the model could involve allowing conditional lemmas. That is, allowing us to prove goals such as $\frac{\sigma}{G}$ where $\sigma \neq \mathsf{True}$. Other extensions could include adding the support for branching, jumping or looping programs. Such programs could be used to describe proof search. Exploring more expressive cut-program languages remains future work.

5.4.4 Composing programs

Finally, we will look at program composition. This can facilitate the translation of hybrid proofs, as described in Section 5.3.

Let us wrap the compilation and interpretation functions, described in Section 5.4.2, into a single function called compile. We will use the notation $\mathsf{compile}_{L_1,L_2}$ to describe such a function that takes proofs in logic L_1, and produces data that can be executed by a machine to produce proofs in L_2. In symbols:

$$\mathsf{compile}_{L_1,L_2} : \pi \mapsto (\rho,\ p)$$

where p is a Cut program.

Let a machine's *output* be its goal formula when an END command is executed successfully. For a machine M we indicate this as follows:

$$M(\rho,\ p) = F$$

In the target logic, this establishes that

$$\vdash_\rho F$$

Now if we have different proofs, perhaps encoded in different source logics, that are intended to be combined to prove a single result in the target logic, then we can adapt the notion of *linking* from program compilation, to provide a way of composing these proofs into a single proof in the target logic.

The abstract execution model does not need to change to accommodate program composition. Instead, we could refine the compilation step further to do the following:

1. Decompose a proof into subproofs that are encoded in different logics.
2. Proceed *inside-out*, translating the subproofs into the target logic, and add each subproof's result to ρ.
3. When all the subproofs have been reconstructed, mark *adaptation inferences* for inclusion in ρ. These are inferences that adapt between a source logic and the relying logic. For instance, in the case of LEO-II+E, these inferences carry out the HOL-to-FOL translation.
4. We are now left with a proof in the relying logic—LEO-II in the case of LEO-II+E. Proceed with the compilation as for proofs that have a single source logic, using the pair $(\rho,\ p)$.

Formalising more of this process remains future work.

5.4.5 Implemented language

We saw an example program on page 88, the commands of which were explained below it. In this section we describe the remaining commands that are supported in the implementation of the cut-machine framework.

The datatype used to represent cut programs is shown in Figure 5.5. In addition to the semantics described on page 88, we add the following:

```
datatype rolling_stock =
    Step of step_id
  | Assumed
  | Unconjoin
  | Split of step_id (*where split occurs*) *
             step_id (*where split ends*) *
             step_id list (*children of the split*)

   (*A step that does not necessarily appear in the original
     proof, or that has been modified slightly for better
     handling by Isabelle*)
  | Synth_step of step_id

  | Axiom of step_id (*Mirrors TPTP role*)
  | Definition of step_id (*Mirrors TPTP role*)
  | Caboose
```

Figure 5.5: The datatype encoding cut programs, encoded in Standard ML.

- *Split* $(\mathrm{ID}_1, \mathrm{ID}_2, \vec{\mathrm{ID}})$: Indicates that a split takes place at this point in the proof, which corresponds to ID_1. The split ends at ID_2—at which point, the sub-proofs are recombined. Finally, the label of the first formula of each sub-proof is included in the list $\vec{\mathrm{ID}}$.

- *Synth_step* ID: This refers to an inference labelled ID, that was not part of the original proof, but was added during preprocessing. We call such inferences *synthetic*. Some transformations described in Section 5.2 add synthetic inferences to the proof. As with all other inferences, a synthetic inference is later processed by the emulation process, described in Section 5.5.

- *Axiom* ID: This command simply applies the axiom labelled by ID. For the command to succeed, the current proof goal must be identical to the axiom. Thus this command is used to discharge a subgoal.

- *Definition* ID: A definition is an axiom that has a very specific shape—it is an equation, the left side of which is a constant applied to zero or more terms. The behaviour of the Definition command is similar to that of the Axiom command: it too is used to discharge a subgoal.

Recall the example of splitting shown in Figure 4.1. The graph is reproduced in Figure 5.6, but instead of showing the formulas themselves it shows inference identifiers. These identifiers are referenced in the cut program below, that is used to reconstruct the proof. The program is indented to improve readability. We will step through it below.

```
1 Step "25", Split ("inode2", "25", ["17", "24"]),
2   Step "inode2", Step "6", Step "5", Step "4", Step "3", Assumed,
3   Step "17", Unconjoin,
4     Step "16", Step "14",
```

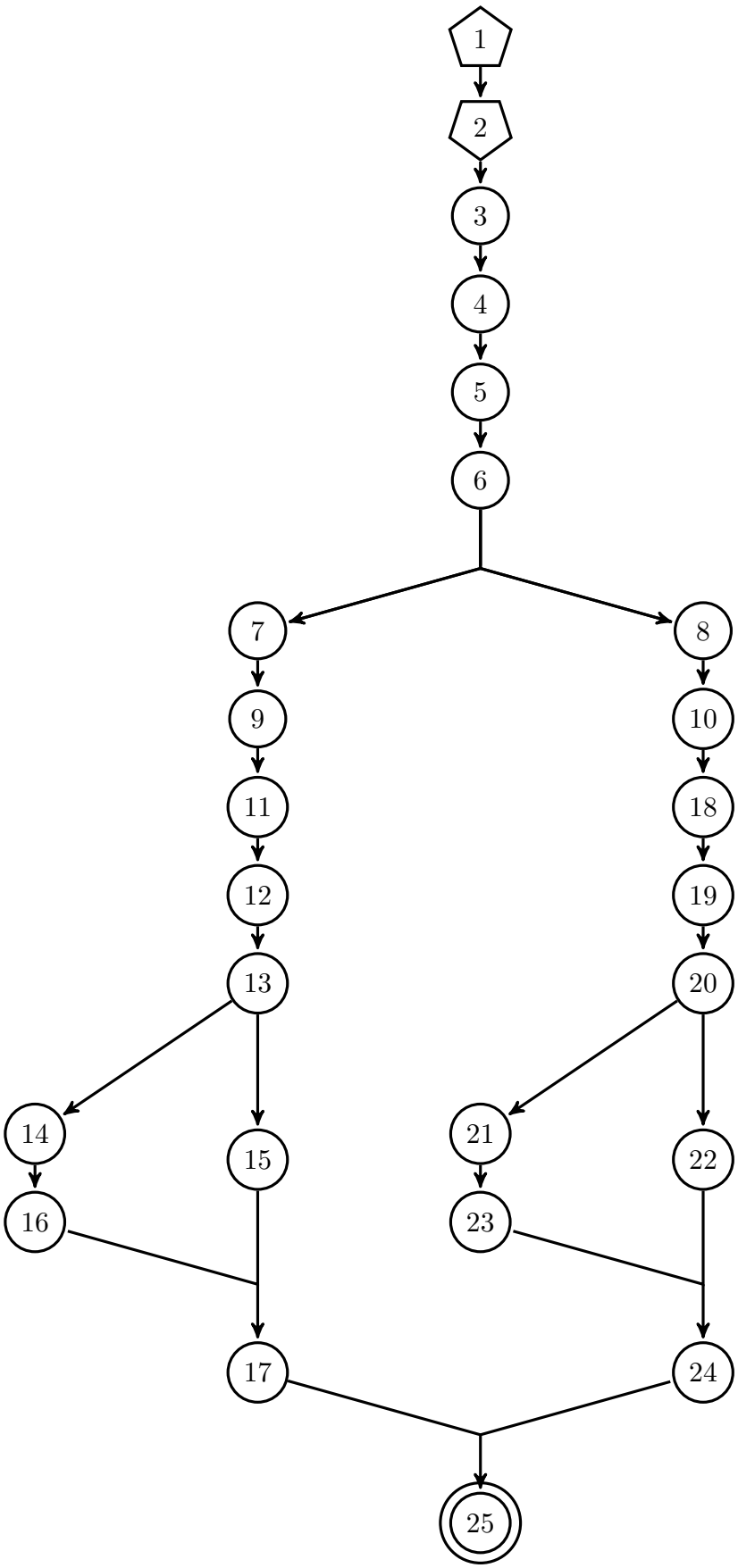


Figure 5.6: Graph for Leo-II's proof of SEV161^5. This is an abstract version of the proof shown in Figure 4.1, showing the inference identifiers instead of their conclusion formulas.

```
5       Step "13", Step "12", Step "11", Step "9", Assumed,
6     Step "15",
7       Step "13", Step "12", Step "11", Step "9", Assumed,
8   Assumed,
9   Step "24", Unconjoin,
10    Step "23", Step "21",
11      Step "20", Step "19", Step "18", Step "10", Assumed,
12    Step "22",
13      Step "20", Step "19", Step "18", Step "10", Assumed,
14  Assumed,
15 Caboose]:
```

We start from the bottom of the graph (step 25) and work our way up. Note that we immediately find that a split has occurred between inode2 (which is a synthetic inference, added after inference 6) and the current node. We then work on three strands: first, the inferences between (and including) the start of the split and the start of the proof (i.e., nodes 1-6), then the two chains of inferences making up the split (one ending at node 17, and the other at 24). As remarked earlier, note the repeated traversal of nodes shown in lines 5 and 7, and in lines 11 and 13. Lemma extraction would remove this duplication.

5.5 Emulation

Emulation here is taken to mean the interpretation of inferences of the source logic in the target logic. In the implementation, Leo-II's inferences are interpreted to be meta-theorems or, more generally, tactics of Isabelle/HOL. Recall that our approach to proof reconstruction is made up of two phases: the shunting of inferences (i.e., proof transformation followed by generating a cut program), and the interpretation of inferences. The first phase generates a cut program, while the second phase assists in the execution of this program. The second phase populates the set ρ that will be used when executing the cut program. Executing the cut program will yield a proof in the target logic.

The two phases are related, but have very different purposes:

- The *shunting* of inferences involves (globally) meaning-preserving transformations applied to a proof, to facilitate its reconstruction in the target system. Shunting is the process of re-ordering rolling stock, and it is a useful metaphor to use here.
- Emulation (i.e., the interpretation of inferences) relates to an embedding of one calculus in a different calculus, that is a mapping of the inferences of one calculus to chains of inferences in another. In the case of Leo-II and Isabelle/HOL, Leo-II's inferences are interpreted as Isabelle/HOL meta-formulas, and reconstructed to give meta-theorems.

It is possible, and advantageous, to separate the implementation of the two, bringing some benefit of modularity, since some details of one can be encapsulated from the other.

We have already covered the shunting phase in some detail in the last pages, and we now turn to emulation—that is, the interpretation of inferences. Essentially, this phase provides the precise interpretation for each Step (and Synth_step) command in the language described in Figure 5.5. We will use the word *emulator* to mean a function that maps an inference rule in the source logic to the target calculus. A single inference rule might have several instances in a proof, but all of these will be handled by the same emulator.

5.5.1 Inference emulation

In the context of a specific proof, an inference emulator is a mapping from an inference identifier (in a proof) into a tactic that validates that inference in the target logic. That is, given that inference as a conjecture, the tactic turns it into a valid inference rule.

We have encountered inference identifiers before—such as in the cut program on page 101. We also saw an example of how inferences are encoded in TPTP on page 31, and how they make use of inference identifiers. An inference identifier is used to extract three pieces of information from the (transformed) proof graph:

- the inference rule used,
- the identifiers of the parent inferences (to obtain the premises of the inference),
- and the conclusion formula.

The second and third items are used to form the meta-formula in the target logic. Then the first item is used to validate the meta-formula into a meta-theorem. Emulators are tasked with providing the means of validating these inferences in the target logic. They can be classified according to their complexity. We describe different kinds of emulators below.

First however, note that the emulation phase depends on the earlier shunting phase, since shunting might have altered the structure of the proof.

5.5.1.1 Simple emulation

Simple emulation involves the use of small, straightforward tactics to validate an inference.

For example, on page 43 we saw how Leo-II's clause_copy inference rule can sometimes do more than simply literally copy a formula: it may explicate that a formula has a positive polarity. This explication is an instance of the following rule, formalised in Isabelle:

```
(*This is not an actual rule used in Leo2, but it seems to be
  applied implicitly during some Leo2 inferences.*)
lemma polarise: "P ==> P = True" by auto
```

Then the emulator for the clause_copy rule simply maps to the following tactic using Isabelle's API:

```
HEADGOAL
 (atac
  ORELSE'
     (rtac @{thm polarise}
      THEN' atac))
```

This tactic tries to apply the assumption tactic (`atac`), which succeeds when the premise and conclusion are identical. (This, strictly speaking, is the expected semantics of the clause_copy rule.) Should this fail, then it attempts to apply the polarise lemma, before applying the assumption tactic.

5.5.1.2 Complex emulation

Some inference rules might require a good deal more work to emulate compared to the previous example. Complex emulation is a straightforward extension of simple emulation, except that many more lemmas might be involved, and the tactic might involve a more complex search for a proof among alternative possibilities.

5.5.1.3 Prover emulation

The most interesting kind of emulation consists of a combination of different emulators. In essence, this is an implementation of an ATP system embedded in a proof translator.

In our importer, this implements Leo-II-like proof search in Isabelle/HOL. The implementation consists of a *preprocessing* stage, a *loop* stage, and a *clean-up* stage. Preprocessing can include stripping quantifiers and simplifying the inference's conclusion. The loop stage consists of applying various inferences until the inference becomes trivial. The clean-up stage consists of simplifying the inference, to facilitate matching with a Skolem equation, or removing duplicate hypotheses.

The prover is parametrised by a configuration that explicates which stages should be carried out, and, within each stage, which inferences should be attempted. In other words, it is a very configurable theorem prover.

As with any theorem prover, the main loop carries out most of the recognisable logical inference. In this prover, the main loop contains inferences that emulate Leo-II's inference rules. It made sense to collect these rules into a small prover that can be specialised to carry out specific inferences, since various rules shared similar preprocessing steps, and code.

In principle, it is possible to use this small prover for proof-search over all the inference rules, but this would not be ideal since the prover is not at all optimised. Also, there is sufficient information in the proofs (in the form of intermediate inferences) to avoid unguided proof-search.

5.5.2 Dependencies

Some of the emulators make use of the *blast* tactic. This tactic is described in Section 6.3. It implements a tableau prover integrated with Isabelle (Paulson, 1999). This tactic has a depth limit (to bound proof search) and it was noticed during experiments that this limit can make a significant difference to success and performance. If the limit is set too low, it diminishes success. If it is set to high, it degrades performance.

Isabelle uses higher-order unification during proof checking, as it resolves rules into its goal state. The implementation of unification too has a search bound, and it was noticed that this has a similar influence to the depth bound mentioned above.

After a small number of experiments, both these values were set to values that allowed the validation of most of Leo-II's proofs for THF TPTP problems within 30 seconds (an evaluation is provided in Section 5.8). It is very likely that additional proofs could be validated if we increase these parameters' values, but this might degrade performance in general.

For an example of an inference that requires a higher unification-bound to emulate, consider this instance from Leo-II's proof for SEV385^5:

$$\frac{(\mathsf{sK2}_{\mathrm{SY13}}\, X \;=\; \mathsf{sK2}_{\mathrm{SY13}}\, Y) = \mathsf{False}}{(X \;=\; Y) = \mathsf{False}}$$

This inference should be trivial to validate, but in this precise instance the symbols X and Y map to rather large terms, requiring us to relax the bounds described earlier. The symbol X stands for the term:

$$\left(\begin{array}{l}(SV_{36} : \mathsf{b} \to \mathsf{b} \to \mathsf{b} \to \mathsf{b} \to \mathsf{b} \to \mathsf{a}) \\ \quad (\mathsf{sK1}_{\mathrm{SY12}}\,(\lambda SX_0 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\,(\lambda SX_1 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}} \\ \quad\quad (\lambda SX_2 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\,(\lambda SX_3 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\,(SV_{36}\,SX_0\,SX_1\,SX_2\,SX_3)))))) \\ \quad (\mathsf{sK9}_{\mathrm{E}}\,(\lambda SX_0 : \mathsf{b},\ SX_1 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\,(\lambda SX_2 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}} \\ \quad\quad (\lambda SX_3 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\,(SV_{36}\,SX_0\,SX_1\,SX_2\,SX_3))))) \\ \quad (\mathsf{sK10}_{\mathrm{E}}\,(\lambda SX_0 : \mathsf{b},\ SX_1 : \mathsf{b},\ SX_2 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}} \\ \quad\quad (\lambda SX_3 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\,(SV_{36}\,SX_0\,SX_1\,SX_2\,SX_3)))) \\ \quad (\mathsf{sK11}_{\mathrm{E}}\,(\lambda SX_0 : \mathsf{b},\ SX_1 : \mathsf{b},\ (SX_2 : \mathsf{b})\,SX_3 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}} \\ \quad\quad (SV_{36}\,SX_0\,SX_1\,SX_2\,SX_3)))\end{array}\right)$$

and Y is

$$\lambda SX_0 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\ \left(\begin{array}{l}SV_{36}\,(\mathsf{sK9}_{\mathrm{E}} \\ \quad\quad (\lambda SX_1 : \mathsf{b},\ SX_2 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\,(\lambda SX_3 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}} \\ \quad\quad (\lambda SX_4 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\,(SV_{36}\,SX_1\,SX_2\,SX_3\,SX_4))))) \\ \quad (\mathsf{sK10}_{\mathrm{E}}\,(\lambda SX_1 : \mathsf{b},\ SX_2 : \mathsf{b},\ SX_3 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}} \\ \quad\quad (\lambda SX_4 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}}\,(SV_{36}\,SX_1\,SX_2\,SX_3\,SX_4)))) \\ \quad (\mathsf{sK11}_{\mathrm{E}}\,(\lambda SX_1 : \mathsf{b},\ SX_2 : \mathsf{b},\ SX_3 : \mathsf{b},\ SX_4 : \mathsf{b}.\ \mathsf{sK2}_{\mathrm{SY13}} \\ \quad\quad (SV_{36}\,SX_1\,SX_2\,SX_3\,SX_4)))\,SX_0\end{array}\right)$$

5.5.3 Handling large formulas

Let us consider implementing a tactic that validates the inference rule

$$\frac{A_1\ \ldots\ A_m}{B_1\ \ldots\ B_n}$$

If $A_1, \ldots, A_m, B_1, \ldots, B_n$ include large terms, then performance is likely to suffer, even if the inference itself is very simple—as we saw in the previous section.

The following implementation technique was found to be useful to improve performance. First create a new, abstract version of the inference

$$\frac{A'_1(\ulcorner A_1 \urcorner)\ \ldots\ A'_m(\ulcorner A_m \urcorner)}{B'_1(\ulcorner B_1 \urcorner)\ \ldots\ B'_n(\ulcorner B_n \urcorner)}$$

In this rule, the symbols $\ulcorner A_1 \urcorner, \ldots, \ulcorner A_m \urcorner, \ulcorner B_1 \urcorner, \ldots, \ulcorner B_n \urcorner$ are fresh variables. The terms $A'_1, \ldots, A'_m, B'_1, \ldots, B'_n$ are abstractions of the terms $A_1, \ldots, A_m, B_1, \ldots, B_n$ respectively up to some depth. Note that $A'_1, \ldots, B'_1, \ldots$ need not necessarily be monadic.

That is, the resulting terms $A'_1(\ulcorner A_1 \urcorner), \ldots, A'_m(\ulcorner A_m \urcorner), B'_1(\ulcorner B_1 \urcorner), \ldots, B'_n(\ulcorner B_n \urcorner)$ are more abstract versions of $A_1, \ldots, A_m, B_1, \ldots, B_n$. An example of this transformation was shown in the previous section, where X and Y were variables that abstracted over arbitrary terms, including the large terms shown on the previous page.

Owing to the simplicity of the resulting rule (with the large terms abstracted away), validating it is very likely to be very fast. Once we have validated the rule, we obtain a meta-theorem that is universal in $\ulcorner A_1 \urcorner, \ldots, \ulcorner A_m \urcorner, \ulcorner B_1 \urcorner, \ldots, \ulcorner B_n \urcorner$. Now if we instantiate these to get $A_1, \ldots, A_m, B_1, \ldots, B_n$ respectively, we have obtained a validated instance of the rule we sought originally. Instantiation is a very simple operation, and it too is likely to take place very quickly. The implementation uses this abstraction technique, and Isabelle's default unification bound. The user may vary the unification bound by setting a global attribute associated with Isabelle's unification engine.

5.6 Executing cut programs

So far, we have seen how the proof is parsed, analysed, transformed, and represented as a program. Some of the commands of this program have semantics that are independent of the proof, while other commands have semantics that depend on the inference instances used in the proof. In the previous section we saw how these semantics are described and implemented using tactics.

Now we turn to the evaluation of programs. This involves mapping the cut program into a theorem in the target system, or into a function that could yield such a value. A tactic is such a function. This has been implemented in two ways in Isabelle/HOL:

- *Tactic compiler.* This evaluates a cut program into an Isabelle tactic. The tactic is specialised to prove the conjecture formula. The generation of the tactic provides some assurance that the reconstruction of the proof will succeed; but full assurance is only obtained once a theorem value is actually produced. A theorem can be produced when the tactic is applied to the conjecture. This is similar to the *lazy theorems* idea proposed by Boulton (1993). There the idea is to avoid costly generation of theorem values by producing closures instead, which can be evaluated to yield a theorem. Theorem generation can be expensive in LCF-style systems since such systems are typically *fully expansive*: that is, ultimately all inference is done using primitive inferences of the logic.[4]

  The implementation also makes use of memoization, to improve performance while generating the theorem value. When building the tactic, we extend a map from Step-commands to tactics, and reuse subproofs (i.e., tactics) when possible. Since proofs are recombinant trees, and since inferences are individually named, whenever we see a Step command (cf. Figure 5.5) with the same inference identifier more than once, we can be sure that the same sequence of commands will follow until we reach the root. Here, the root of the proof graph is the conjecture formula. Thus, to avoid redoing computation, we memoize tactics related to Step commands, and reuse a tactic rather than rebuild it.

  It is unlikely that the tactic we obtain at the end of the compilation process will succeed to prove a different formula, since the tactic is purpose-built to prove the conjecture.

- *Tactic translator.* This maps a cut program into a sequence of tactics, corresponding to the commands making up the program. These tactics are then applied in sequence to an Isabelle goal (the conjecture) in order to yield a theorem, as in the description of the tactic compiler.

Both options described above are *safe*: when executing the program, the use of inferences is checked by Isabelle's trusted kernel; it acts as the final arbiter of validity, and protects against any bugs in the implementation. Compiling is the more efficient of the two, but having both available is useful, particularly for debugging.

Partial proof-reconstruction. Recall that proof reconstruction can fail if one of the inference rules cannot be emulated. Partial proofs are allowed in the framework, and this feature

[4] This contrasts with *computational reflection*, where inference can be done using additional meta-logical rules added to the calculus. Harrison (1995) discusses the LCF approach and reflection in detail.

is used for testing the emulator. It is possible to extend this feature to end-users, to allow them to help the reconstruction process by manually proving meta-theorems that could not be emulated.

5.7 Implementation

The code of the implementation has been made available online.[5] The preceding sections described the design of the framework and its components, and how it was implemented to import Leo-II proofs into Isabelle/HOL. In this section we look at some limitations of the implemented prototype:

1. Currently we do not handle the rules in Leo-II's calculus that deal with the Axiom of Choice. AC-support is a relatively new feature in Leo-II (Benzmüller and Sultana, 2013), but it only involves instantiating AC. Reconstructing AC-related Leo-II-inferences in Isabelle/HOL is a natural extension to this work.

2. Recall that Leo-II collaborates with E to find a refutation. Currently the framework described in this chapter is not used to reconstruct the contributions that E makes to Leo-II. Instead, E's proofs are re-found by using Metis.

   It sometimes happens that Metis cannot re-find E's proofs. As described in the next section, the success on our evaluation benchmark increases from 82.1% to 93.8% if we treat E as an oracle instead of using Metis to re-find its proofs. That is, if we assume that E was sound, and that we had perfect reconstruction of E's proofs, we could fully reconstruct 14.3% more proofs in this benchmark.

   A different approach to reconstructing E's proofs would involve the tableau embedding, described in Section 4.4. There is no obvious reason why this should not work with the current version of E (1.8). Implementing this would consist of completing the prototype described in Section 4.4.2. Using this method, we would map E's TPTP proofs into Isar, then we would process the Isar script in Isabelle/HOL to obtain an Isabelle/HOL theorem from E's theorem, and finally use this as a lemma within the proof being imported from Leo-II.

   Alternatively, we could use the framework described in this chapter to import E's TPTP proofs directly into Isabelle/HOL. At present, this is made more difficult since E often returns proofs containing compound inferences, called *derivation stacks*. These are similar in principle to Leo-II's extcnf_combined (cf. Section 3.1.1), and E currently does not support the expansion of derivation stacks into primitive inferences. In principle, we could reconstruct E's proofs without expanding them into primitive inferences, but, as discussed in Section 3.1.1, we would need to carry out proof-search to validate the compound inferences.

5.8 Evaluation

To evaluate the proof importer, a set of test proofs was first obtained by running Leo-II 1.6 for 30 seconds on all THF problems in the TPTP 5.4.0 problem set. In these experiments, Leo-II cooperated with version 1.8 of the E theorem prover. Leo-II produced 1537 proofs.

[5] `http://www.cl.cam.ac.uk/~ns441/files/recon_framework/`

The translator was then run with a timeout of 10 seconds. By using Metis to emulate E, 1262 (82.1%) proofs were reconstructed entirely. If we treat E as an oracle (i.e., assuming E is sound, and that we have a perfect reconstruction for its proofs), then the number of reconstructed proofs increases to 1442 (93.8%). Currently, Sledgehammer reconstructs Leo-II proofs by refinding using Metis and Z3, as described in the next section. On the same problem set, Metis and Z3 were able to reconstruct 57.3% and 68.9% of the proofs respectively. This is described in more detail in the next section.

Note that these proofs were reconstructed into theorems, not only as closures (as in the lazy-theorem style described in Section 5.6). Full results and scripts are available online.[6] A repository version of Isabelle2013 was used, and the experiments were done on a 1.6GHz Intel Core 2 Duo box, with 2GB RAM, and running Linux.

The failures were analysed, and they could be classified into three categories:[7]

Syntactic. There were two such cases: one related to an esoteric node-naming feature of TPTP (that only appears in a syntax-testing problem), and the other related to a bug in Leo-II's proof output.

Partial reconstruction. There were 12 cases of this. Half of these cases appeared to be problems in the implementation of inference emulation. Increasing the unification depth improved completeness but hurt performance. The other half of these cases appeared to arise from alignment problems when a proof was translated to the intermediate language which was used to represent cut programs. On occasion, one of the instructions was too strong, and reconstructed more than one step of the proof; this caused the subsequent translation steps to fail.

Timeouts. There were 81 cases of this. The most likely cause of a timeout is problem size. All of these problems are fairly big: ordered by file-size, the first timeout occurs at the 1088th problem, and the second at the 1432nd (out of 1537). That is, if we make a list of all problems, and order the list by problem size, then the timed-out problems are clustered at the end of the list. This suggests that the time-out occurs because of the problem's size, which is in line with expectation.

Comparison with existing reconstruction

The implementation described in this chapter was compared to re-finding the proofs using Metis (Hurd, 2003) and Z3 (De Moura and Bjørner, 2008)—these are the means currently used by Sledgehammer to reconstruct proofs found by external provers (Paulson and Blanchette, 2010). These experiments used the versions of Metis and Z3 that are packaged for Isabelle2013.

Metis and Z3 were applied to the same problem files that were given to Leo-II. Before running the experiments, a check was made to ensure that Metis and Z3 did not use any additional facts from the Isabelle library of lemmas, to avoid putting them at a disadvantage or advantage.

Since in this case proof re-finding did not benefit from the structure of the Leo-II proof, essentially this test involved using Metis and Z3 to re-find the proof in 10 seconds (which

[6] `http://www.cl.cam.ac.uk/~ns441/files/recon_framework_results.tgz`

[7] This analysis was carried out on the results of the experiment in which E was treated as an oracle, to focus on the reconstruction of Leo-II inferences.

	Judg. Day	Arith. Ext.
Leo-II	98.4	97.9
Satallax	97.5	97.9
E	97.2	98.2

	Judg. Day	Arith. Ext.
SPASS	99.1	99.6
Vampire	98.7	97.5
Z3	99.9	95.3

Table 5.1: Success rate (%) of proof reconstruction, from earlier experiments (Sultana et al., 2012). This set of experiments was carried out by Jasmin Blanchette.

Leo-II might have taken 30 seconds to find). As a result, Metis and Z3 reconstructed 57.3% and 68.9% of the proofs respectively.

It is unsurprising that a purpose-built proof importer outperforms proof re-finding. On the other hand, in earlier experiments we were surprised at the effectiveness of proof re-finding. Table 5.1 shows the success rates of reconstruction (using Metis and Z3), but in that work we did not try proof reconstruction on TPTP problems. Table 5.1 involves problems from the Judgement Day and Arithmetic Extensions benchmarks (Böhme and Nipkow, 2010), which had been used to evaluate Sledgehammer in earlier work. The reconstruction evaluated in Table 5.1 benefitted from the library of lemmas formalised in Isabelle/HOL, and made available by Sledgehammer. This was not the case in the TPTP experiments in the current evaluation. Also, Isabelle-based benchmarks have long been used to evaluate Sledgehammer (including reconstruction through re-finding), therefore Sledgehammer-related functionality has a better chance of performing better on these benchmarks. Notwithstanding this, it is impressive that Z3 can reconstruct 99.9% of the problems in the Judgement Day benchmark!

Conclusion

In this chapter we looked at a framework for reconstructing proofs, targeting Isabelle/HOL. It separates the analysis and transformation of proof structure from the emulation of inferences. There is a dependency between the two—after all, the emulation will ultimately process any synthetic inferences added during proof transformation—but the two can be encapsulated into separate modules.

A program is produced by the analysis and transformation phase. This program guides proof reconstruction, and will yield a theorem when executed. The emulation phase assists in the execution of such a program. This chapter described the core features of the language of such programs, and the semantics of its instructions.

Finally, we saw how to apply this framework to reconstruct Leo-II proofs in Isabelle/HOL. This was implemented and evaluated.

Chapter 6

Related work

The translation of proofs is of both theoretical and practical interest, and has received plenty of attention over the years. In this chapter we survey related work. Work that uses the same source or target logics mentioned in previous chapters is described in more detail, but it will be framed in a broader context to include other logics and approaches. We will also expand on the survey of related work given in Section 2.3.3.

6.1 Outline of general approaches

There are various strands of research that examine proof translation in a general, abstract setting. This is often done for at least one of three reasons: to minimise the commitments to the source and target logics; to argue for the embedding of various logics inside a host logic; or to encode proofs in an expressive common language, as *proof certificates*.

The distinction between these three is not clear-cut—indeed, there are significant overlaps. Proof certificates are themselves expressions in a logical language. Their well-formedness affects their validity as proofs. Thus the language in which they are encoded is implicitly being used to interpret other logics. This encoding involves translation, and the translation is often, but not always, described using a general programming language. We will start by looking at logical frameworks, which can be used to describe logics, and translations between logics.

6.1.1 Meta-logical frameworks

The first category of research we will survey involves meta-logical frameworks. One such system was Logosphere (Schürmann and Stehr, 2006). It relied on embedding logics in the Twelf meta-logical framework (Pfenning and Schürmann, 1999), which is based on LF (Harper et al., 1993). If a relation formalising the translation between the embedded logics can also be described in this framework, then this could be the starting point for a verified implementation of the translation between the embedded logics. Such a setup could also be used to check proofs, by checking the the proof terms to be well-typed. This has been proposed for checking proofs in THF0 (Benzmüller et al., 2008a). Since different theorem provers usually use different calculi, one would have to encode the inferences of each THF0 prover of interest. Systems producing TPTP proofs can use arbitrary inference rules. Recall the critique that TPTP does not fix a language of inferences in Section 2.5.

Another approach is based on the treatment of logics as mathematical theories, and uses a (semi-)formal language, such as OMDoc (Kohlhase, 2006), to describe mathematics. This language is used to describe logics and relations over logics. Rabe (2008) gives a detailed account of this approach, where he uses modules to describe mathematical theories and implementations of theories to describe morphisms between theories. The character of this research is algebraic, relying on a map (called the *signature morphism*) from one theory's signature to another theory, and classifying this as being a *theory morphism* if theorems are preserved. The preservation of theorems means that the logical entailment relation, between sets of formulas and single formulas, is preserved by the morphism.

This thread of work is still active, and Rabe et al. (2011) build on it to describe a general approach for integrating systems. The approach of Rabe et al. is based on translations between mathematical theories. As in earlier work, theories are structured using module systems, but the integration problem is described in terms of answering queries—as in logic programming. Rabe et al. explore the idea of *untrusted communication* between mathematical assistants, including proof assistants and computer algebra systems. They discuss different foundational commitments, using an example of the encoding of Peano arithmetic in ZF set theory and in the Calculus of Inductive Constructions. Following common practice, they organise systems into layers, with a lower layer serving as a meta-language to a higher layer (called the object language). They give prominence to *faithful* translations, which map "encodings of a mathematical notion to the corresponding encoding of the same notion." They tackle the problem of foundational commitment by encoding both meta and object languages in a common language, which they call MMT (a Module system for Mathematical Theories), a sub-language of OMDoc. They also generalise the notion of theory morphisms to *partial* theory morphisms, for cases when not all of the objects in one theory can be mapped to another.

The work in this area tends to focus on the high-level combination of logics, and this affords it conceptual elegance. Engineering details are abstracted away, delegated to automated support for parsing and interpreting mathematical content.

The Heterogeneous Tool Set (HETS) by Mossakowski et al. (2007) is a tool related to the ideas described in this section, and in which "logic translations (e.g. codings between logics) are first-class citizens".[1] Mossakowski et al. claim that the difficulty of adding a new logic to HETS "can be compared to embedding the new logic in a HOL prover", and once a logic has been added, one can then formalise the translation to other logics. In this dissertation we narrowed our focus to a simpler setup: we shallowly embedded logics in Isabelle/HOL, and formalised the translation in ML. Hence, unlike in HETS, our translations are not first-class. The intermediate language described in Chapter 5.4 can be reused for different source logics, but each logic needs to be shallowly embedded in Isabelle/HOL. Another difference is that HETS does not carry out any checking of proofs built by external tools. Other tools can be used to check for consistency, but their output is not checked either. Recent work integrated HETS with Twelf, and this is intended to eventually use Twelf for proof-checking, if the combined proof tools produce LF proof terms (Codescu et al., 2010).

[1]Note that *heterogeneous* reasoning could also refer to a mode of reasoning that uses different *representations* of formulas and proofs—for example, by combining diagrammatic reasoning with traditional logical syntax (Barker-Plummer et al., 2008; Urbas and Jamnik, 2012). That sort of heterogeneity is beyond the scope of this dissertation.

6.1.2 Host logics

The research in this vein has similar concerns to those of the work described in the previous section, except that the host language tends to be a more established mathematical theory such as set theory. Unlike meta-logical frameworks, the host logics in this section do not usually allow us to encode relations between systems in the language. Instead we would need to describe the translation in a meta-language, such as a programming language (or some other containing theory).

Logical monism is the idea that all logics can be encoded into a single logic. This is similar to Leibniz's idea for a "characteristica universalis" (Harrison, 2008). Hintikka (1998) and Robinson (2000) seem to take a *pragmatic* stance on monism. They suggest that to carry out mathematical development, all one needs is FOL, provided that it is suitably extended by additional axioms, such as those for ZF set theory.

Instead of using ZF set theory, another suggestion is to use simple type theory as the host logic (Church, 1940). Simple type theory is too weak to directly interpret logics such as those of Twelf or Coq, for example, but there are many other interesting logics that can be embedded in simple type theory. This is the basis of the work by Benzmüller (2010). Benzmüller and Paulson (2013) explore the embedding of modal logics in simple type theory. The present dissertation too is aligned to pragmatic monism, since we rely on the interpretation of the source logics in Isabelle/HOL, which is based on simple type theory. The ΩMEGA system (Siekmann et al., 2006) was also based on simple type theory. It sought to internalise proofs from various systems, representing them in a common language. ΩMEGA was pioneering in several respects, and we will encounter it again later in this chapter.

Keller (2013) uses an extension of the Calculus of Constructions (as implemented in Coq) as the host logic for theorems proved by SAT and SMT solvers. Essentially, she develops a trusted SMT proof checker that is then used to check proofs produced by other SMT solvers, or to interpret those proofs in Coq's logic. Keller's trusted checker is extracted from a constructive proof in Coq. This idea has been explored in the past for developing trusted implementations of proof procedures (Underwood, 1990; Weich, 1998). Essentially, Keller's checker follows the design of an SMT solver: it mediates between theory-specific solvers to refute a conjecture. The theory-specific solvers in Keller's work consist of theory-specific decision-procedures implemented using Coq. In order to use Keller's system, a proof from an SMT solver must first be translated into a form that can be processed by the trusted checker. This is done as a preprocessing step, and this realises an embedding of the source calculus in the target calculus. The pure logic component of SMT is identical to that of SAT: proofs consist of refutations expressed using resolution. This means that the pure logic component of Keller's system is much simpler than the systems developed in this dissertation: for one thing, SMT lacks quantification. In a related respect, Keller's system is more complex than the systems developed in this dissertation, since Keller's system supports a range of theories—as is expected in SMT. Currently, the state of the art in higher-order ATP systems does not interpret any theories other than equality.

Recently Liang and Miller (2011) have devised a logic that can be specialised to classical, intuitionistic, and linear fragments by polarising literals, connectives and, as a result, formulas. In addition to this, structural rules—which, in the sequent calculus show up a distinction between different logics—are sensitive to these polarities. Thus for instance, one could test whether a formula is classically, but not intuitionistically, valid. Within this setup, intuitionistic logic appears as a hybrid of classical and linear logics. Deductions can be used across logics by using the cut rule. This work was the basis for research on *foundational proof*

certificates, which are described in the next section.

6.1.3 Certificate-oriented languages

While the desire for a shared language for mathematical truths goes back centuries, the degree of variety within logic alone (never mind other areas of mathematics) makes it very difficult to define such a language.

Recall the critique that TPTP does not fix a language of inference rules, from Section 2.5. By fixing a set of inference rules, we would be implicitly defining a logic, which could host proofs. Fixing a logic often involves making a foundational commitment, which tends to exclude other kinds of foundations. This is rather contrary to the spirit of combining logics and systems, which seeks to be inclusive.

As mentioned in the previous section, despite its eponymous simplicity, simple type theory is not ideal as a universal language of proofs. Proposals for alternatives have emerged from research in type theory and proof theory.

The first is $\lambda\Pi$ *calculus modulo* (Cousineau and Dowek, 2007; Boespflug et al., 2012) in which proofs from pure type systems can be encoded. Many high performance ATP systems are based on resolution and superposition, and it was recently shown that such proofs can be embedded in $\lambda\Pi$ *calculus modulo*, thus allowing the proofs to be checked by a type checker (Burel, 2013).

The second approach relies on the LKU logic mentioned in the previous section (Liang and Miller, 2011). Chihani et al. (2013) describe an architecture based on higher-order logic programming for checking *proof certificates*. These certificates can take different forms, depending on the proof calculus of the source logic they relate to. Checking these certificates involves interpreting them into a form that can be checked as an LKU proof. In particular, Chihani et al. give examples of certificates for proofs given in a matings calculus (Andrews, 1981) and in a resolution calculus.

These approaches are powerful and elegant, but neither, to my knowledge, has yet been applied to an independent ATP system. The approach taken in this dissertation involves translating proofs into HOL; the systems described in this section use more sophisticated logics. These logics ultimately play the role of a host logic.

I will dissect the work by Chihani et al. in more detail, owing to its scope and its relation with my work.

First, they differentiate between *foundational* and *technological* approaches, and concentrate on the former. Essentially, a tool is technology; Chihani et al. focus on deeper, more unifying features that can transcend technologies. In this setting, *foundational* seems to be synonymous with prover- or logic-agnostic, and it is achieved by focussing on proof theory. My work leans more towards technology, but not entirely—as can be seen by the contents of Chapter 4.

Second, they argue for using λProlog to host their implementation, and describe the good fit between what it offers and what is needed to achieve their research objectives. λProlog is based on simple type theory, which has a mature (well-understood) meta-theory, and which underlies Isabelle/HOL too. Chihani et al. use λProlog in a very similar way to how Isabelle is used in my work. λProlog is used as a system in which to specify logics and carry out reasoning. Chihani et al. base their work on an encoding of LKU in λProlog. I base my work on an encoding of HOL in Isabelle, but this is rather incidental; λProlog would make an excellent, if not more elegant, platform on which to implement my work.

Essentially, Isabelle is an ML library for implementing logic tools, while λProlog serves as a language for implementing tools using logic.

Third, Chihani et al. use LKU as a host language. They argue that other logics can be embedded in LKU, by disabling some of its rules—thus constraining LKU, specialising it into classical, intuitionistic, or multiplicative-additive linear logic. I use HOL instead of LKU; HOL cannot be modified in the same way that LKU can, but HOL is also much simpler to use—LKU has four polarities! Both LKU and HOL are used in the spirit of pragmatic monism; either host logic might not be suitable to encode (or capable of interpreting) all logics, but it is chosen as a *good enough* system on account of its features.

Fourth, the design of Chihani et al.'s system consists of the following components:

Kernel. This consists of an implementation of LKU's proof system in λProlog. The implementation benefits from features offered by λProlog, such as declarative specification of the calculus, abstract datatypes for theorems, etc. In my work, this corresponds to the encoding of HOL using Isabelle, which in turn benefits from features of ML.

Client. This is the proof producer: the ATP system. It produces a proof in some format chosen by the authors of the ATP system. That is, Chihani et al. describe an importer-style system, similar in spirit to the system described in Chapter 5.

Clerks and Experts. These are two types of agent-like functions that carry out proof construction in LKU. They correspond to the two phases of proof-construction in a focussed proof system. This component serves to interpret the proof certificate into an LKU proof, that can then be checked by the kernel.

Essentially this component constitutes an embedding of the source logic into a fragment of LKU. The clerks and experts seem to carry out a similar role to that of Keller's preprocessor, described above. (Recall that a different preprocessor might be needed for each ATP system whose proofs we want to import.) In my work, this role is carried out by the emulations of inference rules, which was described in Section 5.5.

Using Keller's approach does not require you to describe the embedding in the host system. In contrast, the approach taken by Chihani et al., and by me in this dissertation, do require this. In the case of Chihani et al., clerks and experts are described using λProlog clauses, while in my case the embedding is described as a collection of tactics—one for each inference rule in the source calculus. This difference seems superficial, since it arises from the different frameworks we used to implement our languages (λProlog and Isabelle). It will become possible to compare these approaches when more details about Chihani et al. become available.

6.2 Handling TPTP proofs

Plenty of support exists to translate *problems* into TPTP format. This has been found to be useful by users of proof assistants, providing them with automated proof support (Paulson and Blanchette, 2010; Rudnicki and Urban, 2011). The TPTP format is also used by software verification tools, to encode proof goals dispatched to ATP systems (Bobot et al., 2011; Bove et al., 2012).

There has been work to analyse TPTP proofs (Sutcliffe et al., 2010), and synthesise such proofs (Sutcliffe et al., 2011) from other TPTP proofs. This involves translating the TPTP

proofs into a richer representation in which to carry out the analysis. The TPTP infrastructure contains tools for translating problems from various formats into TPTP, but it does not contain tools for translating proofs.

Despite the importance of TPTP proofs, there is a dearth of stand-alone tools that translate TPTP proofs into other formats. There are however translation facilities within other proof tools, to provide a facility to *import* proofs. TPTP proofs serve as an intermediate format between proof representations in different proof tools. For instance, Isabelle/HOL supports the translation of TPTP proofs into Isar (Wenzel, 2002) format (Paulson and Susanto, 2007; Smolka and Blanchette, 2013). The next section includes a description of the different ways in which TPTP proofs are imported into Isabelle/HOL.

6.3 Isabelle/HOL-targeted proof translation

Following LCF and the HOL system (Gordon and Melham, 1993), proofs in Isabelle were originally constructed entirely by using *tactics* at the ML level. Tactics are functions that simplify proof goals, ultimately reducing them to tautologies. An example of an Isabelle tactic was given in Section 5.5.1.1. *Tacticals* combine tactics to yield new tactics with richer behaviour. For instance, tacticals could be used to organise the search through a space.

Isabelle and other LCF-style systems provide automatic proof-checking, but initially their proof-*finding* features were very spartan: they relied entirely on the user even for the most obvious or tedious of proofs. The support for automated proof-finding in Isabelle was improved when Larry Paulson started developing the family of tactics that have come to be known as the *classical reasoner* in Isabelle. These were developed in two generations. This was followed by a third generation of development, during which the Sledgehammer tool was built to interface with external ATP systems. These developments will be described next.

The first generation (Paulson, 1997) of tactics (`fast_tac`, `Fast_tac`, `best_tac`, etc) worked directly on Isabelle's reasoning engine. These were generic—in the sense that they could be supplied with arbitrary rules, using which they could attempt to build proofs. The rules were given in natural deduction form. This allowed using the proof-theoretic semantics approach: giving meaning to symbols in terms of introduction and elimination rules. Other than being generic, another advantage of this approach was its potential to yield high-level proofs, in the sense that the proofs were built using the rules given, rather than their expansion into the host logic. Another advantage was that, since rules were applied in a syntax-directed manner, the characterisation of symbols via their rules allowed the simulation of a lazy unfolding of definitions. This last benefit is significant when large (and redundant) definitions are contained in a formula.

These rules were used by a custom-built tableau prover that conducted the search directly on Isabelle's Prolog-like engine. The downside to this approach was performance. A possible reason for this was that Isabelle's internals had not been designed for proof search. The internals, and the tools built on top of them, might be difficult to use directly for specialised uses. For instance, in the first work on integrating an SMT solver with Isabelle (Fontaine et al., 2006), the authors decided not to rely entirely on automation provided by Isabelle. They implemented specialised tactics instead of using the simplifier.

Another disadvantage was that the handling of quantifiers was incomplete: only one instantiation was permitted. This incompleteness was deliberate, to target *obvious* proof goals, as defined by Martin Davis. Paulson (1997) describes this motivation in detail.

Since the proofs were built directly using Isabelle's reasoning engine, no proof translation was necessary.

The *blast* tactic (Paulson, 1999) is a second-generation proof-finding tool in Isabelle. It is a tactic that performs proof-search independently from Isabelle. When a proof is found, it is reconstructed in Isabelle by passing Isabelle a list of tactics. Paulson describes settings where reconstruction can fail.

As far as quantifier handling is concerned, this approach is complete (unlike the previous generation of tactics). However, Paulson argues that completeness of proof-search is of secondary importance when used in a proof assistant. Moreover, he argues that soundness is also not paramount either, as long as a proof is then checked by Isabelle's kernel. This rationale places productivity of ATP systems above their soundness and completeness. The measure of success is performance in practice, rather than fulfilment of theoretical conventions. This rationale was also used later in Sledgehammer (Meng and Paulson, 2008), and was also shared by Hurd (2003) when describing the type encodings used in Metis.

Blast can be given arbitrary sets of tableau rules to reason with, making it usable for theories beyond pure logic. Paulson reports that integration with Isabelle constrained which refinements could be used in blast.

Similar in spirit to Paulson's approach, Asperti and Tassi (2007) describe the integration of a purpose-built superposition prover with Matita. Asperti and Tassi do not address the full (superposition) calculus, and focus on unit equalities. They try to improve the readability of proofs found—for instance, by translating them to natural language.

We now come to the third generation of development. Proof search using Isabelle (Paulson, 1997, 1999) is very often outperformed by purpose-built, standalone tools. Sledgehammer (Paulson and Blanchette, 2010) was developed to outsource proof-search to external, specialised tools. Initially Sledgehammer targeted first-order provers such as E, SPASS and Vampire, but now the list includes SMT solvers (Blanchette et al., 2011) and higher-order ATP systems too (Sultana et al., 2012). Various aspects of Sledgehammer are described in the dissertation by Blanchette (2012). The approaches to proof translation in Sledgehammer are described next:

- Early experiments involved parsing a proof produced by SPASS (Weidenbach et al., 2009) and reconstructing it in Isabelle/HOL as an Isar proof script (Meng et al., 2006, §8). This was very difficult, as "SPASS used many inference rules that differed slightly from one another. The information it delivered was incomplete, omitting details such as which subterm of which literal had been rewritten. Finally, SPASS re-ordered the literals in the clauses it was given." (Paulson and Susanto, 2007).

  This difficulty is a common lament when reconstructing proofs (Meng et al., 2006; Hurd, 1999; Weber and Amjad, 2009; Dunchev et al., 2012). It arises because proof reconstruction needs to rediscover information that was not made explicit in the proof's representation—such as the syntactical changes described by Paulson and Susanto above. The reconstruction process needs to work around idiosyncrasies, and emulate (often poorly documented) proof steps of varying complexity. This was discussed at length in Section 3.1.1.

- A more successful approach involved using the external ATP system to filter out axioms that were not used in the proof, then try to *re-find* the proof using Metis, which was ported to Isabelle/HOL.[2] This involved extracting an UNSAT core—i.e., the formulas

[2] In the sense that its inference rules were interpreted in Isabelle/HOL, and that any theorem proved by

actually used to derive a contradiction—from the proof found by the external ATP system.[3] This set of formulas was then given to Metis to prove inconsistent.

This approach was limited by the strength of the Metis (first-order) prover. Metis only supports the theory of equality, while more modern ATP systems have dedicated solvers for more theories. This approach is also limited to refutation proofs, but these are not generally usable by systems implementing non-classical logic, such as Agda. Foster and Struth (2011) describe such an integration.

- In order to help overcome the limit of Metis' strength, described above, the method was improved as follows (Paulson and Susanto, 2007):

  1. The TPTP output of the external ATP system was parsed, and each proof step mapped back to an Isabelle/HOL formula. This required mapping type-classes, types, definitions, and axioms mentioned in the proof back to the ones in Isabelle/HOL from which they originated. The result of parsing the proof is a tree of Isabelle/HOL formulas, each node annotated with (Isabelle/HOL) axioms used in the derivation. This tree can be seen as the structure of the proof found by the ATP system; each step can be regarded as a hint, or waypoint, towards the final refutation step.
  2. Metis was called to justify each proof step, given the relevant axioms determined from the ATP system's proof (i.e., the annotation in the proof tree). This was more successful than giving Metis the entire UNSAT core, since this method helped refine the granularity of Metis' proof search. Similar methods, albeit using different provers, were used earlier in Otterfier (Zummer et al., 2004; Sutcliffe et al., 2005) to check proofs using Otter (McCune, 1990), and more recently in GAPT (Dunchev et al., 2012) using Prover9 (McCune, 2005).
  3. Instead of using Metis to justify each step, the resulting proof script could be shortened by working modulo some number of steps. This bunches together intermediate inferences. It is more difficult for Metis to validate these inferences, and the resulting proof is not guaranteed to be clearer to understand, despite being shorter.

  The parsed TPTP proof was shown to the user as an Isar proof script (Wenzel, 2002). The user could inspect it, and check it using Isabelle/HOL. Checking the proof involved using Metis to validate the proof's inferences, and chain them together. The Isar proof script is human-readable, though it is often only understandable by very determined humans.

  Although not presented as such by Paulson and Susanto, the Isar proof script could be regarded as a *proof certificate*, and used to reconstruct the proof later on without re-invoking the source ATP system. Its downside, as with all certificates, is its brittleness to upstream changes made to Metis, for example, that might result in the certificate being no longer checkable in the future.

  Metis was chosen as the medium to interface between Isabelle and ATP systems for the following reasons:

Metis could then be regarded as a theorem in Isabelle/HOL.

[3] Note that the UNSAT core must contain the negated conjecture, otherwise it would mean that the problem's hypotheses are inconsistent.

- Metis was designed for a closely-related purpose: just as the *blast* tactic was a first-order ATP system designed for integration with Isabelle, the first-order prover Metis (Hurd, 2003) had been conceived to improve proof-search in the HOL4 proof assistant;
- it has a small calculus consisting of only five rules;
- it has an LCF-style kernel in which, conceptually, its primitive inferences are elaborated into those of Isabelle to yield an interpretation;
- on the practical side, it is distributed under a permissive license, it is open-source, and, like Isabelle, it is written in ML.

These reasons would suggest that Metis should be easier to integrate with Isabelle/HOL. However, it proved surprisingly difficult to integrate (Paulson and Susanto, 2007, §3). Recently, Kaliszyk and Urban (2013) have implemented proof reconstruction for HOL Light, partly based on the approach taken by Paulson and Susanto (2007).

- A further improvement was made to avoid Metis being flummoxed by large UNSAT cores. This involved minimising the core, and was reported to be beneficial in benchmark tests (Böhme and Nipkow, 2010).

- Sledgehammer has also been extended to use SMT solvers (Blanchette et al., 2011). In addition to using Metis for reconstruction, the proof can be re-found by Z3 (De Moura and Bjørner, 2008) then reconstructed (Böhme and Weber, 2010). Z3 proofs can also be stored as certificates, to reconstruct the proof later on without re-invoking Z3.

  Böhme and Weber (2010) describe four categories of reconstruction methods:

  1. Using a single primitive inference or schematic theorem to interpret an inference rule. This is the cheapest option, since there is no search involved if an ATP system's inference can be mapped directly to (an instantiation of) a primitive inference or theorem of Isabelle/HOL.
  2. A combination of the above—that is, a collection of primitive inferences and schemes that emulate an ATP system's inference. This approach is more expensive since it involves search to yield a sequence of instantiations. This method is useful when an ATP system's inference rule does not neatly fit what can be encoded as a rule in Isabelle/HOL, but it is also fairly tidy since the search space is expressed declaratively as a finite set of schematic formulas.
  3. A proof procedure. This involves search, therefore it adds both power and expense. The description of the search space might not be as concise as in the previous approach.
  4. A combination of the above—that is inferences, schemes, and tactics.

  The reconstruction of Z3 in Isabelle/HOL is described further by Böhme (2012) in his dissertation.

- There is an interesting line of work on trying to improve the readability of machine-generated proofs. This is of longstanding interest in the theorem proving community, since the contents of machine-generated proofs are often not human-intelligible. This could be alleviated by making proofs look "natural", as discussed by (Calude and Müller, 2009).

Blanchette (2012) described a technique, based on contraposition, which transforms refutation proofs generated by ATP systems into direct proofs encoded in Isar. Smolka and Blanchette (2013) describe further enhancements that improve the reliability of Sledgehammer. These include simulating the Skolemisation done by the (external) ATP system, constraining Sledgehammer's output to ATP systems using type annotations, preplaying proofs to filter out non-proofs, and shortening proofs by fusing together consecutive inferences in a proof.

Over the years there have been several proof translations involving LCF-style systems (Denney, 2000; Naumov et al., 2001; McLaughlin et al., 2006; Obua and Skalberg, 2006; Gordon et al., 2006; Kaliszyk and Krauss, 2013; Keller, 2013). Some of these translations involve ATP systems, while others involve solely proof assistants. When translating proofs from proof assistants, the chances of automatically re-finding (using an ATP system) the whole proof are vanishingly small if finding the proof required human guidance in the first place. For this reason, translations between proof assistants usually involve an interpretation of one system's inferences inside another system.

Kaliszyk and Krauss (2013) described a translation from HOL Light (Harrison, 1996) to Isabelle/HOL. They benefit from lessons learned from the work of Obua and Skalberg (2006) to obtain much better performance. Keller (2013) compares the proof recording format used by Obua and Skalberg (2006) (when translating HOL Light and HOL4 formalisations into Isabelle/HOL) with the more recent, and actively developed, OpenTheory format (Hurd, 2011). Both formats were examined by Kaliszyk and Krauss (2013) in their work, but they opted to use a new custom format. Traces in this format were produced by instrumenting the HOL Light kernel. These traces were preprocessed to facilitate their subsequent interpretation in Isabelle/HOL. The performance of the resulting system, called Import, is very impressive, and is described in detail by Kaliszyk and Krauss. Import also provides better support for mapping concepts, such as numbers, across systems. This kind of map is often important in proof translation (Rabe et al., 2011).

6.4 Hybrid proofs

It appears that JProver (Schmitt et al., 2001) is to the MetaPRL logical framework (Hickey et al., 2003) what the *blast* tactic (cf. Section 6.3) is to Isabelle. JProver uses MetaPRL as a library. Unlike blast, JProver was designed with a sophisticated interface, intended to facilitate the integration of JProver with other systems; blast was mainly intended to serve Isabelle. JProver is an ATP system for first order logic, and Schmitt et al. (2001) briefly describe its integration with NuPRL, a proof assistant based on extensional type theory (Constable et al., 1986).

JProver's JLogic module serves to interpret the source logic in terms of JProver's, and to interpret JProver's inferences in terms of the source logic's. Thus proof goals can be transferred to JProver, and proofs can be communicated back to the source prover. This facilitates the development of *hybrid proofs* (Schmitt et al., 2001, §4): "proofs created by multiple provers with different formalisms".

The LEO prover (Benzmüller, 1999) was originally intended to serve as an automatic reasoning engine for the ΩMEGA system (Siekmann et al., 2006), a proof assistant based on higher-order logic. Compared to other proof assistants, one of the distinguishing features of ΩMEGA was *proof planning* (Bundy, 1996): the application of planning to theorem proving.

ΩMEGA integrated various mathematical tools, and kept a central representation of proofs which could be modified by these tools. This representation was called PDS (for *proof plan data structure*). The proofs generated by external tools could also be represented in the PDS, after suitable transformation. Thus, ΩMEGA's logic would interpret the proofs found by external tools—similar to how we use Isabelle/HOL in this dissertation.

Later work sought to examine the scope for cooperation between LEO and first-order theorem provers (Benzmüller et al., 2008b). This relied upon the ΩANTS agent architecture, where agents could be proxies for external proof tools. The agents could access a so-called blackboard, which provided shared state between the various systems. Each agent would have bounded resources, using which it could try to advance the solution of a problem on the blackboard. If an agent managed to find a solution within the allocated resources, it could place a bid to ΩANTS. Bids were ranked. Unsuccessful bids were stored, to facilitate backtracking should it be needed in the future. This setup still relied on translation into a central proof format, for proof checking.

LEO-II takes a slightly different approach. It does not yet implement the sophisticated blackboard infrastructure mentioned above, but it relies on external ATP systems in order to function. The proofs returned by these tools are not checked by LEO-II, nor are they translated. Instead, they are spliced with LEO-II's proof, and left for an external tool to check (Sultana and Benzmüller, 2012). Because of LEO-II's reliance on external proof tools, the proofs it produces tend to be hybrid proofs.

Chapter 7

Conclusion

In the first chapter of this dissertation I argued that the problem of interfacing logic tools through proof translation is interesting, timely, open, and important. The chapters that followed proposed solutions to this problem from different positions: at the source system which generated the proof (Chapter 3), at the target system which seeks to consume the proof (Chapter 5), and independently of the two systems (Chapter 4).

One of the novelties of this dissertation is the consideration of three different kinds of translators; usually research focusses on only one kind of translator. Indeed, initially it did not cross my mind to consider a system other than an importer, and, on the implementation level, to modify a system other than Isabelle—with which I was familiar. I noticed that most, if not all, of related research place more emphasis on a single system, and do not consider alternatives more liberally. Broadening my work to include other kinds of systems gave me new perspectives on the research problem.

Three contributions were made in this dissertation:

- A compiler-based translator was described in Chapter 5, described in terms of *cut machines*: a theoretical model of a proof translator as a proof builder.

  This idea was implemented as a proof importer in Isabelle/HOL, to reconstruct proofs that were generated by Leo-II. On the TPTP THF benchmark, this improved the reconstruction success in Isabelle/HOL from 57.3% (using Metis) and 68.9% (using Z3) to 82.1%, as described in Section 5.8. The implementation of this system closely followed its specification, and its modular construction should facilitate extension and improvement.

- Two translations of refutation proofs into validity-preserving calculi were described in Chapter 4. Most ATP systems work by refutation, and the two proof translations can be applied to a broad class of such systems. The translations were proved sound and complete, and their complexity was characterised. One of these translations was used in the implementation described in the previous contribution.

- Two ideas were presented in Chapter 3. First, we saw how modifying an ATP system could improve the time complexity of certain reconstruction steps, from factorial to linear. An evaluation of higher-order provers was used to motivate the need to modify an ATP system's proof output to improve reconstruction.

  Second, I extended the work of Teucke (2011) to evaluate the difficulty of targetting a new logic. The difficulty was low, but obtaining a complete translation was difficult

because of limitations in the target input language. I suggested two work-arounds for this. Despite the implementation's limitations, 77.3% of the Isar proofs produced by this extension could be checked fully automatically by Isabelle/HOL on the benchmark used.

As is normal in research, the work leading up to this dissertation involved formulating and chasing a horizon of hypotheses. Some hypotheses were upheld while others were not. I will conclude by recounting the main hypotheses, describe whether they held up, and explain the lessons learnt.

The first hypothesis I made was: given the similarity between the logics of Leo-II and Isabelle/HOL, mapping the proofs of Leo-II to be checked by Isabelle/HOL should not be difficult. This was both naïve and wrong: the similarity of logics is indeed a benefit, but the clarity of a proof's encoding is more important if we are to translate it.

This leads us to the next hypothesis: one should be able to handle the proofs that are produced by the source system *directly*—that is, without modifying that system to produce clearer proofs. As explained in Section 3.1.1, it was not practical to sustain this belief, because of the complexity incurred when reconstructing the proof.

A third hypothesis: it is beneficial to separate the structure of a proof from its inferences, to handle the two separately. That is, the inferences will result in admissable rules in the target logic, while the structure will form a plan for applying these rules to produce the translated proof. This hypothesis seemed like a sensible idea for structuring the proof information; indeed this idea had already occured in related work (Böhme and Weber, 2010). This idea turned out to be very useful in the framework described in Chapter 5, where it underlies how cut machines work.

After a couple of early prototypes of the importer described in Chapter 5, a new hypothesis dawned on me: the reconstruction framework should be structured like a compiler. After all, compilers are language translators, and proof translation involves language translation too. Until then, my prototypes were structured rather monolithically—they did not differentiate between analysis, transformation, emulation, and cut-program formation phases. (This monolithic organisation was partly due to my naïve belief that reconstructing Leo-II's proofs was a simple matter, which led me to think that the structure of the translator should be correspondingly simple.) This hypothesis held up during my research.

An hypothesis I borrowed from the community has held up well: that one can conveniently resort to re-finding in order to make proof reconstruction more robust, or to avoid the complexity of defining a full embedding, as described in Section 2.2. This has also been described in point 2 on page 118. I used re-finding to reconstruct well-defined parts of proofs—such as for extensional unification, or for emulating E, as described in Chapter 5. The key idea here is to make the required inference *shallow enough* to increase the likelihood that re-finding will succeed in emulating the desired inference.

This idea can also be seen in reconstruction methods that involve finite calculi, such as that of Teucke (2011) which was described in Section 3.2.1. There, proof translation involves carving out an adequate finite calculus (using which the theorem should be provable) then performing proof search using trusted tools—that is, tools for which we have a proof reconstruction function. In the framework described in Chapter 5, if the ρ parameter to a cut machine is finite, then we can carry out bounded proof search using the cut machine extended with ρ.

Implicit in this idea is another idea that has received increased attention during the last decade: that we can build better proof tools by combining existing proof tools. This idea was

described in Section 3.1.2. In the case of re-finding, we are using a (trusted) proof-finding tool to assist with proof reconstruction.

Another hypothesis was related to the extent to which the translator needs to replicate the source ATP system—this was discussed in Section 2.2. I believed that, at least, the translator needed to replicate the source calculus, but I learnt that this could be more than sufficient. As discussed in Chapter 4, proof *translation* differs from proof *search* and *checking*: there is no need to check or enforce side-conditions during proof translation. This is not usually the case during search, and certainly not the case during checking!

When we set out to solve a problem, if we are lucky enough to have different tools to choose from, then it is inevitable to ask the question: How do we pick the right tool for this job? I will conclude with an observation and a hypothesis relating to this, offered for further investigation.

First, the observation. While carrying out the work described in Chapter 4 I observed that just as you might pick a particular variation of an algorithm depending on features of the input data—for example, you might pick a particular sorting algorithm if the input is already almost-sorted—you could pick different tableau embeddings depending on the features of the calculus you are mapping. This observation was made in Section 4.5, in the context of calculi having mostly validity-preserving rules. Developing a family of calculus mappings could help ensure better performance for specific calculi.

Finally, the hypothesis. Early in the dissertation, in Section 2.3.2, we identified three ways of positioning translators: exporters, transducers, and importers. Which kind of translator is the most *practical* to implement in the *current circumstances*? By *practical* I mean an implementation that minimises the effort required to build a generic translation facility for an ATP system's proofs. The *current circumstances* are such that proof output from ATP systems often serves to help *represent* a proof, and is often not carefully designed to help *translate* a proof, as explained in Section 2.2. Moreover, there are no standards mandating how proofs should be communicated between systems. TPTP sets a standard for problem and proof syntax, but not for proof rules. As a result, different ATP systems encode ad hoc information within annotations in TPTP-encoded proofs. No two provers report the same information, or report the common information in the same way. In this respect, when used for proofs, TPTP is primarily of syntactic, not semantic, relevance. Having only a notation or syntax for proofs is not enough if we want to translate or check them. We also need to know the specification of inference rules used in that proof. This was discussed further in Chapter 6, which compared different approaches—all of which rely on host languages in which to encode proofs. Each of these approaches embeds into a chosen language, and they serve as de facto standards for (semantical) proof languages.

My hypothesis is that exporters are the best choice. This is because, in the current circumstances, it is easier to extend a tool *forward* (as in an exporter, to output to a different tool) rather than *backward* (as in an importer, to input from a different tool).

This hypothesis is based on the asymmetry between input and output languages in the current circumstances as stated above. Proof languages used for output, such as TPTP, are more intended to represent proofs, since they are not intended to be processed by a specific system. In contrast, proof languages designed for input, such as Isar, have stronger criteria: an Isar proof is worthless unless Isabelle can parse and check it. I think that the asymmetry between input and output languages favours the development of one kind of translator over another.

Independent of the current circumstances, this asymmetry seems to show up at a more

conceptual level. If we can apply Postel's Law—"Be conservative in what you send, be liberal in what you accept"—to the proof-translation problem, then an exporter need only be conservative but an importer must be liberal. That is, an exporter needs to ensure that the target system can process the proof, while an importer might have to reconstruct information ommitted in the input proof. Bearing in mind the experiences described in this dissertation, the task of the importer seems harder than that of the exporter.

In my experience, extending the work of Teucke (2011) to export Isar proofs was much easier than building the importer for Leo-II's TPTP proofs. This relates to the *practical* quality mentioned previously: using the approach of Teucke, an ATP system maintains a single representation of proofs, which is then pretty-printed into different target languages—such as those for Coq and Isabelle. In contrast, the importer implemented a sophisticated system for analysing and transforming proofs in different stages, to make missing information explicit, before reconstructing the proof.

This leads me to think that exporters might be the most practical kind of translator to implement in the current circumstances. It is not clear whether this hypothesis holds up, since writing an exporter might also require us to write some recipient-side code to automate the processing there, as discussed in Section 3.2.2. Also, owing to the tight coupling of an exporter with an ATP system, it might be more difficult to reuse exporter code, compared to adapting an importer to work with a different source ATP system. On the other hand, when building an importer we might have to modify the source prover anyway, as described in Section 3.1.1.

Current thinking seems to implicitly go against this hypothesis: almost all ongoing and related work on proof translation (described in Chapter 6) is focussed on building importer-style systems. It is not clearly documented in the literature whether this design choice is based on an impartial assessment of alternatives, or if it is influenced by the tools and frameworks that the researchers are already familiar with.

Perhaps further work can cast more light on whether this hypothesis holds up. Given the current circumstances, this hypothesis might relate to an accidental, rather than inherent, source of complexity when building translators. The accidental complexity might be lowered if the current circumstances change such that, for example, ATP system developers agreed on a suitable output proof language, or if they became incentivised to assist in the burden of developing an importer. This might make an importer a clearly better design for proof translation. One could also investigate whether this idea, even in the current circumstances, can be convincingly quashed in applications beyond the *translation* of proofs. That is, if one wants to do further processing over proofs from ATP systems, in some of the ways described by Andrews (2005) for instance, then an exporter might not be a suitable design choice.

Bibliography

Andrews, P. B. (1980). Transforming matings into natural deduction proofs. In *Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 281–292. Springer. (Cited on page 20.)

Andrews, P. B. (1981). Theorem Proving via General Matings. *Journal of the ACM*, 28(2):193–214. (Cited on page 114.)

Andrews, P. B. (2001). Classical Type Theory. In Robinson and Voronkov (2001), pages 965–1007. (Cited on pages 26 and 149.)

Andrews, P. B. (2005). Some Reflections on Proof Transformations. In Hutter, D. and Stephan, W., editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 14–29. Springer. (Cited on pages 20 and 126.)

Asperti, A. and Tassi, E. (2007). Higher Order Proof Reconstruction from Paramodulation-Based Refutations: The Unit Equality Case. In *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 146–160. Springer. (Cited on page 117.)

Avenhaus, J., Denzinger, J., and Fuchs, M. (1995). DISCOUNT: A system for distributed equational deduction. In Hsiang, J., editor, *Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 397–402. Springer. (Cited on page 32.)

Bachmair, L., Ganzinger, H., Lynch, C., and Snyder, W. (1992). Basic paramodulation and superposition. In *Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 462–476. Springer. (Cited on page 11.)

Backes, J. and Brown, C. E. (2011). Analytic Tableaux for Higher-Order Logic with Choice. *Journal of Automated Reasoning*, 47(4):451–479. (Cited on page 35.)

Barendregt, H. P., Dekkers, W., and Statman, R. (1977). Typed lambda calculus. In *Handbook of Mathematical Logic*, pages 1091–1132. Elsevier. (Cited on page 28.)

Barker-Plummer, D., Etchemendy, J., Liu, A., Murray, M., and Swoboda, N. (2008). Openproof – A Flexible Framework for Heterogeneous Reasoning. In *Diagrammatic Representation and Inference*, volume 5223 of *Lecture Notes in Computer Science*, pages 347–349. Springer. (Cited on page 112.)

Barrett, C., Deters, M., de Moura, L., Oliveras, A., and Stump, A. (2013). 6 years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277. (Cited on page 14.)

Becker, M. Y., Russo, A., and Sultana, N. (2012). Foundations of Logic-Based Trust Management. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 161–175, Washington, DC, USA. IEEE Computer Society. (Cited on page 15.)

Ben-Ari, M. (2012). *Mathematical Logic for Computer Science*. Springer. (Cited on page 11.)

Benzmüller, C. (1999). *Equality and Extensionality in Higher-Order Theorem Proving*. PhD thesis, Naturwissenschaftlich-Technische Fakultät I, Saarland University. (Cited on pages 42, 65, 120, 139, and 141.)

Benzmüller, C. (2005). System description: LEO – a resolution based higher-order theorem prover. *Empirically Successful Automated Reasoning in Higher-Order Logic*, page 25. (Cited on page 42.)

Benzmüller, C. (2010). Combining Logics in Simple Type Theory. In *Computational Logic in Multi-Agent Systems*, volume 6814 of *Lecture Notes in Computer Science*, pages 33–48. Springer. (Cited on page 113.)

Benzmüller, C., Brown, C. E., and Kohlhase, M. (2004). Higher-Order Semantics and Extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088. (Cited on pages 26, 28, 29, 36, 149, and 150.)

Benzmüller, C., Brown, C. E., and Kohlhase, M. (2009). Cut-Simulation and Impredicativity. *Logical Methods in Computer Science*, 5(1:6):1–21. (Cited on page 94.)

Benzmüller, C. and Paulson, L. C. (2013). Quantified Multimodal Logics in Simple Type Theory. *Logica Universalis*, 7(1):7–20. (Cited on page 113.)

Benzmüller, C., Paulson, L. C., Theiss, F., and Fietzke, A. (2007). Progress Report on LEO-II – An Automatic Theorem Prover for Higher-Order Logic. In *Emerging Trends at Theorem Proving in Higher Order Logics*, pages 33–48. (Cited on page 45.)

Benzmüller, C., Rabe, F., and Sutcliffe, G. (2008a). THF0 – The Core TPTP Language for Classical Higher-Order Logic. In Baumgartner, P., Armando, A., and Gilles, D., editors, *International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 491–506. (Cited on page 111.)

Benzmüller, C., Sorge, V., Jamnik, M., and Kerber, M. (2008b). Combined Reasoning by Automated Cooperation. *Journal of Applied Logic*, 6(3):318–342. (Cited on page 121.)

Benzmüller, C. and Sultana, N. (2013). LEO-II version 1.5. In *International Workshop on Proof Exchange for Theorem Proving*, volume 14 of *EPiC Series*, pages 2–10. EasyChair. (Cited on pages 17, 32, 43, 44, 45, 49, 108, and 139.)

Benzmüller, C., Theiss, F., Paulson, L. C., and Fietzke, A. (2008c). LEO-II – A Cooperative Automatic Theorem Prover for Higher-Order Logic. In Armando, A., Baumgartner, P., and Dowek, G., editors, *International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 162–170. Springer. (Cited on pages 16 and 45.)

Benzmüller, C. and Woltzenlogel-Paleo, B. (2013). Gödel's God on the Computer. In Schulz, S., Sutcliffe, G., and Konev, B., editors, *International Workshop on the Implementation of Logics*, EPiC Series. (Cited on page 15.)

Berger, U., Buchholz, W., and Schwichtenberg, H. (2002). Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114(1):3–25. (Cited on page 20.)

Berghofer, S. (2003). *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München. (Cited on pages 20, 21, and 36.)

Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer. (Cited on page 35.)

Biere, A. (2008). PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97. (Cited on page 52.)

Blanchette, J. C. (2012). *Automatic Proofs and Refutations for Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München. (Cited on pages 43, 48, 117, and 119.)

Blanchette, J. C., Böhme, S., and Paulson, L. C. (2011). Extending Sledgehammer with SMT solvers. *Conference on Automated Deduction*, pages 116–130. (Cited on pages 117 and 119.)

Blanchette, J. C. and Popescu, A. (2013). Mechanizing the Metatheory of Sledgehammer. In *Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Computer Science*, pages 245–260. Springer. (Cited on page 48.)

Blanchette, J. C. and Urban, J., editors (2013). *International Workshop on Proof Exchange for Theorem Proving*, volume 14 of *EPiC Series*. EasyChair. (Cited on pages 130 and 136.)

Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. (2011). Why3: Shepherd your herd of provers. In *International Workshop on Intermediate Verification Languages*, pages 53–64. (Cited on page 115.)

Boespflug, M., Carbonneaux, Q., and Hermant, O. (2012). The $\lambda\Pi$-calculus Modulo as a Universal Proof Language. *International Workshop on Proof Exchange for Theorem Proving*, pages 28–43. (Cited on page 114.)

Böhme, S. (2012). *Proving Theorems of Higher-Order Logic with SMT Solvers*. PhD thesis, Institut für Informatik, Technische Universität München. (Cited on page 119.)

Böhme, S. and Nipkow, T. (2010). Sledgehammer: Judgement Day. *Journal of Automated Reasoning*, pages 107–121. (Cited on pages 14, 110, and 119.)

Böhme, S. and Weber, T. (2010). Fast LCF-style Proof Reconstruction for Z3. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer. (Cited on pages 25, 119, and 124.)

Böhme, S. and Weber, T. (2011). Designing Proof Formats: A User's Perspective. In Fontaine, P. and Stump, A., editors, *International Workshop on Proof Exchange for Theorem Proving*, pages 27–32. (Cited on pages 23, 40, and 41.)

Boolos, G. (1984). Don't Eliminate Cut. *Journal of Philosophical Logic*, 13(4):373–378. (Cited on page 99.)

Boulton, R. J. (1993). *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, Computer Laboratory, University of Cambridge. (Cited on page 107.)

Bove, A., Dybjer, P., and Sicard-Ramírez, A. (2012). Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs. In *Foundations of Software Science and Computational Structures*, volume 7213 of *Lecture Notes in Computer Science*, pages 104–118. Springer. (Cited on page 115.)

Bridge, J. P. and Paulson, L. C. (2013). Case Splitting in an Automatic Theorem Prover for Real-Valued Special Functions. *Journal of Automated Reasoning*, 50(1):99–117. (Cited on page 63.)

Brown, C. E. (2013). Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems. *Journal of Automated Reasoning*, 51(1):57–77. (Cited on page 50.)

Bundy, A. (1983). *The Computer Modelling of Mathematical Reasoning*. Academic Press. (Cited on page 60.)

Bundy, A. (1996). Proof planning. In Drabble, B., editor, *Artificial Intelligence Planning Systems*, pages 261–267. Association for the Advancement of Artificial Intelligence. (Cited on pages 87 and 120.)

Burel, G. (2013). A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo. In Blanchette and Urban (2013), pages 43–57. (Cited on page 114.)

Calude, C. S. and Müller, C. (2009). Formal Proof: Reconciling Correctness and Understanding. Technical report, University of Auckland. (Cited on page 119.)

Cerioli, M. and Meseguer, J. (1997). May I Borrow Your Logic? (Transporting Logical Structures along Maps). *Theoretical Computer Science*, 173(2):311–347. (Cited on page 93.)

Chihani, Z., Miller, D., and Renaud, F. (2013). Foundational Proof Certificates in First-Order Logic. In Bonacina, M. P., editor, *Conference on Automated Deduction*, volume 7898 of *Lecture Notes in Computer Science*, pages 162–177. Springer. (Cited on pages 15, 25, 26, 114, and 115.)

Church, A. (1940). A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68. (Cited on pages 36 and 113.)

Church, A. (1941). *The calculi of λ-conversion*. Princeton University Press. (Cited on pages 26 and 35.)

Claessen, K., Lilliestr\"om, A., and Smallbone, N. (2011). Sort It Out with Monotonicity. In *Conference on Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 207–221. Springer. (Cited on pages 48 and 49.)

Codescu, M., Horozal, F., Kohlhase, M., Mossakowski, T., Rabe, F., and Sojakova, K. (2010). Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In Mossakowski, T. and Kreowski, H.-J., editors, *International Workshop on Recent Trends in Algebraic Development Techniques*, volume 7137 of *Lecture Notes in Computer Science*, pages 139–159. Springer. (Cited on page 112.)

Constable, R. L., Allen, S. F., Bromley, M., Cleaveland, R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., and Smith, S. F. (1986). *Implementing mathematics with the Nuprl proof development system*. Prentice Hall. (Cited on page 120.)

Cousineau, D. and Dowek, G. (2007). Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *Typed Lambda Calculi and Applications*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer. (Cited on page 114.)

De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer. (Cited on pages 109 and 119.)

de Nivelle, H. (2001). Splitting through New Propositional Symbols. In Nieuwenhuis, R. and Voronkov, A., editors, *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 172–185. Springer. (Cited on page 63.)

de Nivelle, H. (2002). Extraction of Proofs from Clausal Normal Form Transformation. In *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 584–598. Springer. (Cited on page 94.)

Delahaye, D. (2000). A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 85–95. Springer. (Cited on page 54.)

Denney, E. (2000). A Prototype Proof Translator from HOL to Coq. In *Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 108–125. Springer. (Cited on page 120.)

Dennis, L. A., Collins, G., Norrish, M., Boulton, R., Slind, K., Robinson, G., Gordon, M., and Melham, T. F. (2003). The PROSPER toolkit. *International Journal on Software Tools for Technology Transfer*, 4:189–210. (Cited on page 39.)

Digricoli, V. J. and Harrison, M. C. (1986). Equality-based binary resolution. *Journal of the ACM*, 33(2):253–289. (Cited on page 11.)

Dowek, G. (2009). Skolemization in Simple Type Theory: the Logical and the Theoretical Points of View. In Benzmüller, C., Brown, C. E., Siekmann, J., and R.Statman, editors, *Festschrift in Honour of Peter B. Andrews on his 70th Birthday*, Studies in Logic and the Foundations of Mathematics. College Publications. (Cited on pages 41 and 81.)

Dunchev, C., Leitsch, A., Libal, T., Riener, M., Rukhaia, M., Weller, D., and Woltzenlogel-Paleo, B. (2012). System Feature Description: Importing Refutations into the GAPT Framework. In Pichardie, D. and Weber, T., editors, *International Workshop on Proof Exchange for Theorem Proving*, volume 878 of *CEUR Workshop Proceedings*, pages 51–57. CEUR-WS.org. (Cited on pages 21, 23, 117, and 118.)

Eén, N. and Sörensson, N. (2003). An Extensible SAT-solver. In Giunchiglia, E. and Tacchella, A., editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer. (Cited on page 50.)

Fietzke, A. and Weidenbach, C. (2009). Labelled splitting. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):3–34. (Cited on page 63.)

Fitting, M. (1996). *First-Order Logic and Automated Theorem Proving*. Springer. (Cited on pages 14 and 65.)

Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L. P., and Tiu, A. F. (2006). Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In Hermanns, H. and Palsberg, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer. (Cited on page 116.)

Foster, S. and Struth, G. (2011). Integrating an Automated Theorem Prover into Agda. In Bobaru, M. G., Havelund, K., Holzmann, G. J., and Joshi, R., editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 116–130. Springer. (Cited on page 118.)

Gentzen, G. (1969). *The Collected Papers of Gerhard Gentzen*. North Holland. Edited by M. E. Szabo. (Cited on page 20.)

Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38:173–98. (Cited on page 13.)

Gordon, M. J. C. (1985). HOL : A Machine Oriented formulation of Higher Order Logic. Technical Report UCAM-CL-TR-68, Computer Laboratory, University of Cambridge. (Cited on page 35.)

Gordon, M. J. C. and Melham, T. F. (1993). *Introduction to HOL: a Theorem-Proving Environment for Higher Order Logic*. Cambridge University Press. (Cited on pages 35, 39, and 116.)

Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979). *Edinburgh LCF: A Mechanised Logic of Computation*. Springer. (Cited on pages 19 and 35.)

Gordon, M. J. C., Reynolds, J., Jr., W. A. H., and Kaufmann, M. (2006). An Integration of HOL and ACL2. In *Formal Methods in Computer-Aided Design*, pages 153–160. IEEE Computer Society. (Cited on page 120.)

Hales, T. C., Harrison, J., McLaughlin, S., Nipkow, T., Obua, S., and Zumkeller, R. (2010). A Revision of the Proof of the Kepler Conjecture. *Discrete & Computational Geometry*, 44(1):1–34. (Cited on page 15.)

Hamadi, Y., Jabbour, S., and Sais, L. (2009). ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262. (Cited on page 14.)

Harper, R., Honsell, F., and Plotkin, G. D. (1993). A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184. (Cited on pages 13 and 111.)

Harrison, J. (1995). Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK. (Cited on pages 35, 66, and 107.)

Harrison, J. (1996). HOL Light: A Tutorial Introduction. In Srivas, M. K. and Camilleri, A. J., editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer. (Cited on page 120.)

Harrison, J. (2008). Formal proof – theory and practice. *Notices of the AMS*, 11:1395–1406. (Cited on page 113.)

Hickey, J., Nogin, A., Constable, R. L., Aydemir, B. E., Barzilay, E., Bryukhov, Y., Eaton, R., Granicz, A., Kopylov, A., Kreitz, C., Krupski, V., Lorigo, L., Schmitt, S., Witty, C., and Yu, X. (2003). MetaPRL - A Modular Logical Environment. In Basin, D. A. and Wolff, B., editors, *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer. (Cited on page 120.)

Hintikka, J. (1998). *The Principles of Mathematics Revisited*. Cambridge University Press. (Cited on page 113.)

Huet, G. P. (1975). A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57. (Cited on pages 42, 45, and 141.)

Hughes, R. J. M. (1982). Super-Combinators: A New Implementation Method for Applicative Languages. In *ACM symposium on LISP and functional programming*, pages 1–10. ACM. (Cited on page 49.)

Hurd, J. (1999). Integrating Gandalf and HOL. In Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., and Théry, L., editors, *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321. Springer. (Cited on pages 23, 39, and 117.)

Hurd, J. (2003). First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In Archer, M., Vito, B. D., and Muñoz, C., editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number CP-2003-212448 in NASA Technical Reports, pages 56–68. (Cited on pages 39, 48, 109, 117, and 119.)

Hurd, J. (2011). The OpenTheory standard theory library. In Bobaru, M., Havelund, K., Holzmann, G. J., and Joshi, R., editors, *International Symposium on NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer. (Cited on page 120.)

Hustadt, U. and Schmidt, R. A. (2000). MSPASS: Modal Reasoning by Translation and First-Order Resolution. In Dyckhoff, R., editor, *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1847 of *Lecture Notes in Artificial Intelligence*, pages 67–71. Springer. (Cited on page 93.)

Kaliszyk, C. and Krauss, A. (2013). Scalable LCF-style Proof Translation. In *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 51–66. (Cited on page 120.)

Kaliszyk, C. and Urban, J. (2013). PRocH: Proof Reconstruction for HOL Light. In *Conference on Automated Deduction*, volume 7898 of *Lecture Notes in Computer Science*, pages 267–276. Springer. (Cited on page 119.)

Keller, C. (2013). *A Matter of Trust: Skeptical Communication Between Coq and External Provers*. PhD thesis, École Polytechnique. (Cited on pages 15, 25, 26, 113, 115, and 120.)

Kohlhase, M. (2006). *OMDoc – An Open Markup Format for Mathematical Documents*, volume 4180 of *Lecture Notes in Artificial Intelligence*. Springer. (Cited on page 112.)

Krstic, S. and Goel, A. (2007). Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *Frontiers of Combining Systems*, volume 4720 of *Lecture Notes in Computer Science*, pages 1–27. Springer. (Cited on page 94.)

Kugler, H., Paoletti, N., Yordanov, B., Hamadi, Y., and Wintersteiger, C. M. (2014). Analyzing and Synthesizing Genomic Logic Functions. In Biere, A. and Bloem, R., editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 343–357. Springer. (Cited on page 15.)

Liang, C. and Miller, D. (2011). A Focused Approach to Combining Logics. *Annals of Pure and Applied Logic*, 162(9):679–697. (Cited on pages 25, 113, and 114.)

Matichuk, D., Wenzel, M., and Murray, T. (2014). An Isabelle Proof Method Language. In *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 390–405. Springer. (Cited on page 53.)

McCune, W. (1990). Otter 2.0. In *Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 663–664. Springer. (Cited on pages 32 and 118.)

McCune, W. (2005). Release of Prover9. In *Mile High Conference on Quasigroups, Loops and Nonassociative Systems*, Denver, Colorado. (Cited on page 118.)

McLaughlin, S., Barrett, C., and Ge, Y. (2006). Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. *Electronic Notes in Theoretical Computer Science*, 144(2):43–51. (Cited on page 120.)

McMillan, K. L. (2011). Interpolants from Z3 Proofs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 19–27, Austin, TX. FMCAD Inc. (Cited on page 15.)

Meng, J. and Paulson, L. C. (2008). Translating Higher-Order Clauses to First-Order Clauses. *Journal of Automated Reasoning*, 40(1):35–60. (Cited on pages 48, 49, and 117.)

Meng, J., Quigley, C., and Paulson, L. C. (2006). Automation for Interactive Proof: First Prototype. *Information and Computation*, 204(10):1575–1596. (Cited on pages 20, 23, and 117.)

Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *Definition of Standard ML (Revised)*. MIT Press. (Cited on pages 19 and 35.)

Mossakowski, T., Maeder, C., and Lüttich, K. (2007). The Heterogeneous Tool Set (Hets). In Beckert, B., editor, *Verification Workshop*, volume 259 of *CEUR Workshop Proceedings*. CEUR-WS.org. (Cited on page 112.)

Naumov, P., Stehr, M.-O., and Meseguer, J. (2001). The HOL/NuPRL Proof Translator. In *Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 329–345. Springer. (Cited on page 120.)

Nieuwenhuis, R. (2002). The impact of CASC in the development of automated deduction systems. *Journal of AI Communications*, 15:77–78. (Cited on page 14.)

Nieuwenhuis, R. and Rubio, A. (2001). Paramodulation-based theorem proving. In Robinson and Voronkov (2001), pages 371–443. (Cited on page 11.)

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer. (Cited on pages 16, 35, and 53.)

Nonnengart, A. and Weidenbach, C. (2001). Computing small clause normal forms. In Robinson and Voronkov (2001), pages 335–367. (Cited on pages 11, 32, 66, and 75.)

Obua, S. and Skalberg, S. (2006). Importing HOL into Isabelle/HOL. In Furbach, U. and Shankar, N., editors, *International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 298–302. Springer. (Cited on page 120.)

Paulson, L. C. (1989). The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397. (Cited on pages 35, 67, 80, and 144.)

Paulson, L. C. (1994). *Isabelle – A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer. (Cited on pages 13, 35, and 37.)

Paulson, L. C. (1997). Generic automatic proof tools. In Veroff, R., editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*. MIT Press. Chapter 3. (Cited on pages 116 and 117.)

Paulson, L. C. (1999). A Generic Tableau Prover and its Integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–87. (Cited on pages 21, 39, 105, and 117.)

Paulson, L. C. and Blanchette, J. C. (2010). Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *International Workshop on the Implementation of Logics*. (Cited on pages 20, 21, 23, 24, 109, 115, and 117.)

Paulson, L. C. and Susanto, K. W. (2007). Source-Level Proof Reconstruction for Interactive Theorem Proving. In *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 232–245. Springer. (Cited on pages 20, 21, 23, 25, 116, 117, 118, and 119.)

Pfenning, F. (1987). *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University. (Cited on page 20.)

Pfenning, F. and Schürmann, C. (1999). System Description: Twelf – A Meta-Logical Framework for Deductive Systems. In *Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer. (Cited on pages 13 and 111.)

Quaife, A. (1992). *Automated Development of Fundamental Mathematical Theories*. Kluwer Academic Publishers. (Cited on page 60.)

Rabe, F. (2008). *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen. (Cited on page 112.)

Rabe, F., Kohlhase, M., and Coen, C. S. (2011). A Foundational View on Integration Problems. In *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 107–122. Springer. (Cited on pages 112 and 120.)

Riazanov, A. and Voronkov, A. (2001). Splitting Without Backtracking. In Nebel, B., editor, *International Joint Conference on Artificial Intelligence*, pages 611–617. Morgan Kaufmann. (Cited on page 63.)

Ridge, T. and Margetson, J. (2005). A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic. In Hurd, J. and Melham, T., editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 295–309. Springer. (Cited on pages 66 and 67.)

Robinson, J. A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41. (Cited on pages 11 and 14.)

Robinson, J. A. (2000). Computational Logic: Memories of the Past and Challenges for the Future. In *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 1–24. Springer. (Cited on page 113.)

Robinson, J. A. and Voronkov, A., editors (2001). *Handbook of Automated Reasoning*. MIT Press. (Cited on pages 127, 135, and 138.)

Rudnicki, P. and Urban, J. (2011). Escape to ATP for Mizar. In Fontaine, P. and Stump, A., editors, *International Workshop on Proof Exchange for Theorem Proving*. (Cited on page 115.)

Schmitt, S., Lorigo, L., Kreitz, C., and Nogin, A. (2001). JProver: Integrating Connection-Based Theorem Proving into Interactive Proof Assistants. In *Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 421–426. Springer. (Cited on page 120.)

Schulz, S. (2002a). A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae. In Haller, S. and Simmons, G., editors, *International Conference of the Florida Artificial Intelligence Research Society*, pages 72–76. Association for the Advancement of Artificial Intelligence. (Cited on page 63.)

Schulz, S. (2002b). E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126. (Cited on pages 16 and 44.)

Schulz, S. (2013). E 1.8 user manual. Distributed with E. (Cited on page 75.)

Schürmann, C. and Stehr, M.-O. (2006). An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In Hermann, M. and Voronkov, A., editors, *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 4246 of *Lecture Notes in Computer Science*, pages 150–166. Springer. (Cited on page 111.)

Siekmann, J. H., Benzmüller, C., and Autexier, S. (2006). Computer supported mathematics with Omega. *Journal of Applied Logic*, 4(4):533–559. (Cited on pages 113 and 120.)

Smolka, S. J. and Blanchette, J. C. (2013). Robust, Semi-Intelligible Isabelle Proofs from ATP Proofs. In Blanchette and Urban (2013), pages 117–132. (Cited on pages 57, 116, and 120.)

Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier. (Cited on page 28.)

Sultana, N. and Benzmüller, C. (2012). Understanding LEO-II's proofs. In Ternovska, E., Korovin, K., and Schulz, S., editors, *International Workshop on the Implementation of Logics*. (Cited on pages 17, 21, 32, 63, 121, and 139.)

Sultana, N., Blanchette, J. C., and Paulson, L. C. (2012). LEO-II and Satallax on the Sledgehammer test bench. *Journal of Applied Logic*. (Cited on pages 17, 25, 43, 44, 45, 50, 89, 110, and 117.)

Sutcliffe, G. (2009). The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362. (Cited on page 29.)

Sutcliffe, G., Chang, C., Ding, L., McGuinness, D., and Da Silva, P. P. (2010). Different proofs are good proofs. In *Workshop on Evaluation Methods for Solvers, and Quality Metrics for Solutions*, pages 1–10. (Cited on page 115.)

Sutcliffe, G., Chang, C., McGuinness, D., Lebo, T., Ding, L., Da Silva, P. P., et al. (2011). Combining proofs to form different proofs. In Fontaine, P. and Stump, A., editors, *International Workshop on Proof Exchange for Theorem Proving*. (Cited on page 115.)

Sutcliffe, G., Denney, E., and Fischer, B. (2005). Practical Proof Checking for Program Certification. In *Workshop on Empirically Successful Classical Automated Reasoning*. (Cited on page 118.)

Sutcliffe, G. and Suttner, C. (2001). Evaluating general purpose automated theorem proving systems. *Artificial Intelligence*, 131(1-2):39–54. (Cited on page 14.)

Tarski, A. (1936). Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405. (Cited on page 13.)

Teucke, A. (2011). Translating a Satallax Refutation to a Tableau Refutation Encoded in Coq. Bachelor's thesis, Saarland University. (Cited on pages 17, 21, 24, 25, 35, 51, 53, 123, 124, and 126.)

Theiss, F. and Benzmüller, C. (2006). Term indexing for the LEO-II prover. In *International Workshop on the Implementation of Logics*. (Cited on pages 32 and 43.)

Trac, S., Puzis, Y., and Sutcliffe, G. (2006). An Interactive Derivation Viewer. In *Workshop on User Interfaces for Theorem Provers*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 109–123. Elsevier. (Cited on pages 22 and 46.)

Troelstra, A. S. and Schwichtenberg, H. (2000). *Basic Proof Theory*. Number 43 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press. (Cited on page 20.)

Tseitin, G. S. (1968). On the Complexity of Derivation in Propositional Calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2(115-125):10–13. (Cited on page 60.)

Underwood, J. (1990). A Constructive Completeness Proof for Intuitionistic Propositional Calculus. Technical report, Cornell University. (Cited on page 113.)

Urbas, M. and Jamnik, M. (2012). Diabelli: A Heterogeneous Proof System. In *Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 559–566. Springer. (Cited on page 112.)

von Gleissenthall, K. and Rybalchenko, A. (2013). An Epistemic Perspective on Consistency of Concurrent Computations. In *CONCUR 2013 – Concurrency Theory*, volume 8052 of *Lecture Notes in Computer Science*, pages 212–226. Springer. (Cited on page 15.)

Wang, H. (1960). Toward Mechanical Mathematics. *IBM Journal of Research and Development*, 4(1):2–22. (Cited on page 14.)

Weber, T. and Amjad, H. (2009). Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7(1):26 – 40. (Cited on page 117.)

Weich, K. (1998). Decision Procedures for Intuitionistic Propositional Logic by Program Extraction. In *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 292–306. Springer. (Cited on page 113.)

Weidenbach, C. (2001). Combining superposition, sorts and splitting. In Robinson and Voronkov (2001), pages 1965–2013. (Cited on page 63.)

Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., and Wischnewski, P. (2009). SPASS Version 3.5. In *Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer. (Cited on page 117.)

Wenzel, M. (1997). Type Classes and Overloading in Higher-Order Logic. In *Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer. (Cited on page 36.)

Wenzel, M. (2002). *Isabelle/Isar – a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München. (Cited on pages 16, 19, 37, 51, 76, 116, and 118.)

Whitehead, A. N. (1911). *An Introduction to Mathematics*. H. Holt. (Cited on page 14.)

Whitehead, A. N. and Russell, B. (1910). *Principia Mathematica (Volume 1)*. Cambridge University Press. (Cited on page 14.)

Zummer, J., Meier, A., Sutcliffe, G., and Zhang, Y. (2004). Integrated Proof Transformation Services. In *Workshop on Computer-Supported Mathematical Theory Development*. (Cited on pages 21, 23, and 118.)

Appendix A

Leo-II

A.1 Calculus

This section lists some of the proof rules implemented in Leo-II, using the notation used in this dissertation (cf. Chapter 2). These rules are described in more detail in the literature (Benzmüller, 1999; Sultana and Benzmüller, 2012; Benzmüller and Sultana, 2013). Some of the rules appearing here are renamed slightly, to improve clarity.

Clause normalisation.

$$\text{extcnf\_not\_pos}\ \frac{\left\{^{+}\neg\mathcal{T}, \ldots\right\}}{\left\{^{-}\mathcal{T}, \ldots\right\}}$$

$$\text{extcnf\_not\_neg}\ \frac{\left\{^{-}\neg\mathcal{T}, \ldots\right\}}{\left\{^{+}\mathcal{T}, \ldots\right\}}$$

$$\text{extcnf\_or\_neg}\ \frac{\left\{^{-}\mathcal{T}_1 \vee \mathcal{T}_2, \ldots\right\}}{\left\{^{-}\mathcal{T}_1, \ldots\right\} \qquad \left\{^{-}\mathcal{T}_2, \ldots\right\}}$$

$$\text{extcnf\_or\_pos}\ \frac{\left\{^{+}\mathcal{T}_1 \vee \mathcal{T}_2, \ldots\right\}}{\left\{^{+}\mathcal{T}_1, {}^{+}\mathcal{T}_2, \ldots\right\}}$$

$$\text{extcnf\_forall\_pos}\ \frac{\left\{^{+}\forall_\tau \mathcal{T}, \ldots\right\}}{x\left\{^{+}\mathcal{T}\ x, \ldots\right\}}\ \text{Fresh } x : \tau \text{ wrt the clause}$$

$$\text{extcnf\_forall\_neg}\ \frac{\left\{^{-}\forall_\tau \mathcal{T}, \ldots\right\}}{\left\{^{-}\mathcal{T}\ \mathsf{sk}, \ldots\right\}}\ \text{Fresh } \mathsf{sk} : \tau \text{ wrt signature}$$

Extensionality.

$$\text{ext\_func}\ \frac{\left\{^{\mathcal{P}}\ \mathcal{T}_1 = \mathcal{T}_2\ , \ldots\right\}}{\left\{^{\mathcal{P}}\ \forall_{\tau_1} X.\ \mathcal{T}_1\, X = \mathcal{T}_2\, X\ , \ldots\right\}}\ \text{where } \mathcal{T}_1, \mathcal{T}_2 : \tau_1 \to \tau_2;\ X \text{ fresh in } \mathcal{T}_1, \mathcal{T}_2$$

$$\text{ext\_boole\_pos}\ \frac{\left\{^{+}\ \mathcal{T}_1 = \mathcal{T}_2\ , \ldots\right\}}{\left\{^{-}\ \mathcal{T}_1\ , ^{+}\ \mathcal{T}_2\ , \ldots\right\} \qquad \left\{^{+}\ \mathcal{T}_1\ , ^{-}\ \mathcal{T}_2\ , \ldots\right\}}\ \text{where } \mathcal{T}_1, \mathcal{T}_2 : o$$

$$\text{ext\_boole\_neg}\ \frac{\left\{^{-}\ \mathcal{T}_1 = \mathcal{T}_2\ , \ldots\right\}}{\left\{^{+}\ \mathcal{T}_1\ , ^{+}\ \mathcal{T}_2\ , \ldots\right\} \qquad \left\{^{-}\ \mathcal{T}_1\ , ^{-}\ \mathcal{T}_2\ , \ldots\right\}}\ \text{where } \mathcal{T}_1, \mathcal{T}_2 : o$$

Unification.

$$\text{uni\_triv}\ \frac{\left\{^{-}\ \mathcal{T} = \mathcal{T}\ , \ldots\right\}}{\{\ldots\}}$$

$$\text{uni\_dec}\ \frac{\left\{^{-}\ \mathcal{T}\, \mathcal{T}_1 \ldots \mathcal{T}_n = \mathcal{T}\, \mathcal{T'}_1 \ldots \mathcal{T'}_n\ , \ldots\right\}}{\left\{^{-}\ \mathcal{T}_1 = \mathcal{T'}_1\ , \ldots\ , ^{-}\ \mathcal{T}_n = \mathcal{T'}_n\ , \ldots\right\}}\ \mathcal{T} \text{ is a constant}$$

$$\text{uni\_flex\_rigid}\ \frac{\left\{^{-}\ X\, \mathcal{T}_1 \ldots \mathcal{T}_n = \mathcal{T}\, \mathcal{T'}_1 \ldots \mathcal{T'}_m\ , \ldots\right\}}{\left\{^{-}\ X\, \mathcal{T}_1 \ldots \mathcal{T}_n = \mathcal{T}\, \mathcal{T'}_1 \ldots \mathcal{T'}_m\ , \ldots\right\}\theta}\ \mathcal{T} \text{ is a constant}$$

The uni_flex_rigid rule produces a function θ that maps the flexible variable X into a term. There are two kinds of mappings that θ can make:

- *Projection* binding: X is mapped to the term

  $$\lambda X_1, \ldots, X_n.\ X_i\ \left(Y_1\, X_1 \ldots X_n\right)\ \ldots\ \left(Y_k\, X_1 \ldots X_n\right)$$

  where k is the arity of X_i, and each Y_j is a fresh variable.

- *Imitation* binding: X is mapped to the term

  $$\lambda X_1, \ldots, X_n.\ \mathcal{T}\ \left(Y_1\, X_1 \ldots X_n\right)\ \ldots\ \left(Y_k\, X_1 \ldots X_n\right)$$

  where k is the arity of some suitably-typed constant $\mathcal{T}$, and each Y_j is a fresh variable.

$$\text{uni\_subst}\ \frac{\left\{^{-}\ X = \mathcal{T}\ , \ldots\right\}}{\{\ldots\}\ [\mathcal{T}/X]}\ X \notin \mathrm{FV}(\mathcal{T})$$

This set of rules is based on the algorithm by Huet (1975). The rule uni_subst does not appear explicitly in Leo-II proofs; it appears as "bind" annotations in other inferences.[1] Rules uni_subst, uni_triv and uni_dec simply carry out straightforward simplification steps, while the uni_flex_rigid rule carries out search for term instances.

Following Huet, Benzmüller does not rely on a flex-flex rule (i.e., handling unification literals having variables in head positions) to ensure completeness, and conjectures that such a rule would be redundant (Benzmüller, 1999).

The uni_flex_rigid rule computes partial bindings, and it is very prolific in introducing flexible head variables—which in turn may result in more applications of the rule. Application of this rule is controlled in Leo-II through the *unification depth* parameter, which limits the number of nested applications of uni_flex_rigid. A judicious choice of unification depth can improve completeness, but at the cost of performance.

Resolution.

$$\text{res}\ \frac{\left\{{}^{\mathcal{P}_1}\,\mathcal{T}_1, \mathcal{C}_1\right\} \qquad \left\{{}^{\mathcal{P}_2}\,\mathcal{T}_2, \mathcal{C}_2\right\}}{\left\{{}^{-}\,\mathcal{T}_1 = \mathcal{T}_2, \mathcal{C}_1, \mathcal{C}_2\right\}}\ \mathcal{P}_1 \neq \mathcal{P}_2$$

$$\text{fac}\ \frac{\left\{{}^{\mathcal{P}}\,\mathcal{T}_1, {}^{\mathcal{P}}\,\mathcal{T}_2\right\}}{\left\{{}^{\mathcal{P}}\,\mathcal{T}_1, {}^{-}\,\mathcal{T}_1 = \mathcal{T}_2\right\}}$$

Note that the factorisation rule handles only binary clauses. It is an open question whether this rule, in the context of the other rules of the calculus, causes the calculus to be incomplete because of this restriction.

$$\text{prim\_sub}\ \frac{\left\{{}^{\mathcal{P}}\,(X:\tau)\,\mathcal{T}_1\ \ldots\ \mathcal{T}_n, \ldots\right\}}{\left\{{}^{\mathcal{P}}\,\mathcal{T}\,\mathcal{T}_1\ \ldots\ \mathcal{T}_n, \ldots\right\}}\ \mathcal{T}:\tau$$

The prim_sub rule instantiates variables in circumstances where we cannot use unification. The instantiation is carried out through an heuristic enumeration of terms. In Leo-II's implementation, the rule prim_sub is controlled by a parameter indicating how complex the substituted formulas should be.

Other rules. Leo-II has various other rules, including rules to simplify formalisations, and add support for the Axiom of Choice. Many of the other rules used in Leo-II serve administrative purposes, and do not add inherently logical strength to the calculus. For instance, the rule polarity_switch simply flips a literal's polarity and adds/removes a prefixing negation, clause_copy creates a new copy of a clause, and unfold_defs unfolds definitions.[2]

There is an additional family of rules in Leo-II that *does* add logical strength. The names of these rules are prefixed by fo_atp_, and suffixed by the name of a first-order theorem prover. These inferences represent oracle calls to the the respective theorem provers. An example, fo_atp_e, representing calls to the E theorem prover, can be seen in Figure 5.2.

[1] A proof transformation for handling such annotations is described in Section 5.2, point 3.

[2] The behaviour of the clause_copy inference rule is described in Section 5.5.1.1 more precisely.

Appendix B

Manual schematic contranegation

To demonstrate the techniques that were described in Chapter 4, this section gives some examples of formal and mechanised proofs. The examples are schematic, in the sense that we use the transformation on the inference rules rather than on instances of inference rules.

We start by proving the formula $\exists A.\forall P.\exists B.\ (P\ A\ B\ \ \vee\ \ \neg P\ B\ A)$ in a $\vdash_{\mathfrak{C}}$-style system based on Leo-II's calculus, then translate the proof into a $\vdash_{\mathfrak{V}}$-style system embedded in Isabelle/HOL.

Each line of the proof, except for the first, represents an inference based on the previous lines. The inferences are not labelled with the rule used; these are simple to determine, owing to the simplicity of the goal. The singleton hypothesis in the first line consists of the negated conjecture. The underlined hypothesis on line 6 is the one that participates in that inference. The "..." indicate previously derived clauses, which are elided to avoid clutter.

1. $\{\neg(\exists A.\forall P.\exists B.(P\ A\ B \vee \neg P\ B\ A))\} \vdash_{\mathfrak{C}} \neg(\forall P.\exists B.(P\ X_{\mathrm{A}}\ B \vee \neg P\ B\ X_{\mathrm{A}}))$
2. $\{\ldots, \neg(\forall P.\exists B.(P\ X_{\mathrm{A}}\ B \vee \neg P\ B\ X_{\mathrm{A}}))\} \vdash_{\mathfrak{C}}$
   $\neg(\exists B.((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ B \vee \neg(\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ B\ X_{\mathrm{A}}))$
3. $\{\ldots, \neg(\exists B.((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ B \vee \neg(\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ B\ X_{\mathrm{A}}))\} \vdash_{\mathfrak{C}}$
   $\neg((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ X_{\mathrm{B}} \vee \neg(\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{B}}\ X_{\mathrm{A}})$
4. $\{\ldots, \neg((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ X_{\mathrm{B}} \vee \neg(\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{B}}\ X_{\mathrm{A}})\} \vdash_{\mathfrak{C}} \neg((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ X_{\mathrm{B}})$
5. $\{\ldots, \neg((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ X_{\mathrm{B}} \vee \neg(\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{B}}\ X_{\mathrm{A}})\} \vdash_{\mathfrak{C}} \neg(\neg(\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{B}}\ X_{\mathrm{A}})$
6. $\{\ldots, \neg((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ X_{\mathrm{B}}), \underline{\neg(\neg(\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{B}}\ X_{\mathrm{A}})}\} \vdash_{\mathfrak{C}} (\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{B}}\ X_{\mathrm{A}}$
7. $\{\ldots, \neg((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ X_{\mathrm{B}}), (\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{B}}\ X_{\mathrm{A}}\} \vdash_{\mathfrak{C}}$
   $((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ X_{\mathrm{B}}) \neq ((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{B}}\ X_{\mathrm{A}})$
8. $\{\ldots, ((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{A}}\ X_{\mathrm{B}}) \neq ((\mathsf{sk}_{\mathrm{P}}\ X_{\mathrm{A}})\ X_{\mathrm{B}}\ X_{\mathrm{A}})\} \vdash_{\mathfrak{C}} \square$

In Isabelle/HOL we start the proof by applying the proof-by-contradiction rule:

$$[[\neg P] \Longrightarrow \mathsf{False}] \Longrightarrow P$$

then proceed as follows. Here we take a different approach to that used in the implemented prototype, described in Section 4.4.2. There we translated the proof by chaining together

instances of the inference rules. Here we do not work with instances: we formalise the inference rules as schemes, and we apply these schemes. Due to the generality of the schemes, we end up with logical variables in the transformed rules. Logical variables are placeholders of the terms that would specialise the inferences further. We represent a logical variable using the notation $\hat{A}_{(\vec{\kappa})}$, where A is the variable's name, and $\vec{\kappa}$ is a list of symbols to which the variable is applied. These symbols consist of $\bigwedge$-bound variables, and serve to ensure sound reasoning. Paulson (1989) describes how they work; the details are not important here.

1. $[\neg(\exists A.\forall P.\exists B.(P\,A\,B \vee \neg P\,B\,A))] \Longrightarrow \mathsf{False}$
2. $\Big[\ldots;\neg(\forall P.\exists B.(P\,\hat{A}_{()}\,B \vee \neg P\,B\,\hat{A}_{()}))\Big] \Longrightarrow \mathsf{False}$
3. $\Big[\ldots;\neg(\exists B.(\mathsf{P}\,\hat{A}_{()}\,B \vee \neg \mathsf{P}\,B\,\hat{A}_{()}))\Big] \Longrightarrow \mathsf{False}$
4. $\Big[\ldots;\neg(\mathsf{P}\,\hat{A}_{()}\,\hat{B}_{(\mathsf{P})} \vee \neg \mathsf{P}\,\hat{B}_{(\mathsf{P})}\,\hat{A}_{()})\Big] \Longrightarrow \mathsf{False}$
5. $\Big[\ldots;\neg(\mathsf{P}\,\hat{A}_{()}\,\hat{B}_{(\mathsf{P})});\underline{\neg(\neg \mathsf{P}\,\hat{B}_{(\mathsf{P})}\,\hat{A}_{()})}\Big] \Longrightarrow \mathsf{False}$
6. $\Big[\ldots;\neg(\mathsf{P}\,\hat{A}_{()}\,\hat{B}_{(\mathsf{P})});\mathsf{P}\,\hat{B}_{(\mathsf{P})}\,\hat{A}_{()}\Big] \Longrightarrow \mathsf{False}$
7. $\Big[\ldots;(\mathsf{P}\,\hat{A}_{()}\,\hat{B}_{(\mathsf{P})}) \neq (\mathsf{P}\,\hat{B}_{(\mathsf{P})})\,\hat{A}_{()}\Big] \Longrightarrow \mathsf{False}$
8. $[\ldots;\mathsf{False}] \Longrightarrow \mathsf{False}$

In steps 3 to 8, there is an implicit $\bigwedge P$ prefixing the state; we elide this here, but write P to indicate that that symbol is not bound at the object level.

B.1 Isabelle/HOL examples

B.1.1 Header

```
theory Example
imports HOL
begin
```

B.1.2 Tableau embedding

```
subsection "Calculus"

lemma extcnf_or_pos:
"[|C | ((A | B) = True);
   [|C | ((A | B) = True);
     C | (A = True) | (B = True)|] ==> False|] ==> False"
by auto
```

```
lemma extcnf_or_neg1:
"[|C | ((A | B) = False);
   [|C | ((A | B) = False);
     C | (A = False)|] ==> False|] ==> False"
by auto

lemma extcnf_or_neg2:
"[|C | ((A | B) = False);
   [|C | ((A | B) = False);
     C | (B = False)|] ==> False|] ==> False"
by auto

lemma extcnf_not_pos:
"[|C | ((~ A) = True);
   [|C | ((~ A) = True);
     C | (A = False)|] ==> False|] ==> False"
by auto

lemma extcnf_not_neg:
"[|C | ((~ A) = False);
   [|C | ((~ A) = False);
     C | (A = True)|] ==> False|] ==> False"
by auto

lemma extcnf_forall_pos:
"[|C | ((All A) = True);
   [|C | ((All A) = True);
     C | ((A X) = True)|] ==> False|] ==> False"
"[|C | ((Ex A) = False);
   [|C | ((Ex A) = False);
     C | ((A X) = False)|] ==> False|] ==> False"
by auto

lemma extcnf_forall_neg:
"[|C | ((All A) = False);
   !! sk. [|C | ((All A) = False);
             C | ((A sk) = False)|] ==> False|] ==> False"
by auto

lemma ext_func_pos:
"[|C | ((M = N) = True);
   [|C | ((M = N) = True);
     C | (((M X) = (N X)) = True)|] ==> False|] ==> False"
by auto
```

```
lemma ext_boole_pos:
"[|C | ((M = N) = True);
   [|C | ((M = N) = True);
     C | (((M --> N) & (N --> M)) = True)|] ==> False|] ==> False"
by auto

lemma ext_func_neg:
"[|C | ((M = N) = False);
   !! sk. [|C | ((M = N) = False);
            C | (((M sk) = (N sk)) = False)|] ==> False|] ==> False"
by auto

lemma ext_boole_neg:
"[|C | ((M = N) = False);
   [|C | ((M = N) = False);
     C | (((M --> N) & (N --> M)) = False)|] ==> False|] ==> False"
by auto

lemma uni_triv:
"[|C | ((A = A) = False);
   [|C | ((A = A) = False);
     C|] ==> False|] ==> False"
by auto

lemma uni_dec1:
"[|C | ((h U = h V) = False);
   [|C | ((h U = h V) = False);
     C | ((U = V) = False) |] ==> False|] ==> False"
by auto

lemma uni_flexrigid:
"[|C | ((F U = h V) = False);
   [|C | ((F U = h V) = False);
     C | (F = G) | ((F U = h V) = False)|] ==> False|] ==> False"
by auto

lemma res:
"[|C | (A = True);
   D | (B = False);
   [|C | (A = True);
     D | (B = False);
     C | D | ((A = B) = False)|] ==> False|] ==> False"
by auto

lemma fac_restr:
"[|(A = p) | (B = p);
   [|(A = p) | (B = p);
```

```
     (A = p) | ((A = B) = False)|] ==> False|] ==> False"
by auto
```

B.1.3 Helper code and lemmas

```
subsection "Helpers"

lemma unfold_def:
  "(A --> B) = (~ A | B)"
  "(A & B) = (~ (~ A | ~ B))"
by auto

ML {*
  fun resolve_unify thm = HEADGOAL (rtac thm THEN' atac)
*}

lemma collapse_false:
  "(False | (False | P)) = (False | P)"
  "((False | False) | P) = (False | P)"
by auto
```

B.1.4 Isabelle/HOL-reconstructed examples

```
subsection "Examples"

lemma "False | (! P. P | ~ P) = False ==> False"
apply (tactic {*resolve_unify @{thm extcnf_forall_neg}*})
apply (tactic {*resolve_unify @{thm extcnf_or_neg1}*})
apply (tactic {*resolve_unify @{thm extcnf_or_neg2}*})
apply (tactic {*resolve_unify @{thm extcnf_not_neg}*})
apply (thin_tac "False | (! P. P | ~ P) = False")
apply (thin_tac "False | (! P. P | ~ P) = False")
apply (thin_tac "False | (sk | ~ sk) = False")
apply (thin_tac "False | (sk | ~ sk) = False")
apply (thin_tac "False | (sk | ~ sk) = False")
apply (thin_tac "False | (~ sk) = False")
apply (thin_tac "False | (~ sk) = False")
apply (rule res[of "False" _ "False"])
apply assumption
apply assumption
apply (simp only: collapse_false)
apply (rule uni_triv)
apply assumption
apply assumption
done
```

```
lemma "False | (! A . ? X . ! B. (A X & B X) --> (B X & A X)) = False
  ==> False"
apply (tactic {*resolve_unify @{thm extcnf_forall_neg}*})
apply (thin_tac "False |
  (ALL A. EX X. ALL B. A X & B X --> B X & A X) = False")
apply (thin_tac "False |
  (ALL A. EX X. ALL B. A X & B X --> B X & A X) = False")
apply (erule extcnf_forall_pos)
apply (tactic {*resolve_unify @{thm extcnf_forall_neg}*})
apply (thin_tac "False |
                 (EX X. ALL B. sk X & B X --> B X & sk X) = False")
apply (thin_tac "False | (ALL B. sk (?X5 sk) & B (?X5 sk) -->
  B (?X5 sk) & sk (?X5 sk)) = False")
apply (thin_tac "False | (ALL B. sk (?X5 sk) & B (?X5 sk) -->
  B (?X5 sk) & sk (?X5 sk)) = False")
apply (simp only: unfold_def)
apply (erule extcnf_or_neg1)
apply (erule extcnf_or_neg2)
apply (thin_tac "False | (~ ~ (~ sk (?X15 sk sk) |
  ~ ska (?X15 sk sk)) | ~ (~ ska (?X15 sk sk) |
  ~ sk (?X15 sk sk))) = False")
apply (erule extcnf_not_neg)
apply (rotate_tac 2)
apply (erule extcnf_not_neg)
apply (rotate_tac -1)
apply (erule extcnf_or_pos)

apply (erule extcnf_not_pos)
apply (rotate_tac -1)
apply (erule extcnf_or_neg1)
apply (rotate_tac -2)
apply (erule extcnf_or_neg2)

(*very unwieldy to control reconstruction manually from here*)
apply auto
done
```

The presence of logical variables made it more difficult to control the reconstruction—at least manually. Unlike this manual experiment, the implemented prototype (Section 4.4.2) relies on *instances* of inference rules, and the translation takes place smoothly.

Appendix C

Proofs

C.1 Contranegation transformation

This section provides proofs related to the validity of the contranegation transformation (Definition 4.4.2). We will use the Henkin semantic framework (Benzmüller et al., 2004), which is commonly used for the formalisation of higher-order logic we use (Andrews, 2001). For consistency, we will use the notation used by Benzmüller et al. (2004). Note that the symbol $\equiv$ simply means equality at the level of truth values.

C.1.1 Lemma C.1.1

In the proof of Lemma 4.4.2, provided in Section C.1.3, we will rely on the following lemma about signature extension. Intuitively, it claims that we can always extend the signature without affecting the set of formulas that a model satisfies.

Lemma C.1.1. *For any Σ-model $\mathcal{M} = (\mathcal{D}, @, \mathcal{E}, v)$ and Σ'-model $\mathcal{M}' = (\mathcal{D}, @, \mathcal{E}', v)$ where $\Sigma \subseteq \Sigma'$, and where $\mathcal{E}_\varphi(\mathsf{c}) = \mathcal{E}'_\varphi(\mathsf{c})$ for all $\mathsf{c} \in \Sigma$, the following property holds: For any formula F with constants only in Σ, $\mathcal{M} \models F$ iff $\mathcal{M}' \models F$.*

Proof. We are to show that $v(\mathcal{E}_\varphi(F)) \equiv v(\mathcal{E}'_\varphi(F))$ for all φ, if $F \in \mathrm{wff}(\Sigma)$. We prove this by showing that $\mathcal{E}_\varphi(F) = \mathcal{E}'_\varphi(F)$ for all φ, if $F \in \mathrm{wff}(\Sigma)$.

Proceed by induction on F:

- Case F is a variable x: Using the definition of evaluation functions (Benzmüller et al., 2004, Remark 3.18), we have that $\mathcal{E}_\varphi(x) = \varphi(x) = \mathcal{E}'_\varphi(x)$.
- Case F is a constant c: Since $F \in \mathrm{wff}(\Sigma)$, then it is necessary that $\mathsf{c} \in \Sigma$. By assumption we have $\mathcal{E}_\varphi(\mathsf{c}) = \mathcal{E}'_\varphi(\mathsf{c})$ for all $\mathsf{c} \in \Sigma$.
- Case F is an application $t_1 t_2$: $\mathcal{E}_\varphi(t_1 t_2) = \mathcal{E}_\varphi(t_1) @ \mathcal{E}_\varphi(t_2) = \mathcal{E}'_\varphi(t_1) @ \mathcal{E}'_\varphi(t_2) = \mathcal{E}'_\varphi(t_1 t_2)$ using IH.
- Case F is an abstraction $\lambda x.\ t$: IH gives us that $\mathcal{E}_{\varphi,[\mathsf{a}/x]}(t) = \mathcal{E}'_{\varphi,[\mathsf{a}/x]}(t)$ for each $\mathsf{a} \in \mathcal{D}$. Using the IH, for arbitrary a we have $\mathcal{E}_\varphi(\lambda x.\ t) @ \mathsf{a} = \mathcal{E}_{\varphi,[\mathsf{a}/x]}(t) = \mathcal{E}'_{\varphi,[\mathsf{a}/x]}(t) = \mathcal{E}'_\varphi(\lambda x.\ t) @ \mathsf{a}$. Therefore $\mathcal{E}_\varphi(\lambda x.\ t) = \mathcal{E}'_\varphi(\lambda x.\ t)$ by extensionality.

□

C.1.2 Lemma 4.4.1

Proof. Recall that $\Phi \vdash_{\mathfrak{C}} \square$ iff there is no Henkin model $\mathcal{M}$ such that $\mathcal{M} \models \Phi$. Recall that $\mathcal{M}$ would consist of a tuple $(\mathcal{D}, @, \mathcal{E}, v)$, and that $\mathcal{M} \models \Phi$ iff for all $\phi \in \Phi$, $v(\mathcal{E}(\phi)) \equiv \top$. (Since each $\phi \in \Phi$ is closed, we need not worry about variable assignments.)

Equivalently, for all Henkin models, there is a $\phi \in \Phi$ such that $v(\mathcal{E}(\phi)) \equiv \bot$. This is equivalent to saying that, for all Henkin models, $v(\mathcal{E}(\phi_1 \wedge \ldots \wedge \phi_n)) \equiv \bot$, where $\{\phi_1, \ldots, \phi_n\} = \Phi$, and where we use the usual interpretation of the conjunction symbol '$\wedge$' (Benzmüller et al., 2004, Figure 2). We will continue to use the symbol Φ to abbreviate its conjunctive form $\phi_1 \wedge \ldots \wedge \phi_n$.

$v(\mathcal{E}(\Phi)) \equiv \bot$ holds iff, for all Henkin models, $v(\mathcal{E}(\Phi \longrightarrow \mathsf{False}) \equiv \top$, using the usual interpretation of '$\longrightarrow$' and 'False'. That is, $\models \Phi \longrightarrow \mathsf{False}$

From the assumption that $\vdash_{\mathfrak{V}}$ is complete, we have $\vdash_{\mathfrak{V}} \Phi \longrightarrow \mathsf{False}$. $\square$

C.1.3 Lemma 4.4.2

Proof. From the soundness proof of $\vdash_{\mathfrak{C}}$ we have that each rule $r \in \vdash_{\mathfrak{C}}$ preserves satisfiability. Rule r can be represented as a tuple of (possibly several) premises and (possibly several) conclusions, $((H_1, \ldots, H_n), (C_1, \ldots, C_m))$. Each H_i and C_i is a free-form formula of HOL. By the soundness of r, if there exists a model $\mathcal{M}$ such that $\mathcal{M} \models H$ then there exists a model $\mathcal{M}'$ such that $\mathcal{M}' \models H$ and $\mathcal{M}' \models C_i$ for some $1 \leq i \leq m$, and where $H = (H_1, \ldots, H_n)$.

Let $C = (C_1, \ldots, C_m)$. Recall that for consistency-preserving rules, we use the $\bigvee$ notation to bind free variables occurring in H and C. This is intended to formalise the idea that the interpretation of free top-level variables (occurring in H and C) by $\mathcal{M}'$ is unconstrained; therefore $\mathcal{M}'$ must satisfy H and C for at least one possible instantiation of their $\bigvee$-bound variables. For clarity, let us make free top-level variables explicit through Notation 4.4.1.[1] Then the statement of r's soundness becomes: if there exists a model $\mathcal{M}$ such that $\mathcal{M} \models \bigvee \vec{x}.H[\vec{x}]$, then there exists a model $\mathcal{M}'$ such that $\mathcal{M}' \models \bigvee \vec{x}.H[\vec{x}]$ and $\mathcal{M}' \models \bigvee \vec{y_i}.C_i[\vec{y_i}]$. Lifting the $\bigvee$ notation into the meta-language (i.e., English) the soundness statement becomes: if there exists a model $\mathcal{M}$ such that $\mathcal{M} \models H[\vec{x}]$ for some $\vec{x}$, then there exists a model $\mathcal{M}'$ such that $\mathcal{M}' \models H[\vec{x}]$ and $\mathcal{M}' \models C_i[\vec{y_i}]$ for some $i, \vec{x}, \vec{y_i}$.

Taking the contrapositive, this is equivalent to the statement: if there does not exist a model $\mathcal{M}'$ such that $\mathcal{M}' \models H[\vec{x}]$ and $\mathcal{M}' \models C_i[\vec{y_i}]$ for some $i, \vec{x}, \vec{y_i}$, then there does not exist a model $\mathcal{M}$ such that $\mathcal{M} \models H[\vec{x}]$ for some $\vec{x}$.

Recall that, for all ϕ, we have that $\models \neg\phi$ iff there does not exist a model $\mathcal{M}$ such that $\mathcal{M} \models \phi$. Also, for all ϕ_1, ϕ_2, we have that $\mathcal{M} \models \phi_1 \wedge \phi_2$ iff $\mathcal{M} \models \phi_1$ and $\mathcal{M} \models \phi_2$. By using these equivalences, and pushing in the negation, we can rewrite the previous statement as: if $\models \neg(H[\vec{x}] \wedge C_i[\vec{y_i}])$ then $\models \neg H[\vec{x}]$. Now this should hold for *arbitrary* $i, \vec{x}, \vec{y_i}$, because we pushed in the negation. Note that the antecedent and consequent occurrences of $\vec{x}$ occur in different scopes; we make this explicit by using the $\bigwedge$ binder when we switch to using Isabelle notation. Using $\bigwedge$ also makes explicit that the formulas are expected to hold for *all* $\vec{x}$ and $\vec{x}, \vec{y_i}$ respectively.

Since $\vdash_{\mathfrak{V}}$ is complete, we should be able to admit this in rule form:[2]

[1] Alternatively, we could have existentially-closed the object formula, but this would involve additional syntactic manipulation; it is best to keep things simple as long as the semantics are understood and preserved.

[2] Even better, this rule is derivable.

$$\frac{\vdash_{\mathfrak{v}} \neg(H[\vec{x}] \wedge C_1[\vec{y_1}]) \qquad \ldots \qquad \vdash_{\mathfrak{v}} \neg(H[\vec{x}] \wedge C_m[\vec{y_m}])}{\vdash_{\mathfrak{v}} \neg H[\vec{x}]}$$

Using the equivalence $(\neg A) = (A \longrightarrow \mathsf{False})$, followed by the implication-introduction rule and simplification we get the equivalent rule

$$\begin{array}{l}(\bigwedge \vec{x}, \vec{y_1}.\ \neg(H[\vec{x}] \wedge C_1[\vec{y_1}])) \Longrightarrow \\ \quad \ldots \Longrightarrow \\ \quad (\bigwedge \vec{x}, \vec{y_m}.\ \neg(H[\vec{x}] \wedge C_m[\vec{y_m}])) \Longrightarrow \\ \quad \bigwedge \vec{x}.\ H[\vec{x}] \Longrightarrow \mathsf{False}.\end{array}$$

By re-using the equivalence mentioned previously, together with the equivalence $(\longrightarrow) \equiv (\Longrightarrow)$, we obtain

$$\begin{array}{l}\left(\bigwedge \vec{x}, \vec{y_1}.\ \left[\begin{array}{c} H[\vec{x}] \wedge C_1[\vec{y_1}] \\ \vdots \\ \mathsf{False} \end{array}\right]\right) \Longrightarrow \\ \quad \ldots \Longrightarrow \\ \quad \left(\bigwedge \vec{x}, \vec{y_m}.\ \left[\begin{array}{c} H[\vec{x}] \wedge C_m[\vec{y_m}] \\ \vdots \\ \mathsf{False} \end{array}\right]\right) \Longrightarrow \\ \quad \bigwedge \vec{x}.\ H[\vec{x}] \Longrightarrow \mathsf{False}\end{array}$$

which, when rewritten using more conventional notation, and by eliminating the top-level iterated conjunction in H (to give $H_1, \ldots, H_n$), is equivalent to

$$\bigwedge \vec{x}\ \frac{\bigwedge \vec{x}, \vec{y_1}.\ \phi(1) \quad \ldots \quad \bigwedge \vec{x}, \vec{y_m}.\ \phi(m) \qquad H_1[\vec{x}] \quad \ldots \quad H_n[\vec{x}]}{\mathsf{False}}$$

where

$$\phi(i) = \left[\begin{array}{c} H_1[\vec{x}], \ldots, H_n[\vec{x}], C_i[\vec{y_i}] \\ \vdots \\ \mathsf{False} \end{array}\right]$$

The expanded formula is equivalent to $r^{\perp}$. Starting with the assumption that r is consistency-preserving we have arrived, via a series of iff-steps, at a validity-preserving rule equivalent to $r^{\perp}$. □

C.1.4 Lemma 4.4.3

Proof. We argue by induction on derivations π.

Case $\pi = ()$**.** Note that $\pi^{\perp} = \pi = ()$. A proof π is empty if it is immediate that $\Phi \vdash_{\mathfrak{C},\pi} \square$. If a proof is empty, then it must be necessary that $\mathsf{False} \in \Phi$. The argument is by contradiction: assume that $\Phi \vdash_{\mathfrak{C}} \square$ can be shown in 0 steps, and assume, for contradiction, that $\mathsf{False} \notin \Phi$. Then it could not be true that $\Phi \vdash_{\mathfrak{C}} \square$ can be shown in 0 steps. The argument for $\vdash_{\mathfrak{V}} \Phi \longrightarrow \mathsf{False}$ is similar.

From the fact that $\mathsf{False} \in \Phi$, it is straightforward to prove the base case.

Case $\pi = (\pi', r)$**.** That is, π consists of the subproof π' followed by an application of rule r. Without loss of generality let r be the rule

$$\frac{A_1 \quad \ldots \quad A_m}{B}$$

Working within an extended signature $\vec{\kappa'}$, the induction hypothesis allows us to assume that this property holds for π' and $\pi'^{\perp}$:

$$\Phi \cup \{A_1, \ldots, A_m, B\} \vdash_{\mathfrak{C},\pi'} \square \quad \text{iff} \quad \vdash_{\mathfrak{V},\pi'^{\perp}} \forall\vec{\kappa'}.\ \Phi \wedge A_1 \wedge \ldots \wedge A_m \wedge B \longrightarrow \mathsf{False}$$

We are to show that:

$$\Phi \cup \{A_1, \ldots, A_m\} \vdash_{\mathfrak{C},\pi} \square \quad \text{iff} \quad \vdash_{\mathfrak{V},\pi^{\perp}} \forall\vec{\kappa}.\ \Phi \wedge A_1 \wedge \ldots \wedge A_m \longrightarrow \mathsf{False}$$

where $\vec{\kappa} \subseteq \vec{\kappa'}$.

We start with the "only if" direction. Assume $\Phi \cup \{A_1, \ldots, A_m\} \vdash_{\mathfrak{C},\pi} \square$ to show

$$\vdash_{\mathfrak{V},\pi^{\perp}} \forall\vec{\kappa}.\ \Phi \wedge A_1 \wedge \ldots \wedge A_m \longrightarrow \mathsf{False}.$$

Note that π consists of π' followed by r, and that if $\Phi \cup \{A_1, \ldots, A_m\} \vdash_{\mathfrak{C},\pi} \square$ then $\Phi \cup \{A_1, \ldots, A_m, B\} \vdash_{\mathfrak{C},\pi'} \square$, by the soundness of r.

We can use this to infer, via the IH, that $\vdash_{\mathfrak{V},\pi'^{\perp}} \forall\vec{\kappa'}.\ \Phi \wedge A_1 \wedge \ldots \wedge A_m \wedge B \longrightarrow \mathsf{False}$. We use this fact as a hypothesis to $r^{\perp}$ (which we obtain via Lemma 4.4.2, together with its admissibility in $\vdash_{\mathfrak{V}}$), and introduce connectives to derive $\vdash_{\mathfrak{V},\pi^{\perp}} \forall\vec{\kappa}.\ \Phi \wedge A_1 \wedge \ldots \wedge A_m \longrightarrow \mathsf{False}$.

For the "if" direction, assume $\vdash_{\mathfrak{V},\pi^{\perp}} \forall\vec{\kappa}.\ \Phi \wedge A_1 \wedge \ldots \wedge A_m \longrightarrow \mathsf{False}$ to show $\Phi \cup \{A_1, \ldots, A_m\} \vdash_{\mathfrak{C},\pi} \square$. As before, the proof π relies on a subproof π' which proves $\vdash_{\mathfrak{V},\pi'^{\perp}} \forall\vec{\kappa'}.\ \Phi \wedge A_1 \wedge \ldots \wedge A_m \wedge B \longrightarrow \mathsf{False}$. Using the IH, we derive $\Phi \cup \{A_1, \ldots, A_m, B\} \vdash_{\mathfrak{C},\pi'} \square$. By the soundness of r, if $S = \Phi \cup \{A_1, \ldots, A_m, B\}$ is inconsistent then so is $S \backslash B$. Therefore we conclude that $\Phi \cup \{A_1, \ldots, A_m\} \vdash_{\mathfrak{C},\pi} \square$.

□

C.1.5 Corollary 4.4.1

This corollary follows from Lemma 4.4.3, but we can improve our confidence in the argument by checking the contranegation transformation on each Leo-II rule, to ensure that the result is derivable in Isabelle/HOL.

We can thus prove the corollary's statement in another way: we can show how chaining together the contranegated Leo-II rules in Isabelle/HOL yields a valid Isabelle/HOL proof. In a sense, this instantiates the argument made in Lemma 4.4.2 (in one direction) then unfolds it in Lemma 4.4.3.

We will make reference to the constituent rules of Leo-II's calculus, described in Section A.1. We will focus on rules that relate to normalisation, extensionality, unification, and resolution; we will ignore rules that carry out a logistical function (such as copying clauses, renaming variables, etc).

Definition C.1.1. *The* core rules *of Leo-II's calculus are those that belong to one of the following families:* $\{\text{normalisation}, \text{extensionality}, \text{unification}, \text{resolution}\}$

We will first seek to reduce a set of formulas Φ and a corresponding derivation π into an equisatisfiable set of formulas Φ' and corresponding derivation π'. This is done to justify why we can focus solely on the *core rules* of Leo-II. That is, π' consists only of instances of core rules. Relying on core rules might seem to impose restrictions on the input formulas Φ, but the next lemma will dispel this by showing that the input formulas can be translated into an equisatisfiable set of formulas Φ'.

Lemma C.1.2. *For each derivation π in Leo-II for Φ, there exists a derivation π' in Leo-II for Φ' such that π' consists of inferences made up solely of core rules, and that Φ is satisfiable iff Φ' is satisfiable.*

Proof. We need to show that each input problem can be replaced with an equisatisfiable input problem, and that arbitrary derivations from the former can be matched by core-rule-only derivations from the latter.

In this setting, a derivation is not necessarily a refutation—it does not need to end in an empty clause. Therefore the final rule r of a derivation could be any rule. We proceed by cases on the terminal rules of a derivation, and assume that the property holds for the preceding steps of the derivation.

If r is a core rule then the proof is immediate—it can be used in π', and does not require us to modify Φ in any way. We sketch the argument for the remaining rules, proceeding by cases on r.

- r is negate_conjecture: It cannot be the case that this occurs in arbitrary points in a derivation, since negate_conjecture occurs only at the very start of a refutation.
- r is polarity_switch: Replace the original literal with its double-negation. (Do this all the way up the derivation.) This does not change the formula's semantics, or the problem's satisfiability. Then double each application of negation-related normalisation rules to that literal. At this point, applying a single negation-related normalisation rule allows us to get the same effect as using the polarity_switch. Thus we can eliminate applications of polarity_switch.
- r is unfold_def: Form Φ' by unfolding the definitions in Φ. This makes Φ' and Φ equisatisfiable. Form π' by taking π and deleting the unfold_def-inferences in π (connecting the parent of the unfold_def inference node directly with the node's child.)

- r is variable_rename: Simply carry out the variable renaming at the source node.
- r is clause_copy: Delete the inference, and have its child point to the copied clause.
- r is fo_atp_e: We do not handle E subproofs; so in this case we could treat this as an oracle step, exclude derivations in which E played a part, or force Leo-II to act alone during proof-finding.
- r is sim: Add the simplification formulas to Φ', and make explicit any inference related to them in π'.
- r is extcnf_combined or standard_cnf: This should not happen, since we work on expanded proofs. (That is, all normalisation steps are explicit. They are core rules.)
- r is extcnf_forall_special_pos: Replace with extcnf_forall_pos, and include the resulting derivation from extcnf_forall_pos.
- r is split_conjecture: Instead of splitting a clause, combine the instantiation information from the split branches, to find the refutation for the full clause.
- r is solved_all_splits: This cannot happen, since occurrences of this inference will be removed when handling split_conjecture.

□

Lemma C.1.2 effectively describes a proof transformation that purifies Leo-II proofs to use the core calculus. This might also involve modifying the source formula set Φ to produce an equisatisfiable set Φ'.

Having shown that we can focus exclusively on the core rules of Leo-II, we now argue that the core rules can be transformed (Definition 4.4.2) to yield validity-preserving rules in Isabelle/HOL.

Lemma C.1.3. *For each rule r in Leo-II, if r is a core rule then we have that $r^{\perp}$ is derivable in Isabelle/HOL.*

Proof. Proceed by cases on the core rules. These were formalised and checked in Isabelle/HOL—the script is included in Section B.1.2. Note that:

- The decomposition rule is an n-ary rule. In the implementation described in Section 5.7, this rule was implemented as a function that accepts a parameter $1 < n < \omega$ and produces (and validates) an n-ary decomposition rule.
- The substitution and primitive substitution rules could not be formalised since they include a meta-logical operation (substitution) that is not explicit in Isabelle. In our description we factor-out instantiation steps from the proof, and apply the instantiations using a custom tactic. This is described in point 3 within Section 5.2.

□

Lemma C.1.4. *Every theorem proved by Leo-II is also a theorem of Isabelle/HOL.*

Proof. Lemma C.1.2 enables us to restrict our attention to the core rules of Leo-II. From Lemma C.1.3 we have that every core rule of Leo-II corresponds to an admissible rule in Isabelle/HOL via Definition 4.4.2. Then we need to map how proofs (as sequences of applications of core rules) formalised in Leo-II's calculus map into a calculus that can be embedded in Isabelle/HOL. This is described in the 'only if' direction of Lemma 4.4.3. □

C.2 Cut machines

C.2.1 Lemma 5.4.1

Proof. The proof proceeds by induction on the length of cut programs.

length is '0' Before the goal is loaded, there are no subgoals and no goal. The former are an empty conjunction—which has the same denotation as True— and the latter is True (as specified in Section 5.4). This is obviously valid.

length is '> 0' Proceed by cases on the type of instruction:

- case PROVE G: This loads the single subgoal G into the stack, and sets G as the program's goal. Thus we need to show that:

$$\frac{\vdash_\rho G}{\vdash_\rho G}$$

  which is obviously true.

- case CUT r: Let the current state be $(\rho,\ B : \sigma,\ F)$. The induction hypothesis is: if B and σ are valid, then F is valid. In symbols:

$$\frac{\vdash_\rho B \qquad \vdash_\rho \sigma}{\vdash_\rho F}$$

  The instruction 'CUT r' must succeed since, by assumption, the program is well-going. Let $r = (A_1, \ldots, A_n, B)$, or in more conventional notation

$$\frac{A_1 \qquad \ldots \qquad A_n}{B}$$

  Note that r is a rule in the target logic. The instruction moves the machine to state $(\rho,\ A_1 : \ldots : A_n : \sigma,\ F)$, and we need to prove that if $A_1, \ldots, A_n, \sigma$ are valid then F is valid. This is proved by applying r to the IH:

$$\frac{\dfrac{\vdash_\rho A_1 \qquad \ldots \qquad \vdash_\rho A_n}{\vdash_\rho B} \qquad \vdash_\rho \sigma}{\vdash_\rho F}$$

- case END: this instruction does not modify the state, and the instruction must succeed (since the program is well-going by assumption). This case simply uses the induction hypothesis directly.

□

C.2.2 Lemma 5.4.3

Proof. The proof proceeds by induction on the length of cut programs. The only interesting case is the induction step in the case of STORE, since none of the other instructions changes ρ.

Let us therefore consider the case of STORE. We have two induction hypotheses:

1. $\rho^0 \subseteq \rho^n$

2. For all $F \in (\rho^n \backslash \rho^0)$, $\vdash_{\rho^0} F$

relating to the machine state at step n: $(\rho^n,\ \sigma,\ G)$. Recall, from the semantics of STORE, that the state at $n+1$ is: $(\rho^{n+1},\ -,\ \mathsf{True})$ and that $\rho^{n+1} = \rho^n \cup_1 (\sigma, G)$. Using this fact, the IH.1 and the transitivity of $\subseteq$, it is straightforward to show that $\rho^0 \subseteq \rho^{n+1}$.

Next we argue that $\vdash_{\rho^0} F$ for all $F \in (\rho^{n+1} \backslash \rho^0)$. Note that

$$\left(\rho^{n+1} \backslash \rho^n\right) = \left\{\frac{\sigma}{G}\right\}.$$

From IH.2, we have that $\vdash_{\rho^0} F$ for all $F \in (\rho^n \backslash \rho^0)$. Therefore our goal is to show that $\vdash_{\rho^0} \frac{\sigma}{G}$. Let p' be the program since the last STORE instruction; that is, the program p consists of some prefix followed by p'. In case there is no previous STORE instruction (if this is the first STORE since the start of the program) then $p' = p$. Let m be the index of ρ at the start of p'; that is, $m \leq n$.[3] Then p' shows how using $\vdash_{\rho^m}$ we can prove $\frac{\sigma}{G}$. Note that since there were no STORE instructions between steps m and n, we have that $\rho^n = \rho^m$. Thus we have that p' shows that

$$\vdash_{\rho^n} \frac{\sigma}{G}$$

From IH.2, we have that all the rules in ρ^n are either present in ρ^0 or derived from its contents. Since we used rules in ρ^n to prove $\frac{\sigma}{G}$, then the latter is ultimately provable from the contents of ρ^0. □

[3] $m = n$ if p' contains a PROVE instruction and nothing else.