Third International Workshop on Proof Exchange for Theorem Proving (PxTP 2013)

Associated with

24th International Conference on
Automated Deduction
(CADE-24)

Lake Placid, New York, USA
June 9 and 10, 2013
Proceedings

Jasmin Christian Blanchette
Josef Urban
Editors

EasyChair Proceedings in Computing (EPiC)
Volume 14

Preface

The Third International Workshop on Proof Exchange for Theorem Proving (PxTP 2013) was held as a satellite event of the 24th International Conference on Automated Deduction (CADE-24) on June 9 and 10, 2013 in Lake Placid, New York.

The workshop featured an invited talk by Thomas C. Hales, presentations of the eleven accepted papers that are collected in this volume, and additional presentations by Pascal Fontaine (on behalf of Bruno Woltzenlogel Paleo), Cezary Kaliszyk (joint work with Alexander Krauss), Geoff Sutcliffe, and Josef Urban.

The focus of the workshop were various aspects of communication, integration, and cooperation between reasoning systems and formalisms. It is becoming clear that the success of deduction tools does not only depend on their power to solve large and difficult problems in an isolated manner, but also depends on their ability to cooperate.

The workshop's mission was thus to facilitate communication between reasoning tools, building of complex reasoning applications, and reuse of reasoning tools by developing and discussing suitable integration, translation and communication methods, standards, protocols, and programming interfaces. The final number of accepted submissions clearly shows that cooperation of reasoning systems is a thriving topic drawing the attention of researchers in automatic and interactive theorem proving.

We would like to thank the CADE organizers, the PxTP program committee, the authors and speakers, Andrei Voronkov of EasyChair, and everyone else who contributed to the workshop's success.

May 24, 2013
Munich, Germany
Nijmegen, Netherlands

Jasmin Christian Blanchette
Josef Urban

Table of Contents

Program Committee

Jasmin Christian Blanchette	Technische Universität München
Chad E. Brown	Saarland University
David Delahaye	CEDRIC/CNAM Paris
Ewen Denney	SGT/NASA Ames
Peter Dybjer	Chalmers University
Pascal Fontaine	Loria, INRIA, University of Nancy
John Harrison	Intel Corporation
Warren Hunt	University of Texas
Chantal Keller	Laboratoire d'Informatique de Polytechnique
Konstantin Korovin	Manchester University
Magnus O. Myreen	University of Cambridge
Jens Otten	University of Potsdam
Andrei Paskevich	Université Paris-Sud 11, IUT d'Orsay
Lawrence C. Paulson	University of Cambridge
David Pichardie	INRIA Rennes - Bretagne Atlantique
Florian Rabe	Jacobs University Bremen
Stephan Schulz	Technische Universität München
Aaron Stump	University of Iowa
Geoff Sutcliffe	University of Miami
Laurent Théry	INRIA
Josef Urban	Radboud University Nijmegen
Tjark Weber	Uppsala University

Additional Reviewers

de Nivelle, Hans
Doczkal, Christian
Kaliszyk, Cezary
Wetzler, Nathan

External Tools for the Formal Proof of the Kepler Conjecture

Thomas C. Hales

University of Pittsburgh, USA

Abstract

The Kepler conjecture asserts that no packing of congruent balls in three-dimensional Euclidean space has density greater than that of the familiar cannonball arrangement. The proof of the Kepler conjecture was announced in 1998, but it went several years without publication because of the lingering doubts of referees about the correctness of the proof. In response to these publication hurdles, the Flyspeck project seeks to give a complete formal proof of the Kepler conjecture using the proof assistant HOL Light.

The original proof of the Kepler relies on long computer calculations, and these calculations present special formalization challenges. A major part of the Flyspeck project requires the integration of external computational tools with the proof assistant. Some of these external tools are the GNU linear programming kit, AMPL (a modeling language for mathematical programming), Mathematica calculations, nonlinear optimization, and custom code in C++, C, C#, Java, and Objective Caml.

Earlier work by A. Solovyev has implemented efficient linear programming in HOL Light. This talk will include a description of his more recent work that automates the link between linear programming and the Flyspeck project.

LEO-II version 1.5

Christoph Benzmüller[1] and Nik Sultana[2]

[1] Freie Universität Berlin, Germany
[2] Cambridge University, UK

Abstract

Leo-II cooperates with other theorem-provers to prove theorems in classical higher-order logic. It returns hybrid proofs, which contain inferences made by Leo-II as well as the backend provers with which it cooperates. This article describes recent improvements made to Leo-II.

1 Introduction

Leo-II [9] is an automatic theorem-prover for classical higher-order logic, more precisely for Church's type theory with Choice, under Henkin semantics [1, 2]. Its cooperation with backend provers is one of its distinguishing characteristics. These provers are regularly invoked by Leo-II for help with finding a refutation. In this article we outline the current system and describe recent improvements. Further details on Leo-II's hybrid proofs are reported in [16].

2 System overview

Leo-II's calculus [16] is a higher-order adaptation of RUE (Resolution by Unification and Equality) [4]. RUE is an approach for extending a resolution calculus to interpret equality, and which allows equality literals to be processed by both resolution and unification. Furthermore, Leo-II's calculus relies on a 'Boolean aware' (or, more generally, 'theory aware') extensional preunification engine (extensional preunification is discussed in [5]). In recent versions, Leo-II's unification algorithm also interprets logical constants — for example, the algorithm in version 1.5 treats disjunction as a commutative function.

Leo-II accepts problems encoded in the CNF (clausal first-order form) and FOF (first-order form) languages from the TPTP [18], but its principal input language is THF0, core typed higher-order form [19].

The logical organisation of the prover is illustrated in Figure 1, and corresponds to the modular organisation of the code. It is structured into four layers, as the figure shows:

Operating mode. The prover can be operated in two ways: (i) Leo-II can be used as a proof assistant when run in *interactive mode*. It provides a command interface through which the user can inspect and manipulate the prover's state, making calls to the calculus' rules as needed. This mode is very valuable for exploring logical problems and for debugging the prover's automatic mode. (ii) The prover is usually run in *automatic mode*: this comprises a set of strategy schedules, and a main loop which drives applications of the calculus' rules.

Prover interface. Both modes use a common infrastructure: they parse a problem and load it into the prover's state, then further manipulate the state by executing *commands*. A command might involve carrying out an inference, inspecting the state, switching flags, calling external provers, etc. Each command makes calls to lower levels of the prover.

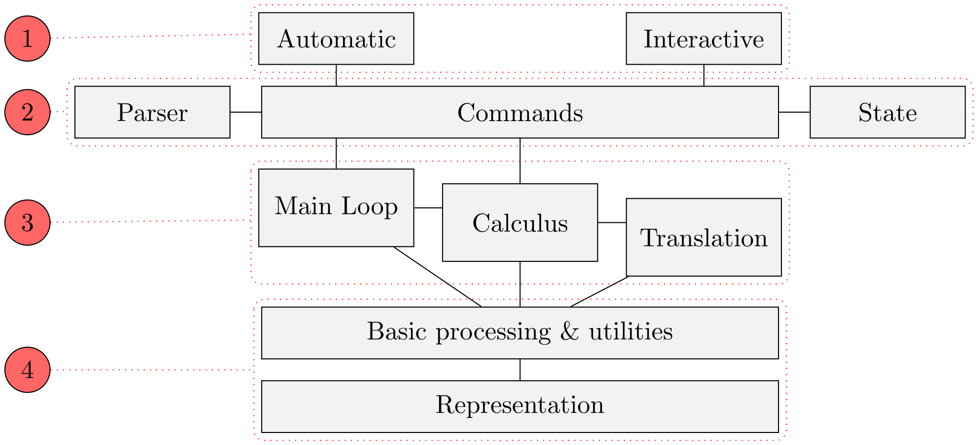


Figure 1: Leo-II's architecture

Logic. The main component in this level consists of the calculus: a collection of functions which accept and return clauses. This level also contains Leo-II's main loop, and an interface to external ATPs (which also translates problems to other formats).

Basis. The lowest level of Leo-II defines the representation of terms and types, and associated operations (e.g. substitution, unification, matching, etc).

3 Improvements

The TPTP problem set [18] is the canonical benchmark by which theorem provers are evaluated. The improvements described in this section are often accompanied by TPTP problem names whose solution is affected by the improvement. These problems consist of THF problems (more precisely, THF0 problems) drawn from TPTP 5.4.0. We have used E version 1.6 as the backend ATP. Our tests were run on a 2GHz AMD Opteron with 4GB RAM, and given 60-second timeout. Leo-II was compiled with OCaml 3.11.2.

3.1 ATP interface

Leo-II cooperates with other provers in order to maximise its potential. Recall that Leo-II proves a theorem by refuting a set of clauses. It gradually accumulates a set of clauses, some of which are first-order; a refutation in these first-order clauses will refute the overall problem. Instead of attempting to refute first-order clauses itself, Leo-II invokes external first-order ATPs to do this, since they are likely to do a far better job than Leo-II on such problems. This leaves Leo-II to focus on higher-order reasoning.

We improved Leo-II's translation to FOL in recognition of this benefit. Version 1.5 includes a better translation into FOF, and added an experimental translation into TFF [20] (a TPTP syntax for sorted first-order logic). We also improved the system interface with backend ATPs, and experimented with additional backends.

3.1.1 Translation into FOL.

It benefits Leo-II to use a translation into FOL which returns the strongest set of clauses, as long as that translation is sound.

Alongside the old translations which were previously implemented in Leo-II, version 1.5 features a new translation module which was written from scratch. This module contains an intermediate language to which problems are first translated, before being transformed further and printed into a specific target syntax. HOL-to-FOL translations consist of a pipeline of functions which bring HOL formulas into this intermediate language, applying analyses and transformations along the way.

Leo-II's old and new FOF encodings can be used through the command-line arguments `--translation fully-typed` and `--translation fof_full` respectively. In version 1.5, the translation `fof_full` is now set as default. The old translation had some undesirable qualities which harmed the performance of the FOL ATPs with which Leo-II cooperated:

1. For some examples, the old translation did omit certain necessary information in its output to the ATP. This information is of two kinds: the first relates to proxy terms, and the second relates to λ-terms.

   Here is a trivial example: when Leo-II is asked to prove

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

   and use the old translation, it would send a single clause to the ATP (after transforming the negated conjecture in its input processing into `~($true)`):

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

   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` (i.e., propositions), where '`o`' itself is a term in the language. That is, the translation encodes types as first-order terms. The constant `leoTi` is used to lift propositional terms (i.e., those typed '`o`') into formulas. Unfortunately, Leo-II did not include an axiom to give semantics to `true`; such as

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

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

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

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

   That is, it notices that instead of using `leoLit` to encode "true", it can simply use the FOF constant with that denotation. When it becomes necessary to use a proxy term such as `true`, then it includes an axiom giving its semantics. For instance, while attempting to prove

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

Leo-II's output to the ATP will include the axiom

```
fof(prox_true1, plain, ($true <=> leoLit(leoTi(true, o)))).
```

Note that Leo-II's current behaviour is not perfect either: Leo-II should be able to spot trivial refutations (as the first one above), and avoid invoking the FOL ATP and instead use its own refutation mechanism only.

The second kind of information relates to the reduction of 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 mentioned earlier, the output of the previous translation includes axioms such as

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

The encoded type `leoFt(o,o)` indicates that the constant `abstrSX0SX0` is of type $o \rightarrow o$. The name `abstrSX0SX0` is derived from serialising the term $\lambda \mathrm{SX}_0 . \mathrm{SX}_0$, but no further definition of this constant is given by the translation. The new translation λ-lifts such terms fully, yielding the pair of axioms

```
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)) )).
```

2. The old translation was verbose, and its use potentially resulted in fairly large first-order formulas due to the encoding of type information. This verbosity causes additional overhead to the ATPs, and this contributes to ATPs missing their timeout to find a refutation. Arguably, the new translation is more verbose, since it tends to include more information. To address this problem we are experimenting with lighter encoding of type information. We have closely followed Claessen et al [11] to implement their monotonicity analysis by producing a SAT encoding, which we send to MiniSat using an interface adapted from Satallax [10, 3]. This translation can be used by giving Leo-II the argument `--translation fof_experiment`.

Problems which become provable in LEO-II using the new `fof_full` translation include NUM636ˆ1.p and LCL631ˆ1.p.

3.1.2 Backend ATPs.

Leo-II is mainly used in combination with E [15], and Leo-II version 1.5 features small improvements in how it interacts with E. Support for SPASS [21] was added during past experiments [8]. In version 1.5 we improved Leo-II's ATP interface and added support for various other backend ATPs, including remote provers on SystemOnTPTP [18].

3.2 Support for Axiom of Choice

The default semantics for THF0 is Henkin semantics with choice. Until version 1.5, LEO-II did not support reasoning with choice, unless naïve Skolemization was used—that is, first-order Skolemization without employing further restrictions (as investigated by Miller [12]). This enables limited reasoning with choice, and succeeds in some example cases, but it fails in many others [6, Section 3.2].

In order to extend LEO-II to support the axiom (scheme) of choice (AC), instances of AC could be automatically added to the input problem. An example is the following instance of AC for type $(\iota \to o) \to \iota$ (where o is the type of propositions as before, and ι is the type of individuals):

$$\exists E_{(\iota\to o)\to\iota} \forall P_{(\iota\to o)}.\ \exists X_\iota (P\ X) \Rightarrow P\ (E\ P) \tag{1}$$

However, such kinds of impredicative axioms should generally be avoided in automated proof search since they allow for simulation of the cut rule in any Henkin-complete THF prover [7].

Our approach involves adding two new rules to LEO-II: detectChoiceFn and choice. The first rule detects and removes instances of AC, such as (1) above, and keeps a register of choice functions CFs. CFs always contains at least one choice function symbol for each choice type. The second rule gives the semantics to choice functions. Taken together, these rules allow AC-valid reasoning without the risk of cut-simulation.

In more detail, rule detectChoiceFn removes choice-axiom clauses from the search space and registers the corresponding choice function symbols f in CFs.

$$\frac{[PX]^{\mathtt{ff}} \vee [P(f_{(\alpha\to o)\to\alpha} P)]^{\mathtt{tt}}}{\mathsf{CFs} \longleftarrow \mathsf{CFs} \cup \{f_{(\alpha\to o)\to\alpha}\}}\ \mathsf{detectChoiceFn}$$

In the notation used above, α is a metavariable ranging over types. $P_{\alpha\to o}$ is a set variable. Literals are enclosed in square brackets, and $\mathtt{ff}$ and $\mathtt{tt}$ indicate negative and position polarity respectively. The rule abuses standard notation: the rule does not describe a logical inference, since the conclusion of the rule indicates a side-effect which extends the set CFs of choice functions.

Rule choice investigates whether a term $\epsilon_{(\alpha\to o)\to\alpha}\mathbf{B}_{\alpha\to o}$ (where $\epsilon \in \mathsf{CFs}$ is a registered choice function or a free variable) is contained as a subterm of a literal $[\mathbf{A}]^p$ in a clause C. In this case it adds the instantiation of AC at type $(\alpha \to o) \to \alpha$, and with term $\mathbf{B}_{\alpha\to o}$ to the search space. Side-conditions guard against unsound reasoning, such as the 'uncapturing' of free variables in $\mathbf{B}$:

$$\frac{C := \mathbf{C}' \vee [\mathbf{A}[E_{(\alpha\to o)\to\alpha}\mathbf{B}]]^p \qquad \begin{array}{c} \epsilon \in \mathsf{CFs},\ E = \epsilon\ \mathit{or}\ E \in \mathit{freeVars}(C), \\ \mathit{freeVars}(\mathbf{B}) \subseteq \mathit{freeVars}(C), Y\ \mathit{fresh} \end{array}}{[\mathbf{B}\ Y]^{\mathtt{ff}} \vee [\mathbf{B}\ (\epsilon_{(\alpha\to o)\to\alpha}\mathbf{B})]^{\mathtt{tt}}}\ \mathsf{choice}$$

Rules detectChoiceFn and choice are obviously sound: detectChoiceFn simply removes clauses from the search space, and for any choice function f, the rule choice only introduces new instances of the corresponding choice axiom.

There is a correspondence with the handling of choice in Satallax. Satallax too considers only selective instantiations of AC in order to avoid cut-simulation. For instance, when (1) is assumed, the terms $\mathbf{T}$ which Satallax considers to be eligible instantiations for variable P are those occurring in formulas of the following forms in a tableau branch (and where ϵ is a choice function): $(\epsilon\ \mathbf{T})\ \mathbf{S_1}\ \ldots\ \mathbf{S_n}$ or $\neg((\epsilon\ \mathbf{T})\ \mathbf{S_1}\ \ldots\ \mathbf{S_n})$, or the disequations $(\epsilon\ \mathbf{T})\ \mathbf{S_1}\ \ldots\ \mathbf{S_n} \neq \mathbf{S}$ or $\mathbf{S} \neq (\epsilon\ \mathbf{T})\ \mathbf{S_1}\ \ldots\ \mathbf{S_n}$. It is easy to see that our rule choice, which is less restrictive, subsumes these cases. We also experimented with Satallax's approach in Leo-II but this led to worse results. Our choice rule is more closely related to that of Mints [13]. Use of the choice rules can be disabled using the `-nuc` command-line switch. A completeness proof for LEO-II's improved handling of choice remains future work.

Problems which become provable in LEO-II using our improved support for choice include SYO517^1.p, SYO534^1.p–SYO537^1.p, and SYO555^1.p.

3.3 Detection of defined equality

Primitive equality in HOL refers to the use of the interpreted constant '='. Equality can also be *defined* in HOL—for example, as

$$\lambda X_\alpha \lambda Y_\alpha \forall P_{\alpha\to o}.\ P\ X \Rightarrow P\ Y$$

or

$$\lambda X_\alpha \lambda Y_\alpha \forall Q_{\alpha\to\alpha\to o}.\ \forall Z_\alpha (Q\ Z\ Z) \Rightarrow Q\ X\ Y$$

The former is known as Leibniz equality and the latter we call Andrews equality (cf. [1], Exercise X5303). Both Leibniz and Andrews equality support cut-simulation due to their impredicative nature [7], and should thus be avoided in proof automation. In fact, using primitive, rather than defined, equality may save many *primitive substitution* steps in proofs. Such steps involve instantiations of set variables, and this generally involves blind guessing. Examples of the benefit of using primitive, rather than defined, equality have been given in the literature [6, Sections 5.1 and 5.2]. In order to address this issue we added the following two rules to Leo-II's calculus; they instantiate the variable P with primitive equality:

$$\frac{\mathbf{C} \vee [P\ \mathbf{A}]^{\mathrm{ff}} \vee [P\ \mathbf{B}]^{\mathrm{tt}}}{\mathbf{C}\{\lambda X.\ \mathbf{A} = X/P\} \vee [\mathbf{A} = \mathbf{B}]^{\mathrm{tt}}}\ \mathsf{LeibEQ} \qquad \frac{\mathbf{C} \vee [P\ \mathbf{A}\ \mathbf{A}]^{\mathrm{ff}}}{\mathbf{C}\{\lambda X \lambda Y.\ X = Y/P\}}\ \mathsf{AndrEQ}$$

Soundness of LeibEQ and AndrEQ is obvious, since both rules simply realise specific instances of primitive substitution. For improved configurability, either rule can be individually disabled from the command-line by using the switches `-nrleq` and `-nraeq` respectively. If LeibEQ is used in combination with the new FOF translations (see Section 3.1) several TPTP problems whose previous SZS [17] status was 'Unknown' can now be solved by Leo-II. Examples include SYO246^5.p, SYO244^5.p, NUM817^5.p, NUM816^5.p and NUM814^5.p. There are also many problems that can now be solved with primitive substitution (blind guessing) disabled when LeibEQ and AndrEQ are available. Examples include SEV081^5.p, SEV158^5.p, SEV992^1.p, and SYO276^5.p. Overall, these two new rules lead to significantly better coverage using the lighter primitive-substitution search modes `-ps 0` or `-ps 1`.

3.4 Strategy scheduling

Strategy schedules were added to Leo-II in version 1.2 and the catalogue of schedules has slowly increased in the versions that followed. In version 1.5 we recoded the strategy-scheduling feature to facilitate the encoding of new strategies, to improve code reuse with other parts of Leo-II, and to have greater flexibility when encoding strategies. The new setup affords greater flexibility: for example, the new setup can schedule varying number of strategies (depending on the problem being processed) and each schedule could be of varying duration. This has opened up many opportunities for experimentation and tuning.

We are also interested in computing strategies on-the-fly based on problem characteristics, and version 1.5 carries out some small initial checks (e.g. size of the problem, and whether it contains instances of AC), and schedules strategies based on that limited analysis. Optimising this further remains as future work.

3.5 Other improvements

Numerous other additions were made to Leo-II. Previously, Leo-II was entirely focused on refutation: that is, until version 1.5, in terms of the SZS classification, Leo-II would judge a problem to be a Theorem (if a refutation exists), Unsatisfiable (if the problem's axioms themselves can be refuted), or diverge (by extending the preunification depth and reattempting a refutation). It can now classify Satisfiable problems and detect CounterSatisfiable problems, thus improving both Leo-II's precision and termination behaviour. The added support for choice was very relevant for achieving this. Leo-II decides that a problem is Satisfiable when the problem consists of a collection of axioms (lacking a conjecture) and Leo-II succeeds in saturating a set of clauses (without finding a refutation, otherwise the problem would be classified as Unsatisfiable).

Leo-II's unification algorithm has been redone, and can be set (from the command-line) to disregard Boolean and functional extensionality. This has strengthened Leo-II's behaviour in non-extensional problems, since disabling the extensional behaviour shrinks the search space.

Numerous other improvements and fixes have been made: these range from system features (such as the parser, status reporting, avoiding redundant computations, etc) to deeper areas in the calculus and main loop (including factorisation, subsumption, and clause selection).

4 Future work

We have started experimenting with using term orderings to influence literal selection. We also plan to revise Leo-II's internals to make full use of the potential benefit they offer. For instance, the shared term graph is currently underutilised.

More work is needed to compute better schedules, paired with better problem analyses. Such analyses can determine the scheduling of specific strategies, which can be better tuned to the problem.

The ATP interface can be improved further to call multiple backend ATPs in parallel. Experiments comparing 30-second invocations of Leo-II on all THF problems, supported by provers E (version 1.6), SPASS (version 3.5) [21] and Vampire (version 2.6) [14] showed us that there were 37, 5 and 20 theorems that were proved exclusively by Leo-II(E), Leo-II(SPASS) and Leo-II(Vampire), respectively. And there were 31, 95 and 98 theorems that Leo-II(E), Leo-II(SPASS) and Leo-II(Vampire) missed, but which one of the others could prove.

Supporting various ATP backends increases the scope for peephole optimisation; we have not yet investigated this. The translation module can be optimised further, and extended to target more formats. Table 1 one shows how the new HOL-to-FOL translation (`fof_full`) and its lighter variant (`fof_experiment`) are superior to Leo-II's preexisting encoding (`fully_typed`). In future work we plan to improve `fof_experiment` further and make it the default translation.

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

Table 1: Comparing FOL encodings in Leo-II 1.5 (30s timeout). Table shows the percentage of matches between Leo-II's SZS output and the 'Status' field of problems.

Timeout (s)	**v1.2**		**v1.4.3**		**v1.5**	
	Thm	*All*	*Thm*	*All*	*Thm*	*All*
30	58.4	51.1	62.1	54.4	64.3	61.3
60	58.7	51.3	65	56.9	67.1	62.9

Table 2: Percentage match between different versions of Leo-II and the Status field of TPTP problems. Leo-II version 1.2 was the winner of the CASC competition in 2010, and version 1.4.3 was the last public release. Version 1.5 was run with the `fof_experiment` encoding.

5 Conclusion

Version 1.5 of Leo-II includes various improvements which affect its performance and coverage. To obtain a broader picture, we compared the results of using Leo-II version 1.5 with earlier versions, and the results are shown in Table 2. In this experiment we counted the matches between Leo-II's SZS output and the TPTP problem's SZS status (included in its header).[1] All the net gains are positive, but a more thorough evaluation (on different benchmarks, and considering various parameters) remains as future work. Within a 30s timeout, Leo-II version 1.5 can classify 196 more problems than its predecessor. The main boost ($\frac{125}{196}$ problems) in this version is provided by the detection of non-theorems (i.e. satisfiable or countersatisfiable problems).

Acknowledgement. Work on Leo-II was supported by the UK Engineering and Physical Sciences Research Council (grant number EP/D070511/1), and the German Research Foundation under grant BE 2501/9-1. The second author was supported by a grant from the German Academic Exchange Service (DAAD) during a visit to Freie Universität Berlin, during which the work described in this paper was carried out. We thank Chad Brown for allowing us to use Satallax's MiniSat interface, and thank Mike Gordon, Grant Passmore, Larry Paulson, and the anonymous referees for feedback on this article.

References

[1] P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Applied Logic Series. Springer, 2002.

[2] P.B. Andrews. Church's type theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2009 edition, 2009.

[3] J. Backes and C.E. Brown. Analytic tableaux for higher-order logic with choice. *Journal of Automated Reasoning*, 47(4):451–479, 2011.

[4] C. Benzmüller. Extensional higher-order paramodulation and RUE-resolution. In *Proceedings of CADE-16*, number 1632 in LNCS, pages 399–413. Springer, 1999.

[1] This also means that 'Unknown' problems which Leo-II now classifies as 'Theorem' count against us, but this experiment was only intended to offer a rough idea of progress.

[5] C. Benzmüller. Comparing approaches to resolution based higher-order theorem proving. *Synthese*, 133(1-2):203–235, 2002.

[6] C. Benzmüller and C.E. Brown. A structured set of higher-order problems. In *Proceedings of TPHOLs 2005*, number 3603 in LNCS, pages 66–81. Springer, 2005.

[7] C. Benzmüller, C.E. Brown, and M. Kohlhase. Cut-simulation and impredicativity. *Logical Methods in Computer Science*, 5(1:6):1–21, 2009.

[8] C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. Progress report on LEO-II – an automatic theorem prover for higher-order logic. In *TPHOLs 2007 Emerging Trends Proceedings*, pages 33–48. Internal Report 364/07, Department of Computer Science, University Kaiserslautern, Germany, 2007.

[9] C. Benzmüller, F. Theiss, L. Paulson, and A. Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In *Proceedings of IJCAR 2008*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008.

[10] C.E. Brown. Reducing higher-order theorem proving to a sequence of sat problems. *Journal of Automated Reasoning*, 51(1):57–77, 2013.

[11] K. Claessen, A. Lillieström, and N. Smallbone. Sort it out with monotonicity: translating between many-sorted and unsorted first-order logic. In *Proceedings of CADE 2011*, pages 207–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] D.A. Miller. *Proofs in Higher-Order Logic.* PhD thesis, Carnegie Mellon University, 1983.

[13] G. Mints. Cut-elimination for simple type theory with an axiom of choice. *Journal of Symbolic Logic*, pages 479–485, 1999.

[14] A. Riazanow and A. Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2/3):91–110, 2002.

[15] S. Schulz. E – A Brainiac Theorem Prover. *AI Communications*, 15(2/3):111–126, 2002.

[16] N. Sultana and C. Benzmüller. Understanding LEO-II's proofs. In E. Ternovska, K. Korovin, and S. Schulz, editors, *The 9th International Workshop on the Implementation of Logics (IWIL-2012, affiliated with LPAR-2012)*, Merida, Venezuela, 2012.

[17] G. Sutcliffe. The SZS ontologies for automated reasoning software. In *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, volume 418, pages 38–49. CEUR Workshop Proceedings, 2008.

[18] G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[19] G. Sutcliffe and C. Benzmüller. Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. *Journal of Formalized Reasoning*, 3(1):1, 2010.

[20] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP typed first-order form with arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *LNCS*, pages 406–419. Springer, 2012.

[21] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In *Proceedings of the CADE-22*, volume 5663 of *LNAI*, pages 140–145. Springer, 2009.

Redirecting Proofs by Contradiction

Jasmin Christian Blanchette

Technische Universität München, Germany

Abstract

This paper presents an algorithm that redirects proofs by contradiction. The input is a refutation graph, as produced by an automatic theorem prover (e.g., E, SPASS, Vampire, Z3); the output is a direct proof expressed in natural deduction extended with case analyses and nested subproofs. The algorithm is implemented in Isabelle's Sledgehammer, where it enhances the legibility of machine-generated proofs.

1 Introduction

The proofs returned by automatic theorem provers (ATPs) are notoriously difficult to read. This is an issue if an ATP solves an open mathematical problem (such as the Robbins conjecture [11]), because users then certainly want to study the proof closely. But even in the context of program verification, where users are normally satisfied with a "proved" or "disproved," they might still want to analyze proofs—for example, if they suspect errors in their axiom set.

Our interest in intelligible ATP proofs has a different origin. The tool Sledgehammer [19] integrates ATPs with the proof assistant Isabelle/HOL [15]. Given an Isabelle conjecture, Sledgehammer heuristically selects relevant lemmas from Isabelle's libraries, translates them along with the conjecture to first-order logic, and sends the resulting problem to state-of-the-art provers such as E [24], SPASS [28], Vampire [22], and Z3 [5].

To guard against bugs in the ATPs and in Sledgehammer's translation module, ATP proofs are reconstructed in Isabelle. This is accomplished through either a single invocation of the built-in resolution prover *metis* [8] or a structured Isar proof [20]. The latter option is useful for larger proofs, which *metis* fails to re-find within a reasonable time. But most users find the proofs unattractive and are disinclined to insert them in their theory text. As an illustration, consider the conjecture $\mathsf{length}\ (\mathsf{tl}\ xs) \le \mathsf{length}\ xs$, which states that the tail of a list (the list from which we remove its first element, or the empty list if the list is empty) is at most as long as the full list. The proof found by Vampire, translated to Isar, is as follows:

```
proof neg_clausify
  assume length (tl xs) ≰ length xs
  hence drop (length xs) (tl xs) ≠ [] by (metis drop_eq_Nil)
  hence tl (drop (length xs) xs) ≠ [] by (metis drop_tl)
  hence ∀u. xs @ u ≠ xs ∨ tl u ≠ [] by (metis append_eq_conv_conj)
  hence tl [] ≠ [] by (metis append_Nil2)
  thus False by (metis tl.simps(1))
qed
```

(The function drop n removes the first n elements from a list.) The *neg_clausify* proof method puts the Isabelle conjecture into negated clause form to ensure that it has the same shape as the corresponding ATP conjecture. The negation of the clause is introduced in the `assume` line, and a sequence of intermediate facts each introduced by `hence` leads to a contradiction.

There is a considerable body of research about making resolution proofs intelligible. Early work focused on translating detailed resolution proofs into natural deduction calculi [13, 21]. Although they are arguably more readable, these calculi operate at the logical level, whereas humans reason mostly at the "assertion level," invoking definitions and lemmas without providing the full logical details. A line of research focused on transforming natural deduction proofs into assertion-level proofs [1, 7], culminating with the systems TRAMP [12] and Otterfier [31]. More related work includes the identification of obvious inferences [4,23], the successful transformation of EQP's proof of the Robbins conjecture using ILF [3], and more recently the use of TPTP-based tools to present Mizar articles [27].

It would have been interesting to try out TRAMP and Otterfier, but these are large pieces of unmaintained software that are hardly installable on modern machines and that only support older ATP systems. Regardless, the problem looks somewhat different in the context of Sledgehammer. Because the provers are given hundreds of lemmas as axioms, they tend to find short proofs with few lemmas. Moreover, Sledgehammer can coalesce consecutive inferences if short proofs are desired. Replaying an inference is a minor issue, thanks to *metis*.

The first obstacle to readability is that the Isar proof, like the underlying ATP proof, is by contradiction. This paper presents an algorithm for transforming proofs by contradiction into direct proofs—or *redirecting* proofs—to improve intelligibility. Knuth, Larrabee, and Roberts call the unnecessary use of proof by contradiction a sin against mathematical exposition [10, § 3]—but since redirection is always possible, what would a necessary use look like?

The redirection algorithm is not be tied to any one calculus or logic, as long as it admits contraposition. In particular, it works on the Isar proofs generated by Sledgehammer or directly on first-order TSTP proofs [26]. The direct proofs are expressed in a simple Isar-like syntax, which can be regarded as natural deduction extended with case analyses and nested subproofs (Section 2). The algorithm is first demonstrated on a few examples (Section 3) before it is presented in more detail, both in prose and as Standard ML pseudocode (Section 4).

For examples with a simple linear structure, such as the Isar proof above, the proof can be turned around by applying contraposition repeatedly:

```
proof –
  have $\mathsf{tl}\ [] = []$ by ($metis\ tl.simps(1)$)
  hence $\exists u.\ xs\ @\ u = xs \wedge \mathsf{tl}\ u = []$ by ($metis\ append\_Nil2$)
  hence $\mathsf{tl}\ (\mathsf{drop}\ (\mathsf{length}\ xs)\ xs) = []$ by ($metis\ append\_eq\_conv\_conj$)
  hence $\mathsf{drop}\ (\mathsf{length}\ xs)\ (\mathsf{tl}\ xs) = []$ by ($metis\ drop\_tl$)
  thus $\mathsf{length}\ (\mathsf{tl}\ xs) \leq \mathsf{length}\ xs$ by ($metis\ drop\_eq\_Nil$)
qed
```

The direct proof is easier to understand than the indirect one, even though it does not quite look like a human-written proof—humans would most likely avoid the detour through drop.

The approach works on arbitrary proofs by contradiction. A prototype demonstrated at earlier workshops [18, 19] sometimes exhibited exponential behavior. This has been resolved: Excluding a linear number of additional inferences that justify case analyses, each inference in the proof by contradiction now gives rise to exactly one inference in the direct proof. The algorithm can easily process proofs with hundreds or thousands of inferences.

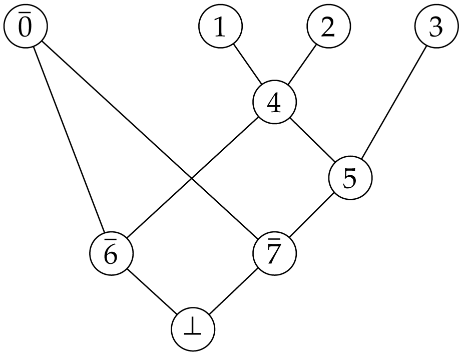
The algorithm is implemented in Sledgehammer. It is also described in Section 6.8 of my Ph.D. thesis [2], whose text largely forms the basis of this paper.

2 Proof Notations

Proof Graph. ATP proofs identify formulas by numbers. There may be several conjectures, in which case they are interpreted disjunctively. The negated conjectures and user-provided axioms are numbered 0, 1, 2, ..., $n-1$, and the derivations performed during proof search (whether or not they participate in the final proof) are numbered sequentially from n. We abstract the ATP proofs by ignoring the formulas and keeping only the numbers. We call formulas *atoms* since we are not interested in their structure. The letters a, b denote atoms.

An atom is *tainted* if it is one of the negated conjectures or has been derived, directly or indirectly, from a negated conjecture. For convenience, we relabel the ATP proof's atoms so that tainted atoms are decorated with a bar, denoting negation. Thus, if atom 3, corresponding to the formula ϕ, is tainted, it is relabeled to $\overline{3}$, but it still stands for ϕ and is called an atom despite the negative bar. After the relabeling, removing the bar negates the formula: $3 \equiv \neg\,\phi$.

A proof graph is a directed acyclic graph in which an edge $a \rightarrow a'$ indicates that atom a is used to derive atom a'. Proof graphs are required to have exactly one sink node, whose formula is $\bot$, and only one connected component. We adopt the convention that derived nodes appear lower than their parent nodes in the graph and omit the arrowheads:



It is natural to write $\bot$ rather than a numeric label for the sink node in examples.

In Sledgehammer, unary inferences are collapsed, and the first-order formulas are translated back to HOL before the proof is redirected. This is outside the scope of this paper and is explained in more detail in a companion paper [25].

Isar Proofs. Proof graphs cannot represent proofs by case analysis and only serve for the redirection algorithm's input. We need more powerful notations for the output. Isar proofs [14, §4; 29] are a linear representation of natural deduction proofs in the style of Jaśkowski [9]. Unlike Gentzen-style trees [6], they allow the sharing of common derivations. The proof on the left-hand side is by contradiction; that on the right-hand side is the corresponding direct proof:

```
proof $neg\_clausify$
  assume $\overline{0}$
  have 4 by ($metis$ 1 2)
  have 5 by ($metis$ 3 4)
  have $\overline{6}$ by ($metis$ $\overline{0}$ 4)
  have $\overline{7}$ by ($metis$ $\overline{0}$ 5)
  show $\bot$ by ($metis$ $\overline{6}$ $\overline{7}$)
qed
```

```
proof –
  have 4 by ($metis$ 1 2)
  have 5 by ($metis$ 3 4)
  have $6 \lor 7$ by $metis$
  moreover
  { assume 6
    have 0 by ($metis$ 4 6) }
  moreover
  { assume 7
    have 0 by ($metis$ 5 7) }
  ultimately show 0 by $metis$
qed
```

Notice that the direct proof involves a 2-way case analysis on a disjunction. Generalized disjunctions of the form $a_1 \lor \cdots \lor a_m$ are called *clauses* and are denoted by the letters c, d, e. Clauses are considered equal modulo associativity, commutativity, and idempotence. Sets of clauses are denoted by Γ.

Proof redirection requires that inferences can be redirected using the contrapositive but otherwise makes no assumptions about the proof calculus. Inferences that introduce new symbols can also be redirected; for example, skolemization becomes "un-herbrandization" [25, §4].

Shorthand Proofs. The last proof format is an ad hoc shorthand notation for a subset of Isar. In their simplest form, these shorthand proofs are a list of derivations $c_1, \ldots, c_m \rhd c$ whose intuitive meaning is: "From the hypotheses c_1 and … and c_m, infer c." The clauses on the left-hand side are interpreted as a set Γ.

If a hypothesis c_i is the previous derivation's conclusion, we can omit it and write $\blacktriangleright$ instead of $\rhd$. This notation mimics Isar, with $\rhd$ for `have` or `show` and $\blacktriangleright$ for `hence` or `thus`. Depending on whether we use the abbreviated format, our running example becomes

$$\begin{array}{rcl@{\qquad\qquad\qquad}rcl}
1,2 & \rhd & 4 & 1,2 & \rhd & 4 \\
3,4 & \rhd & 5 & 3 & \blacktriangleright & 5 \\
\overline{0},4 & \rhd & \overline{6} & \overline{0},4 & \rhd & \overline{6} \\
\overline{0},5 & \rhd & \overline{7} & \overline{0},5 & \rhd & \overline{7} \\
\overline{6},\overline{7} & \rhd & \bot & \overline{6} & \blacktriangleright & \bot
\end{array}$$

Each derivation $\Gamma \rhd c$ is essentially a sequent with Γ as the antecedent and c as the succedent. For proofs by contradiction, the clauses in the antecedent are either the negated conjecture ($\overline{0}$), atoms that correspond to background facts (1, 2, and 3), or atoms that were proved in preceding sequents (4, 5, $\overline{6}$, and $\overline{7}$); the succedent of the last sequent is always $\bot$.

Direct proofs can be presented in the same way, but the negated conjecture $\overline{0}$ may not appear in any of the sequents' antecedents, and the last sequent must have the conjecture 0 as its succedent. In some of the direct proofs, it is useful to introduce case analyses. For example:

$$\begin{array}{r}
1,2 \rhd 4 \\
3 \blacktriangleright 5 \\
\rhd 6 \vee 7
\end{array}$$

$$\left[\begin{array}{r|r}
[6] & [7] \\
4 \blacktriangleright 0 & 5 \blacktriangleright 0
\end{array}\right]$$

In general, case analysis blocks have the form

$$\left[\begin{array}{r|c|r}
[c_1] & \cdots & [c_m] \\
\Gamma_{11} \rhd d_{11} & \cdots & \Gamma_{m1} \rhd d_{m1} \\
\vdots & & \vdots \\
\Gamma_{1k_1} \rhd d_{1k_1} & \cdots & \Gamma_{mk_m} \rhd d_{mk_m}
\end{array}\right]$$

with the requirement that a sequent with the succedent $c_1 \vee \cdots \vee c_m$ has been proved immediately above the case analysis. Each of the branches must also be a valid proof. The assumptions $[c_i]$ may be used to discharge hypotheses in the same branch, as if they had been sequents $\rhd c_i$. The case analysis will sometimes be regarded as a sequent

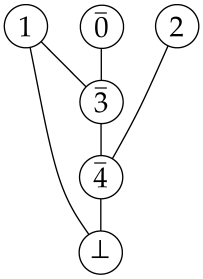
$$c_1 \vee \cdots \vee c_m,\ \bigcup_{i,j} \big(\Gamma_{ij} - c_i - \bigcup_{j'<j} d_{ij'}\big)\ \rhd d_{1k_1} \vee \cdots \vee d_{mk_m}$$

by ignoring its internal structure.

3 Examples of Proof Redirection

Before reviewing the redirection algorithm, we consider four examples of proofs by contradiction and redirect them to produce a direct proof. The first example has a simple linear structure, the second and third examples involve a "lasso," and the last example has no apparent structure.

A Linear Proof. We start with a simple proof by contradiction expressed as a proof graph and in our shorthand notation:



$$\begin{array}{r}
\overline{0},1 \rhd \overline{3} \\
2,\overline{3} \rhd \overline{4} \\
1,\overline{4} \rhd \bot
\end{array}$$

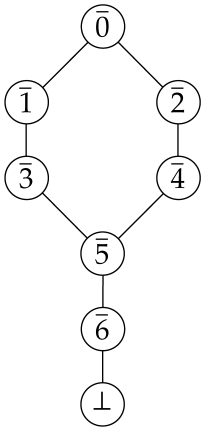
We redirect the sequents using sequent-level contraposition to eliminate all taints (represented as bars after the relabeling). This gives

$$\begin{array}{r@{\ }c@{\ }l} 1,3 & \rhd & 0 \\ 2,4 & \rhd & 3 \\ 1 & \rhd & 4 \end{array}$$

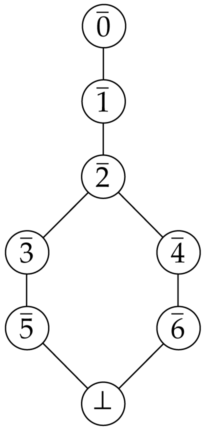
We then obtain the direct proof by reversing the order of the sequents, and introduce $\blacktriangleright$ wherever possible:

```
proof –
  have 4 by (metis 1)        1 ▷ 4
  hence 3 by (metis 2)       2 ▶ 3
  thus 0 by (metis 1)        1 ▶ 0
qed
```

Lasso-Shaped Proofs. The next two examples look superficially like lassos but are of course acyclic, as required of all proof graphs:



$$\begin{array}{r@{\ }c@{\ }l} \overline{0} & \rhd & \overline{1} \\ \overline{0} & \rhd & \overline{2} \\ \overline{1} & \rhd & \overline{3} \\ \overline{2} & \rhd & \overline{4} \\ \overline{3},\overline{4} & \rhd & \overline{5} \\ \overline{5} & \rhd & \overline{6} \\ \overline{6} & \rhd & \bot \end{array}$$



$$\begin{array}{r@{\ }c@{\ }l} \overline{0} & \rhd & \overline{1} \\ \overline{1} & \rhd & \overline{2} \\ \overline{2} & \rhd & \overline{3} \\ \overline{2} & \rhd & \overline{4} \\ \overline{3} & \rhd & \overline{5} \\ \overline{4} & \rhd & \overline{6} \\ \overline{5},\overline{6} & \rhd & \bot \end{array}$$

We start with the example on the left-hand side. Starting from $\bot$, it is easy to redirect the stem:

$$\begin{array}{r@{\ }c@{\ }l} & \rhd & 6 \\ 6 & \rhd & 5 \\ 5 & \rhd & 3 \vee 4 \end{array}$$

When applying the contrapositive to eliminate the negations in $\overline{3},\overline{4} \rhd \overline{5}$, we obtain a disjunction in the succedent: $5 \rhd 3 \vee 4$. To continue from there, we introduce a case analysis. In each branch, we can finish the proof:

$$\left[\begin{array}{r@{\ }c@{\ }l|r@{\ }c@{\ }l} & & [3] & & & [4] \\ 3 & \rhd & 1 & 4 & \rhd & 2 \\ 1 & \rhd & 0 & 2 & \rhd & 0 \end{array}\right]$$

In the second lasso example, the cycle occurs near the end of the contradiction proof. A disjunction already arises when we redirect the last derivation. Naively finishing each branch independently leads to a fair amount of duplication:

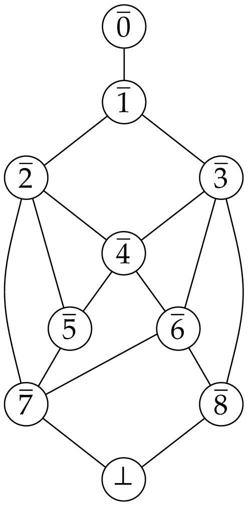
$$
\begin{array}{c}
\rhd\ 5 \lor 6 \\
\left[\begin{array}{r|r}
{[5]} & {[6]} \\
5 \rhd 3 & 6 \rhd 4 \\
3 \rhd 2 & 4 \rhd 2 \\
2 \rhd 1 & 2 \rhd 1 \\
1 \rhd 0 & 1 \rhd 0
\end{array}\right]
\end{array}
$$

The key observation is that the two branches can share the last two inferences. This yields the following proof (without and with ▶):

$$
\begin{array}{c}
\rhd\ 5 \lor 6 \\
\left[\begin{array}{r|r}
{[5]} & {[6]} \\
5 \rhd 3 & 6 \rhd 4 \\
3 \rhd 2 & 4 \rhd 2
\end{array}\right] \\
2 \rhd 1 \\
1 \rhd 0
\end{array}
\qquad\qquad
\begin{array}{c}
\rhd\ 5 \lor 6 \\
\left[\begin{array}{r|r}
{[5]} & {[6]} \\
\blacktriangleright 3 & \blacktriangleright 4 \\
\blacktriangleright 2 & \blacktriangleright 2
\end{array}\right] \\
\blacktriangleright 1 \\
\blacktriangleright 0
\end{array}
$$

Here we were fortunate that the branches were joinable on the atom 2. To avoid duplication, we must in general join on a disjunction $a_1 \lor \cdots \lor a_m$, as in the next example.

A Spaghetti Proof. The final example is diabolical (and slightly unrealistic, perhaps):



$$
\begin{array}{r@{\ }c@{\ }l}
\overline{0} & \rhd & \overline{1} \\
\overline{1} & \rhd & \overline{2} \\
\overline{1} & \rhd & \overline{3} \\
\overline{2},\overline{3} & \rhd & \overline{4} \\
\overline{2},\overline{4} & \rhd & \overline{5} \\
\overline{3},\overline{4} & \rhd & \overline{6} \\
\overline{2},\overline{5},\overline{6} & \rhd & \overline{7} \\
\overline{3},\overline{6} & \rhd & \overline{8} \\
\overline{7},\overline{8} & \rhd & \bot
\end{array}
$$

We start with the contrapositive of the last sequent:

$$\rhd\ 7 \lor 8$$

We perform a case analysis on $7 \lor 8$. Since we want to avoid duplication in the two branches, we first determine which nodes are reachable in the refutation graph by navigating upward from either $\overline{7}$ or $\overline{8}$ but not from both. The only such nodes here are $\overline{5}$, $\overline{7}$, and $\overline{8}$. In each branch, we can perform derivations of the form $\Gamma \rhd b$ where $\Gamma \cap \{5,7,8\} \neq \emptyset$ without fearing duplication. Following this rule, we can only perform one inference in the right branch before we must stop:

$$\begin{array}{c} [8] \\ 8 \rhd 3 \lor 6 \end{array}$$

Any further inferences would need to be repeated in the left branch, so it is indeed a good idea to stop. The left branch starts as follows:

$$\begin{array}{c} [7] \\ 7 \rhd 2 \lor 5 \lor 6 \end{array}$$

We would now like to perform the inference $5 \rhd 2 \lor 4$. This would certainly not lead to any duplication, because $\overline{5}$ is not reachable from $\overline{8}$ by navigating upward in the refutation graph. However, we cannot discharge the hypothesis 5, having established only the disjunction $2 \lor 5 \lor 6$. We need a case analysis on the disjunction to proceed:

$$\left[\begin{array}{c|c|c} & [5] & \\ [2] & 5 \rhd 2 \lor 4 & [6] \end{array}\right]$$

The 2 and 6 subbranches are left alone, because there is no node that is reachable only from $\overline{2}$ or $\overline{6}$ but not from the other two nodes in $\{\overline{2}, \overline{5}, \overline{6}\}$ by navigating upward in the refutation graph. Since only one branch is nontrivial, it is arguably more aesthetically pleasing to abbreviate the entire case analysis to

$$2 \lor 5 \lor 6 \rhd 2 \lor 4 \lor 6$$

Putting this all together, the outer case analysis becomes

$$\left[\begin{array}{l|l} \quad [7] & \\ \blacktriangleright\ 2 \lor 5 \lor 6 & \quad [8] \\ \blacktriangleright\ 2 \lor 4 \lor 6 & \blacktriangleright\ 3 \lor 6 \end{array}\right]$$

The left branch proves $2 \lor 4 \lor 6$, the right branch proves $3 \lor 6$; hence, both branches together prove $2 \lor 3 \lor 4 \lor 6$. Next, we perform the inference $6 \rhd 3 \lor 4$. This requires a case analysis on $2 \lor 3 \lor 4 \lor 6$:

$$\left[\begin{array}{c|c|c|c} & & & [6] \\ [2] & [3] & [4] & 6 \rhd 3 \lor 4 \end{array}\right]$$

This proves $2 \lor 3 \lor 4$. Since only one branch is nontrivial, we prefer to abbreviate the case analysis to

$$2 \lor 3 \lor 4 \lor 6 \rhd 2 \lor 3 \lor 4$$

It may help to think of such abbreviated inferences as instances of rewriting modulo associativity, commutativity, and idempotence. Here, 6 is rewritten to $3 \lor 4$ in $2 \lor 3 \lor 4 \lor 6$, resulting in $2 \lor 3 \lor 4$. Similarly, the sequent $4 \rhd 2 \lor 3$ gives rise to the case analysis

$$\left[\begin{array}{c|c|c} & & [4] \\ [2] & [3] & 4 \rhd 2 \lor 3 \end{array}\right]$$

which can be abbreviated as well. We are left with $2 \lor 3$. The rest is analogous to the second lasso-shaped proof:

$$\left[\begin{array}{c|c} [2] & [3] \\ 2 \rhd 1 & 3 \rhd 1 \end{array}\right]$$
$$1 \rhd 0$$

Putting all of this together, we obtain the following proof, expressed in Isar and in shorthand. The result is quite respectable, considering the spaghetti-like graph we started with:

```
proof –
  have 7 ∨ 8 by metis
  moreover
  { assume 7
    hence 2 ∨ 5 ∨ 6 by metis
    hence 2 ∨ 4 ∨ 6 by metis }
  moreover
  { assume 8
    hence 3 ∨ 6 by metis }
  ultimately have 2 ∨ 3 ∨ 4 ∨ 6 by metis
  hence 2 ∨ 3 ∨ 4 by metis
  hence 2 ∨ 3 by metis
  moreover
  { assume 2
    hence 1 by metis }
  moreover
  { assume 3
    hence 1 by metis }
  ultimately have 1 by metis
  thus 0 by metis
qed
```

$$\rhd\ 7 \lor 8$$
$$\left[\begin{array}{c|c} [7] & \\ \blacktriangleright\ 2 \lor 5 \lor 6 & [8] \\ \blacktriangleright\ 2 \lor 4 \lor 6 & \blacktriangleright\ 3 \lor 6 \end{array}\right]$$
$$\blacktriangleright\ 2 \lor 3 \lor 4$$
$$\blacktriangleright\ 2 \lor 3$$
$$\left[\begin{array}{c|c} [2] & [3] \\ \blacktriangleright\ 1 & \blacktriangleright\ 1 \end{array}\right]$$
$$\blacktriangleright\ 0$$

4 The Redirection Algorithm

The process we applied in the examples above can be generalized into an algorithm. The algorithm takes an arbitrary proof by contradiction expressed as a set of sequents as input, and produces a proof in our Isar-like shorthand notation, with sequents and case analysis blocks. The proof is constructed one inference at a time starting from $\top$ (the negation of $\bot$) until the conjecture—in general, the disjunction of the conjectures—is proved.

Basic Concepts. A fundamental operation is sequent-level contraposition. Let $a_1,\ldots,a_m$ be the untainted atoms and $\overline{b_1},\ldots,\overline{b_n}$ the tainted atoms of a proof by contradiction. The proof then consists of the following three kinds of sequent (with $n > 0$):

$$a_1,\ldots,a_m,\overline{b_1},\ldots,\overline{b_n} \rhd \bot \qquad a_1,\ldots,a_m,\overline{b_1},\ldots,\overline{b_n} \rhd \overline{b} \qquad a_1,\ldots,a_m \rhd a$$

Their *contrapositives* are, respectively,

$$a_1,\ldots,a_m \rhd b_1 \vee \cdots \vee b_n \qquad a_1,\ldots,a_m,b \rhd b_1 \vee \cdots \vee b_n \qquad a_1,\ldots,a_m \rhd a$$

We call the contrapositives of the sequents in the proof by contradiction the *redirected sequents*.

Based on the set of redirected sequents, we define the *atomic inference graph* (AIG) with, for each redirected sequent $\Gamma \rhd c$, an edge from each atom in Γ to each atom in c, and no additional edges. The AIG encodes the order in which the atoms can be inferred in a direct proof. Navigating forward (downward) in this graph along the unnegated tainted atoms b_j corresponds to navigating backward (upward) in the refutation graph along the $\overline{b_j}$'s.

Like the underlying refutation graph, the AIG is acyclic and connected. Potential cycles would involve either only untainted atoms a_i, only tainted atoms b_j's, or a mixture of both kinds. A cycle $a_{i_1} \to \cdots \to a_{i_k} \to a_{i_1}$ is impossible, because the contrapositive leaves these inferences unchanged and hence the cycle would need to occur in the (acyclic) refutation graph. A cycle $b_{j_1} \to \cdots \to b_{j_k} \to b_{j_1}$ is impossible, because the contrapositive turns all the edges around and hence the reverse cycle would need to occur in the refutation graph. Finally, mixed cycles necessarily involve an edge $b \to a$, which is impossible because redirected sequents with untained atoms a can only have untainted atoms as predecessors (cf. $a_1,\ldots,a_m \rhd a$).

Given a set of (tainted or untainted) atoms A, the *zone* of an atom $a \in A$ with respect to A is the set of possibly trivial descendants of a in the AIG that are not descendants of any of the other atoms in A. As a trivial descendant of itself, a will either belong to its own zone or to no zone all at (depending on whether it is a descendant of a node $a' \in A - \{a\}$). Zones identify inferences that can safely be performed inside a branch in a case analysis.

The Algorithm. The algorithm keeps track of the *last-proved clause* (initially $\top$), the set of *already proved atoms* (initially the set of facts taken as axioms), and the set of *remaining sequents* to use (initially all the redirected sequents provided as input). It performs these steps:

1. If there are no remaining sequents, stop.
2. If the last-proved clause is $\top$ or a single atom:

   2.1. Pick a sequent $\Gamma \rhd c$ among the remaining sequents that can be proved using only already proved atoms, preferring sequents with a single atom in their succedent.

   2.2. Append $\Gamma \rhd c$ to the proof.

   2.3. Make c the last-proved clause, add c to the already proved atoms if it is an atom, and remove $\Gamma \rhd c$ from the remaining sequents.

   2.4. Go to step 1.

3. Otherwise, the last-proved succedent is of the form $a_1 \vee \cdots \vee a_m$. An m-way case analysis is called for:[1]

   3.1. Compute the zone of each atom a_i with respect to $\{a_1, \ldots, a_m\}$.

   3.2. For each a_i, compute the set $\mathscr{S}_i$ of sequents $\Gamma \rhd c$ such that Γ consists only of already proved atoms or atoms within a_i's zone.

   3.3. Recursively invoke the algorithm m times, once for each a_i, each time with a_i as the last-proved clause, a_i added to the already proved atoms, and $\mathscr{S}_i$ as the set of remaining sequents. This step yields m (possibly empty) subproofs $\pi_1, \ldots, \pi_m$.

   3.4. Append the following case analysis block to the proof:

$$\left[\begin{array}{c|c|c} [a_1] & \cdots & [a_m] \\ \pi_1 & \cdots & \pi_m \end{array}\right]$$

   3.5. Make the succedent $b_1 \vee \cdots \vee b_n$ of the case analysis block (regarded as a sequent) the last-proved clause, add b_1 to the already proved atoms if $k = 1$, and remove all sequents belonging to any of the sets $\mathscr{S}_i$ from the remaining sequents.

   3.6. Go to step 1.

Whenever a redirected sequent is generated, it is removed from the set of remaining sequents. In step 3, the recursive calls operate on pairwise disjoint subsets $\mathscr{S}_i$ of the remaining sequents. Consequently, each redirected sequent appears at most once in the generated proof, and the resulting direct proof contains the same number of inferences as the initial proof by contradiction. In Isar, each case analysis is additionally justified by one *metis* inference.

In the degenerate case where no atoms are tainted (i.e., the proof exploits an inconsistency in the axiom set), the generated proof is simply a linearization of the refutation graph, and the last inference proves $\bot$ (which is, unusually, untainted). To produce a syntactically valid Isar proof, a trivial inference must be added to derive the conjecture from $\bot$.

Pseudocode. To make the above description more concrete, the algorithm is presented in Standard ML pseudocode below.[2] The pseudocode is fairly faithful to the description above. Atoms are represented by integers and literals by sets (lists) of integers. Go-to statements are implemented by recursion, and the state is threaded through recursive calls as three arguments (*last*, *earlier*, and *seqs*). One notable difference, justified by a desire to avoid code duplication, is that the set of already proved atoms, called *earlier*, excludes the last-proved clause *last*. Hence, we take *last* $\cup$ *earlier* to obtain the already proved atoms, where *last* is either the empty list (representing $\top$) or a singleton list (representing a single atom).

Shorthand proofs are represented as lists of *inference*s:

`datatype` *inference* = Have `of` *int list* × *int list* | Cases `of` (*int* × *inference list*) *list*

[1] A generalization would be to perform a m'-way case analysis, with $m' < m$, by keeping some disjunctions. For example, we could perform a 3-way case analysis with $a_1 \vee a_2$, a_3, and a_4 as the assumptions instead of breaking all the disjunctions in a 4-way analysis. This could lead to nicer proofs if the disjuncts are carefully chosen.

[2] The actual ML code is distributed with Isabelle. A recent repository version is available at `http://isabelle.in.tum.de/repos/isabelle/file/e5303bd748f2/src/HOL/Tools/ATP/atp_proof_redirect.ML`.

The main function implementing the algorithm follows:

```
fun redirect $last$ $earlier$ $seqs$ =
  if null $seqs$ then
    []
  else if length $last \leq 1$ then
    let val $provable$ = filter (fn $(\Gamma, \_) \Rightarrow \Gamma \subseteq last \cup earlier$) $seqs$
          val $horn\_provable$ = filter (fn $(\_, [\_]) \Rightarrow$ true | $\_ \Rightarrow$ false) $provable$
          val $(\Gamma, c)$ = hd ($horn\_provable$ @ $provable$)
    in Have $(\Gamma, c)$ :: redirect $c$ $(last \cup earlier)$ $(seqs - \{(\Gamma, c)\})$ end
  else
    let val $zs$ = zones_of (length $last$) (map (descendants $seqs$) $last$)
          val $\mathscr{S}$ = map (fn $z \Rightarrow$ filter (fn $(\Gamma, \_) \Rightarrow \Gamma \subseteq earlier \cup z$) $seqs$) $zs$
          val $cases$ = map (fn $(a, ss) \Rightarrow (a,$ redirect $[a]$ $earlier$ $ss$)) (zip $last$ $\mathscr{S}$)
    in Cases $cases$ :: redirect (succedent_of_cases $cases$) $earlier$ $(seqs - \bigcup \mathscr{S})$ end
```

The code uses familiar ML functions, such as map, filter, and zip. It also relies on a descendants function that returns the descendants of the specified node in the AIG associated with *seqs*; its definition is omitted. Finally, the code depends on the following straightforward functions:

```
fun zones_of 0 _ = []
  | zones_of $n$ $(B :: Bs)$ = $(B - \bigcup Bs)$ :: zones_of $(n - 1)$ $(Bs$ @ $[B])$

fun succedent_of_inf (Have $(\_, c)$) = $c$
  | succedent_of_inf (Cases $cases$) = succedent_of_cases $cases$
and succedent_of_case $(a, [])$ = $[a]$
  | succedent_of_case $(\_, infs)$ = succedent_of_inf (last $infs$)
and succedent_of_cases $cases$ = $\bigcup$(map succedent_of_case $cases$)
```

Correctness. It is not hard to convince ourselves that the proof output by redirect is correct by inspecting the code. A Have (Γ, c) sequent is appended only if all the atoms in Γ have been proved (or assumed) already, and a case analysis on $a_1 \vee \cdots \vee a_m$ always follows a sequent with the succedent $a_1 \vee \cdots \vee a_m$. Whenever a sequent is output, it is removed from *seqs*. The function returns only if *seqs* is empty, at which point the conjecture must have been proved (except in the degenerate case where the negated conjecture does not participate in the refutation).

Termination is not quite as obvious. The recursion is well-founded, because the pair (length *seqs*, length *last*) becomes strictly smaller with respect to the lexicographic extension of $<$ on natural numbers for each of the three syntactic recursive calls.

- For the first recursive call, the list $seqs - \{(\Gamma, c)\}$ is strictly shorter than *seqs* since $(\Gamma, c) \in seqs$.
- The second call is performed for each branch of a case analysis; the *ss* argument is a (not necessarily strict) subset of the caller's *seqs*, and the list $[a]$ is strictly shorter than *last*, which has length 2 or more.

- For the third call, the key property is that at least one of the zones is nonempty, from which we obtain $\mathit{seqs} - \bigcup \mathscr{S} \subset \mathit{seqs}$. If all the zones were empty, each atom a_i would be the descendant of at least one atom $a_{i'}$ in the AIG (with $i' \neq i$), which is impossible because the AIG is acyclic.

As for run-time exceptions, the only worrisome construct is the hd call in $\mathsf{redirect}$'s second branch. We must convince ourselves that there exists at least one sequent $(\Gamma, c) \in \mathit{seqs}$ such that $\Gamma \subseteq \mathit{last} \cup \mathit{earlier}$. Intuitively, this is unsurprising, because *seqs* is initialized from a well-formed refutation graph: The nonexistence of such a sequent would indicate a gap or a cycle in the refutation graph. More precisely, if there exist untainted atoms $\notin \mathit{last} \cup \mathit{earlier}$, these can always be processed first; indeed, the preference for sequents with a single atom in their succedent ensures that they are processed before the first case analysis. Otherwise:

- If *last* is [] (representing $\top$) or an untainted atom, the contrapositive $a_1, \ldots, a_m \rhd b_1 \vee \cdots \vee b_n$ of the very last inference is applicable since it only depends on untainted atoms, all of which have already been proved.
- Otherwise, *last* is a tainted atom b. The refutation graph must contain an inference $a_1, \ldots, a_m, \overline{b_1}, \ldots, \overline{b_n} \rhd \overline{b}$, whose redirected inference is $a_1, \ldots, a_m, b \rhd b_1 \vee \cdots \vee b_n$. Since it only depends on b and untainted atoms, it is applicable.

Inlining. As a postprocessing step, we can abbreviate case analyses in which only one branch is nontrivial, transforming

$$\left[\begin{array}{c|c|c|r|c|c|c} & & & [c_i] & & & \\ & & & d_{11}, \ldots, d_{1k_1} \ \rhd\ e_1 & & & \\ & & & \vdots & & & \\ {[c_1]} & \cdots & [c_{i-1}] & d_{n1}, \ldots, d_{nk_n} \ \rhd\ e_n & [c_{i+1}] & \cdots & [c_m] \end{array}\right] \quad \text{into} \quad \begin{array}{r} \widetilde{d_{11}}, \ldots, \widetilde{d_{1k_1}} \ \rhd\ \widetilde{e_1} \\ \vdots \\ \widetilde{d_{n1}}, \ldots, \widetilde{d_{nk_n}} \ \rhd\ \widetilde{e_n} \end{array}$$

where the function $\widetilde{\ }$ is the identity except for the assumption c_i and the conclusions $e_1, \ldots, e_n$:

$$\widetilde{c_i} = c_1 \vee \cdots \vee c_m \qquad\qquad \widetilde{e_j} = c_1 \vee \cdots \vee c_{i-1} \vee e_j \vee c_{i+1} \vee \cdots \vee c_m$$

It is debatable whether such inlining is a good idea. The resulting proof has a simpler structure, with fewer nested proof blocks. However, these nested blocks can help make complex proof more intelligible. Moreover, the n-fold repetition of the disjuncts $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_m$ clutters the proof and can slow it down.

The above procedure can be generalized to arbitrary case analysis blocks. I am grateful to Konstantin Korovin for insights that lead me to realize this. We can rewrite an m-way case analysis

$$\left[\begin{array}{r|c|r} [c_1] & \cdots & [c_m] \\ \Gamma_{11} \ \rhd\ e_{11} & \cdots & \Gamma_{m1} \ \rhd\ e_{m1} \\ \vdots & & \vdots \\ \Gamma_{1k_1} \ \rhd\ e_{1k_1} & \cdots & \Gamma_{mk_m} \ \rhd\ e_{mk_m} \end{array}\right]$$

into a sequence of m case analyses:

$$\left[\begin{array}{c|c|c|r|c|c|c} & & & [c_i] & & & \\ & & & \Gamma_{i1} \rhd e_{i1} & & & \\ & & & \vdots & & & \\ [e_{1k_1}] & \cdots & [e_{(i-1)k_{i-1}}] & \Gamma_{ik_i} \rhd e_{ik_i} & [c_{i+1}] & \cdots & [c_m] \end{array}\right]$$

Each of these has only one nontrivial branch and can be inlined, yielding a branch-free proof. For the spaghetti proof of the previous section, this process yields

$$\begin{array}{l} \rhd\ 7 \lor 8 \\ \blacktriangleright\ 2 \lor 5 \lor 6 \lor 8 \\ \blacktriangleright\ 2 \lor 4 \lor 6 \lor 8 \\ \blacktriangleright\ 2 \lor 3 \lor 4 \lor 6 \\ \blacktriangleright\ 2 \lor 3 \lor 4 \\ \blacktriangleright\ 2 \lor 3 \\ \blacktriangleright\ 1 \lor 3 \\ \blacktriangleright\ 1 \\ \blacktriangleright\ 0 \end{array}$$

The example shows clearly that we rapidly obtain large disjunctions. In practice, each of the disjuncts would be an arbitrarily complex formula. Local definitions could be used to avoid repeating the formulas, but the loss of modularity is deplorable. Indeed, similar concerns about Hoare-style proof outlines for separation logic have lead to the development of ribbon proofs [30], whose parallel ribbons evoke the branches of a case analysis.

If branch-free proofs are nonetheless desired, they can be generated more directly by iteratively “rewriting” the atoms, following a suggestion by Korovin. For example, starting from the sequent $\rhd 7 \lor 8$, rewriting 7 would involve resolving $\rhd 7 \lor 8$ with $7 \rhd 2 \lor 5 \lor 6$, resulting in $2 \lor 5 \lor 6 \lor 8$. In general, rewriting a tainted atom b_j within a sequent $\Gamma \rhd b_1 \lor \cdots \lor b_n$ involves resolving that sequent with the redirected sequent that has b_j in its assumptions. To guarantee that the procedure is linear, it suffices to rewrite atoms only if all their ancestors in the AIG have already been rewritten, thereby preventing multiple rewrites of the same atom.

5 Conclusion

This paper presented an algorithm that transforms proofs by contradiction as returned by automatic theorem provers into direct proofs. It sometimes introduces case splits but avoids duplicating inferences in the different branches of the split by joining again as early as possible. The resulting proofs are direct Isar proofs that have some of the structure Isabelle users have come to expect. The described procedure is admittedly fairly straightforward; it would not be surprising if it were part of folklore or a special case of existing work.

While the output is designed for replaying proofs, it also has a pedagogical value: Unlike Isabelle’s automatic tactics, which are black boxes, the proofs delivered by Sledgehammer can be inspected and understood. The direct proof also forms a good basis for manual tuning. Further steps toward robust, intelligible Isar proofs, are described in a companion paper [25]. In future work, I am interested in transformations that increase proof readability and in the automatic discovery of concepts and lemmas, such as those available for Mizar proofs [16,17].

Acknowledgment. Tobias Nipkow, Lawrence Paulson, Geoff Sutcliffe, and Josef Urban motivated me to pursue this research. Andrei Popescu helped simplify the approach considerably. Charles Francis, sponsored by Google Summer of Code, developed a prototype. Sascha Böhme, Konstantin Korovin, Tobias Nipkow, Hans de Nivelle, Josef Urban, Tjark Weber, and two (still) anonymous reviewers produced a wealth of comments that helped enhance this text. I thank them all. The research was financed by the Deutsche Forschungsgemeinschaft (DFG) projects Quis Custodiet (grant Ni 491/11-1) and Hardening the Hammer (grant Ni 491/14-1).

References

[1] S. Autexier, C. Benzmüller, A. Fiedler, H. Horacek, and Q. B. Vo. Assertion-level proof representation with under-specification. *Electr. Notes Theor. Comput. Sci.*, 93:5–23, 2004.

[2] J. C. Blanchette. *Automatic Proofs and Refutations for Higher-Order Logic*. Ph.D. thesis, Dept. of Informatics, Technische Universität München, 2012.

[3] B. I. Dahn. Robbins algebras are Boolean: A revision of McCune's computer-generated solution of Robbins problem. *J. Algebra*, 208(2):526–532, 1998.

[4] M. Davis. Obvious logical inferences. In P. J. Hayes, editor, *IJCAI '81*, pages 530–531. William Kaufmann, 1981.

[5] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[6] G. Gentzen. Untersuchungen über das logische Schließen. *Math. Z.*, 39:176–210, 1935.

[7] X. Huang. Translating machine-generated resolution proofs into ND-proofs at the assertion level. In N. Y. Foo and R. Goebel, editors, *PRICAI '96*, volume 1114 of *LNCS*, pages 399–410. Springer, 1996.

[8] J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. Di Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.

[9] S. Jaśkowski. On the rules of suppositions in formal logic. *Studia Logica*, 1:5–32, 1934.

[10] D. E. Knuth, T. L. Larrabee, and P. M. Roberts. *Mathematical Writing*. Mathematical Association of America, 1989.

[11] W. McCune. Solution of the Robbins problem. *J. Autom. Reasoning*, 19(3):263–276, 1997.

[12] A. Meier. TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level (system description). In D. McAllester, editor, *CADE-17*, volume 1831 of *LNAI*, pages 460–464. Springer, 2000.

[13] D. Miller and A. Felty. An integration of resolution and natural deduction theorem proving. In *AAAI-86*, volume I: Science, pages 198–202. Morgan Kaufmann, 1986.

[14] T. Nipkow. Programming and proving in Isabelle. `http://isabelle.in.tum.de/dist/Isabelle2013/doc/prog-prove.pdf`, 2013.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[16] K. Pąk. The methods of improving and reorganizing natural deduction proofs. In *MathUI10*, 2010.

[17] K. Pąk. Methods of lemma extraction in natural deduction proofs. *J. Autom. Reasoning*, 50(2):217–228, 2013.

[18] L. C. Paulson. Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. In B. Konev, R. Schmidt, and S. Schulz, editors, *PAAR-2010*, pages 1–10, 2010.

[19] L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In G. Sutcliffe, E. Ternovska, and S. Schulz, editors, *IWIL-2010*, 2010.

[20] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.

[21] F. Pfenning. Analytic and non-analytic proofs. In R. E. Shostak, editor, *CADE-7*, volume 170 of *LNCS*, pages 393–413. Springer, 1984.

[22] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Comm.*, 15(2-3):91–110, 2002.

[23] P. Rudnicki. Obvious inferences. *J. Autom. Reasoning*, 3(4):383–393, 1987.

[24] S. Schulz. System description: E 0.81. In D. Basin and M. Rusinowitch, editors, *IJCAR 2004*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[25] S. J. Smolka and J. C. Blanchette. Robust, semi-intelligible Isabelle proofs from ATP proofs. In J. C. Blanchette and J. Urban, editors, *PxTP 2013*, 2013.

[26] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP data-exchange formats for automated theorem proving tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, volume 112 of *Frontiers in Artificial Intelligence and Applications*, pages 201–215. IOS Press, 2004.

[27] J. Urban, G. Sutcliffe, S. Trac, and Y. Puzis. Combining Mizar and TPTP semantic presentation and verification tools. In A. Grabowski and A. Naumowicz, editors, *Computer Reconstruction of the Body of Mathematics*, volume 18(31) of *Studies in Logic, Grammar and Rhetoric*, pages 121–136. University of Białystok, 2009.

[28] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, pages 1965–2013. Elsevier, 2001.

[29] M. Wenzel. Isabelle/Isar—A generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof—Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*. University of Białystok, 2007.

[30] J. Wickerson, M. Dodds, and M. J. Parkinson. Ribbon proofs for separation logic. In M. Felleisen and P. Gardner, editors, *ESOP 2013*, volume 7792 of *LNCS*, pages 189–208. Springer, 2013.

[31] J. Zimmer, A. Meier, G. Sutcliffe, and Y. Zhan. Integrated proof transformation services. In C. Benzmüller and W. Windsteiger, editors, *IJCAR WS 7*, 2004.

From Classical Extensional Higher-Order Tableau to Intuitionistic Intentional Natural Deduction

Chad E. Brown[1]
and Christine Rizkallah[2]

[1] Saarland University, Saarbrücken, Germany
`cebrown@ps.uni-saarland.de`
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
`crizkall@mpi-inf.mpg.de`

Abstract

We define a translation that maps higher-order formulas provable in a classical extensional setting to semantically equivalent formulas provable in an intuitionistic intensional setting. For the classical extensional higher-order proof system we define a Henkin-complete tableau calculus. For the intuitionistic intensional higher-order proof system we give a natural deduction calculus. We prove that tableau provability of a formula implies provability of a translated formula in the natural deduction calculus. Implicit in this proof is a method for translating classical extensional tableau refutations into intuitionistic intensional natural deduction proofs.

1 Introduction

We describe a way of translating a simply-typed higher-order formula s into a semantically equivalent formula s' such that if s is provable in a classical extensional higher-order tableau calculus, then s' is provable in an intuitionistic intentional higher-order natural deduction calculus. A potential application of such a translation is mapping refutations found by a classical extensional tableau-based automated theorem prover like Satallax [4] to corresponding proof terms in an intuitionistic intensional natural deduction based system like Coq [15, 3].

The problem of translating classical logic into intuitionistic logic has a long history. A result by Glivenko [11] states that if s is a propositional tautology, then $\neg\neg s$ is intuitionistically provable. This result does not hold for first-order formulas. Kuroda [14] showed a similar result if one translates by double negating the formula and adding double negations to the bodies of universal quantifiers. We recently proved that the Kuroda translation generalizes to give a translation from classical intentional higher-order logic to intuitionistic intentional higher order logic [5]. The generalized Kuroda translation does not suffice in the presence of functional extensionality, for an example see [5]. The work of Gandy [9] provides a method to translate from a higher-order logic with extensionality into a higher-order logic without extensionality. The translation we give in this paper uses ideas similar to those of Kuroda and Gandy to deal with both issues at once. The Gandy translation translates equality with the help of a binary relation and a predicate that are defined by mutual recursion. Our translation is simpler in that it translates equality with the help of a single binary relation that is defined inductively on types.

Many logical systems of quite different character are commonly referred to as *higher-order logics*. Some forms of higher-order logic allow classical reasoning while others only allow intuitionistic reasoning. For example, formulas such as $\forall p.p \vee \neg p$ and $\forall p.\neg\neg p \to p$ are provable if and only if the higher-order logic is classical. Likewise, some forms of higher-order logic allow extensional reasoning while others do not. Formulas such as $\forall fg.(\forall x.fx = gx) \to f = g$

and $\forall pq.(p \to q) \wedge (q \to p) \to p = q$ are provable if the higher-order logic is extensional and are not provable otherwise. The formula $(\lambda x.\neg\neg x) = (\lambda x.x)$ is only provable if the logic is both classical and extensional. The extensionality properties can be factored into propositional extensionality (i.e., boolean extensionality) and functional extensionality (which can itself be factored into extensionality properties η and ξ) [1]. In this paper we consider a tableau system with full extensionality and a natural deduction system with no extensionality.

Many automated and interactive theorem provers (e.g., Isabelle/HOL [16], LEO-II [2] and Satallax [4]) are based on classical extensional versions of higher-order logic. Automated and interactive theorem provers are used to create proofs, while the task of a proof checker is checking the correctness of proofs. In order to check a proof, the proof must be represented as an explicit object. A well-known way of representing proofs is via the Curry-Howard-de Bruijn correspondence [13, 8]: a natural deduction proof of P can be encoded as a λ-term of type P. It is easy to write a natural deduction system for higher-order logic for which one can obtain proof terms in an obvious way. However, this creates a natural deduction system which is intuitionistic (not classical) and intentional (not extensional).

One way to resolve this mismatch is to add classical extensional axioms to the natural deduction system. Currently, Satallax produces Coq proof terms under the assumption that one has added such axioms to Coq [20]. Using the ideas in this paper, one can avoid assuming such axioms in Coq and still produce Coq proof terms from Satallax refutations (unless Satallax makes use of a choice operator).

The basic definitions and lemmas used in the translation have all been formalized and proven in Coq, as described in [18, 19]. However, there is currently no implementation of the translation as a whole. In particular, Satallax has not yet been extended to produce Coq proof terms using these definitions and lemmas.

Since the translation changes the formula, a Coq user could not typically call a classical extensional theorem prover like Satallax to request a proof term for an arbitrary formula in Coq. On the other hand, one could use the translation to automatically create a Coq development from a development in a classical extensional simple type theory.

The rest of the paper is organized as follows. In Section 2 we give a brief overview of simply typed λ-calculus and in Section 3 we introduce a tableau calculus $\mathcal{T}$. We then present the targeted natural deduction calculus $\mathcal{N}$ in Section 4. In Section 5 we present our translation. We prove that it maps $\mathcal{T}$-refutable branches to $\mathcal{N}$-refutable contexts in Section 6. We conclude in Section 7 and provide suggestions for future work.

More details about the translation in this paper and other translations can be found in the Master's thesis of one of the authors [18].

2 Simply Typed Lambda Calculus

2.1 Syntax

We describe simply typed λ-calculus in the style of Church [7]. The set of *types* $\mathbb{T}$ is given inductively: o and ι are types and $\sigma\tau$ is a type whenever σ and τ are types. The types o (the type of propositions) and ι (the type of individuals) are called *base types*. Types of the form $\sigma\tau$ are called *function types*. Often the function type $\sigma\tau$ is written as $\sigma \to \tau$, but we use the shorter notation since there are only base types and function types. We use σ and τ to range over types.

For each type σ, let $\mathcal{N}_\sigma$ be an infinite set of *names* of type σ. Some of the names are *logical constants*:

- $\top$ and $\bot$ are (distinct) logical constants in $\mathcal{N}_o$.
- $\wedge$, $\vee$, and $\to$ are (distinct) logical constants in $\mathcal{N}_{ooo}$.
- For each σ, $=^\sigma$ is a logical constant in $\mathcal{N}_{\sigma\sigma o}$.
- For each σ, $\forall^\sigma$ and $\exists^\sigma$ are (distinct) logical constants in $\mathcal{N}_{(\sigma o)o}$.

The remaining names are *variables* . Let $\mathcal{C}_\sigma$ be the set of logical constants of type σ and $\mathcal{V}_\sigma$ be the (infinite) set of all variables of type σ. Let $\mathcal{N} = \bigcup_\sigma \mathcal{N}_\sigma$, $\mathcal{C} = \bigcup_\sigma \mathcal{C}_\sigma$ and $\mathcal{V} = \bigcup_\sigma \mathcal{V}_\sigma$.

For any $\mathcal{C}' \subseteq \mathcal{C}$ we define a family of sets of terms $\Lambda_\sigma^{\mathcal{C}'}$ for each type σ by induction.

- For every $x \in \mathcal{V}_\sigma$, $x \in \Lambda_\sigma^{\mathcal{C}'}$.
- For every $c \in \mathcal{C}_\sigma \cap \mathcal{C}'$, $c \in \Lambda_\sigma^{\mathcal{C}'}$.
- For every $x \in \mathcal{V}_\sigma$ and $s \in \Lambda_\tau^{\mathcal{C}'}$, $(\lambda x.s) \in \Lambda_{\sigma\tau}^{\mathcal{C}'}$.
- For every $s \in \Lambda_{\sigma\tau}^{\mathcal{C}'}$ and $t \in \Lambda_\sigma^{\mathcal{C}'}$, $(st) \in \Lambda_\tau^{\mathcal{C}'}$.

We defined $\Lambda_\sigma^{\mathcal{C}'}$ relative to a set of logical constants. There are two particular sets of logical constants of interest in this paper: the full set $\mathcal{C}$ of logical constants and the set

$$\mathcal{C}^- := \{\to, \forall^\sigma | \sigma \text{ is a type}\}.$$

To simplify notation, we define Λ_σ to be $\Lambda_\sigma^{\mathcal{C}}$ and Λ_σ^- to be $\Lambda_\sigma^{\mathcal{C}^-}$. Note that $\Lambda_\sigma^{\mathcal{C}'} \subseteq \Lambda_\sigma$ for any $\mathcal{C}' \subseteq \mathcal{C}$. An element of Λ_σ is called a *term of type* σ. A *term* is an element of $\bigcup_\sigma \Lambda_\sigma$. Terms of the form $(\lambda x.s)$ are called λ-*abstractions*. Terms of the form (st) are called *applications*. A *formula* is a term of type o.

We write $\neg s$ for $((\to s)\bot)$. We write stu for $(st)u$, except that $\neg st$ means $\neg(st)$. We use the infix notations $s \wedge t$, $s \vee t$, $s \to t$ and $s =^\sigma t$ as shorthand for $\wedge st$, $\vee st$, $\to st$ and $=^\sigma st$, respectively. We also write $s \neq^\sigma t$ for $\neg(s =^\sigma t)$. Using infix notation note that $\neg s$ is the same term as $s \to \bot$. We sometimes write $\forall x : \sigma.s$ and $\exists x : \sigma.s$ for $\forall^\sigma(\lambda x.s)$ and $\exists^\sigma(\lambda x.s)$, respectively. We may also omit the type entirely from a quantified formula, equation or disequation and write $\forall x.s$, $\exists x.s$, $s = t$ or $s \neq t$ when the types are clear from the context.

If s is a term, $x \in \mathcal{N}_\sigma$ and t is a term of type σ, then $s[x := t]$ is defined to be the result of substituting t for x in s via a capture-avoiding substitution. A *simultaneous substitution* θ substitutes several variables simultaneously. We use the notation $\theta, [x := t]$ to mean the simultaneous substitution that agrees with θ on all variables except (possibly) x which is mapped to t. A term of the form $(\lambda x.s)t$ is called a β-*redex* with β-*reduct* $s[x := t]$. We say s β-*reduces to* t (and write $s \to_\beta t$) if a subterm of s is a β-redex such that t is the result of replacing this subterm by its β-reduct. We define $s \sim_\beta t$ to be the least equivalence relation containing $\to_\beta$. When $s \sim_\beta t$ holds, we say s and t are β-*equivalent*. A term is β-*normal* if it has no β-redexes. It is well-known that β-reduction is confluent and terminates on simply typed terms. Hence for every $s \in \Lambda_\sigma$ there is a β-normal form of s which is unique (up to names of bound variables). We write $\lceil s \rceil^\beta$ to denote the β-normal form of s.

The set of *free variables* of a term, written FV, is defined as follows.

$$\begin{array}{lll} FV(x) & := & \{x\} \\ FV(s\,t) & := & FV(s) \cup FV(t) \\ FV(\lambda x.s) & := & FV(s) - \{x\} \end{array}$$

A term s is *ground* if $FV(s) = \emptyset$. The set of *free variables* of a set of terms B is defined as $FV(B) := \bigcup_{s \in B} FV(s)$.

2.2 Semantics

Henkin proved completeness of a form of Church's simple theory of types [7] relative to a semantics now known as Henkin semantics [12]. We briefly describe Henkin semantics. The interested reader may find more details of a similar presentation in [6].

A *frame* is a function $\mathcal{D}$ defined on $\mathbb{T}$ such that $\mathcal{D}(o) = \{0,1\}$, $\forall\sigma \in \mathbb{T} : \mathcal{D}(\sigma) \neq \emptyset$ and $\forall\sigma,\tau \in \mathbb{T} : \mathcal{D}(\sigma\tau) \subseteq \{f \mid f : \mathcal{D}(\sigma) \to \mathcal{D}(\tau)\}$. An *assignment into a frame* $\mathcal{D}$ is a function $\mathcal{I}$ defined on $\mathbb{T} \cup \mathcal{V}$ such that $\mathcal{I}(x) \in \mathcal{D}(\sigma)$ for all types σ and variables $x : \sigma$. Let $\mathcal{I}$ be an assignment into a frame $\mathcal{D}$, $x : \sigma$ be a variable and $a \in \mathcal{D}(\sigma)$. We write $\mathcal{I}^x_a$ to denote the assignment into $\mathcal{D}$ that agrees everywhere with $\mathcal{I}$ except possibly on x where it yields a. We define a partial evaluation function $\hat{}$ that maps assignments $\mathcal{I}$ and terms $s \in \Lambda_\sigma$ possibly to values $\hat{\mathcal{I}}(s)$ in $\mathcal{D}(\sigma)$ as follows:

1. $\hat{\mathcal{I}}(x) := \mathcal{I}(x)$
2. $\hat{\mathcal{I}}(c) := f$ if $c : \sigma$, $f \in \mathcal{D}(\sigma)$ and f has the usual classical meaning of c
3. $\hat{\mathcal{I}}(s\,t) := \hat{\mathcal{I}}(s)(\hat{\mathcal{I}}(t))$
4. $\hat{\mathcal{I}}(\lambda x.s) := f$ if $\lambda x.s : \sigma\tau$, $f \in \mathcal{D}(\sigma\tau)$ and $\forall a \in \mathcal{D}(\sigma) : \widehat{\mathcal{I}^x_a}(s) = f a$

Note that $\hat{\mathcal{I}}$ is a partial function from typed terms into the frame. An *interpretation* is a pair $(\mathcal{D},\mathcal{I})$ where $\mathcal{D}$ is a frame, $\mathcal{I}$ is an assignment into $\mathcal{D}$ and $\hat{\mathcal{I}}$ is a total function, i.e., $\hat{\mathcal{I}}s$ is defined for every $s \in \Lambda_\sigma$. We write *Interp* for the set of all interpretations.

A *formula* is a term of type o. We say an interpretation $(\mathcal{D},\mathcal{I})$ *satisfies* a formula s if $\hat{\mathcal{I}}(s) = 1$. A set A of formulas is *satisfiable* if there is an interpretation $(\mathcal{D},\mathcal{I})$ simultaneously satisfying all the formulas in A. Otherwise, we say A is *unsatisfiable*. We say two terms $s,t \in \Lambda_\sigma$ are *semantically equivalent* (written $s \approx t$) if $\hat{\mathcal{I}}s = \hat{\mathcal{I}}t$ for all interpretations $(\mathcal{D},\mathcal{I})$.

3 Tableau Calculus $\mathcal{T}$

We briefly describe a tableau calculus for classical extensional higher-order logic which is complete relative to Henkin semantics. A similar tableau calculus is presented and proven complete in [6]. The calculus in [6] only uses the logical constants $=_\sigma$ and $\neg$ while here we include many more logical constants. Also, we include rules such as Cut and DeMorgan which are not needed for completeness. We include these rules because they are sound relative to Henkin semantics and the translation we give later is able to handle the extra rules.

A *branch* is a set of β-normal formulas. A *step* is an $n+1$-tuple $\langle A_1,\ldots,A_n,A\rangle$ of branches with $n \geq 0$. Given a set of steps T, one can inductively define the set of branches which are T*-refutable* as follows: If $\langle A_1,\ldots,A_n,A\rangle \in T$ and A_i is T-refutable for $i \in \{1,\ldots,n\}$, then A is T-refutable.

A *rule* is a set of steps. The rules are presented in the form

$$\mathit{RuleName}\,\frac{C}{B_1 \mid \cdots \mid B_n}$$

to indicate the set of steps of the form $\langle A_1,\ldots,A_n,A\rangle$ where $C \subseteq A$ and $A_i = A \cup B_i$ for each $i \in \{1,\ldots,n\}$. There are also sometimes side conditions on the branch A. For example if we say a variable y must be *fresh* in a rule, this means that for the step $\langle A_1,\ldots,A_n,A\rangle$ to be in the rule there is the additional requirement that $y \notin FV(A)$. In most cases C is a singleton set $\{s\}$ and in the remaining cases C is either empty (e.g, Cut) or contains two formulas (e.g., Mat).

Definition 3.1 (Tableau Calculus $\mathcal{T}$)**.** We define the *tableau calculus* $\mathcal{T}$ as the union of the rules in Figure 1. This also defines the corresponding notion of $\mathcal{T}$-refutability. We say a formula s is *$\mathcal{T}$-refutable* if the branch $\{\lceil s\rceil^\beta\}$ is $\mathcal{T}$-refutable. We say a formula s is *$\mathcal{T}$-provable* if the formula $\neg\lceil s\rceil^\beta$ is $\mathcal{T}$-refutable.

$$\textsf{Closed}\bot\frac{\bot}{} \qquad \textsf{Closed}\neg\top\frac{\neg\top}{} \qquad \textsf{Closed}\frac{s,\neg s}{} \qquad \textsf{Closed}\neq\frac{s\neq s}{}$$

$$\textsf{ClosedSym}\frac{(s=t),\ (t\neq s)}{} \qquad \textsf{Cut}\frac{}{s \mid \neg s} \qquad \textsf{Dneg}\frac{\neg\neg s}{s} \qquad \textsf{And}\frac{s\wedge t}{s,t} \qquad \textsf{Or}\frac{s\vee t}{s \mid t}$$

$$\textsf{Imp}\frac{s\to t}{\neg s \mid t} \qquad \textsf{NegAnd}\frac{\neg(s\wedge t)}{\neg s \mid \neg t} \qquad \textsf{NegOr}\frac{\neg(s\vee t)}{\neg s,\neg t} \qquad \textsf{NegImp}\frac{\neg(s\to t)}{s,\neg t}$$

$$\textsf{Forall}\frac{\forall s}{\lceil s\ t\rceil^\beta} \qquad \textsf{DeMorgan}\forall\frac{\neg\forall s}{\exists x.\neg\lceil s\ x\rceil^\beta}x\notin FV(s) \qquad \textsf{Exists}\frac{\exists s}{\lceil s\ y\rceil^\beta}y \text{ is fresh}$$

$$\textsf{DeMorgan}\exists\frac{\neg\exists s}{\forall x.\neg\lceil s\ x\rceil^\beta}x\notin FV(s) \qquad \textsf{Bool}=\frac{s=^o t}{s,t \mid \neg s,\neg t} \qquad \textsf{BoolExt}\frac{s\neq^o t}{s,\neg t \mid t,\neg s}$$

$$\textsf{Func}=\frac{s_1=^{\sigma\tau} s_2}{\lceil s_1\ t=^\tau s_2\ t\rceil^\beta} \qquad \textsf{FuncExt}\frac{s\neq^{\sigma\tau} t}{\lceil s\,x\neq^\tau t\,x\rceil^\beta}x \text{ is fresh} \qquad \textsf{Mat}\frac{x\,s_1\ldots s_n,\neg x\,t_1\ldots t_n}{s_1\neq t_1 \mid \ldots \mid s_n\neq t_n}$$

$$\textsf{Dec}\frac{x\,s_1\ldots s_n\neq^\iota x\,t_1\ldots t_n}{s_1\neq t_1 \mid \ldots \mid s_n\neq t_n} \qquad \textsf{Con}\frac{s_1=^\iota t_1,\ s_2\neq^\iota t_2}{s_1\neq s_2,\ t_1\neq s_2 \mid s_1\neq t_2,\ t_1\neq t_2}$$

Figure 1: Tableau rules used to define the tableau calculus $\mathcal{T}$

Many variations of the tableau rules are possible. For example, instead of the rule

$$\textsf{DeMorgan}\exists\frac{\neg\exists s}{\forall x.\neg\lceil s\ x\rceil^\beta}x\notin FV(s)$$

we could have an alternative rule

$$\textsf{Neg}\exists\frac{\neg\exists s}{\neg\lceil s\ x\rceil^\beta}x \text{ is fresh.}$$

Note that if we use Neg∃, then x must be fresh rather than the weaker requirement $x\notin FV(s)$ in DeMorgan∃. By using the DeMorgan∃ version, there is one fewer rule for which the freshness condition needs to be taken care of later. One could similarly modify the rule FuncExt so that the only rule with a freshness condition would be Exists. In this paper the form of the tableau rules are chosen to match those considered in [18].

We briefly consider two examples of $\mathcal{T}$-provable formulas.

Example 3.2. Let p be a variable of type o. We show the formula $p\neq\neg p$ is $\mathcal{T}$-provable, i.e., the branch $A_0 := \{p\neq\neg p\}$ is $\mathcal{T}$-refutable. The branch A_0 is $\mathcal{T}$-refutable because of the NegOr rule and the fact that $A_1 := A_0\cup\{p,\neg\neg\neg p\}$ and $A_2 := A_0\cup\{\neg p,\neg\neg p\}$ are $\mathcal{T}$-refutable. We know A_2 is $\mathcal{T}$-refutable using the Closed rule. We know A_1 is $\mathcal{T}$-refutable using the Dneg and

Closed rules. The reason we call this a *tableau refutation* is that one can display the refutation as a picture we refer to as a *tableau*. For this example the following is the corresponding tableau:

$$\begin{array}{c} p \neq^o \neg p \\ \begin{array}{c|c} p & \neg p \\ \neg\neg\neg p & \neg\neg p \\ \neg p & \end{array} \end{array}$$

Example 3.3. Let p be a variable of type o. The formula $(\lambda p.\neg\neg p) =^{oo} (\lambda p.p)$ is $\mathcal{T}$-provable. In this case we simply show the tableau and note that the steps are justified by the FuncExt, BoolExt, Dneg and Closed rules.

$$\begin{array}{c} (\lambda p.\neg\neg p) \neq (\lambda p.p) \\ (\neg\neg p) \neq p \\ \begin{array}{c|c} \neg\neg p & \neg\neg\neg p \\ \neg p & p \\ & \neg p \end{array} \end{array}$$

4 Natural Deduction Calculus $\mathcal{N}$

We now present a natural deduction calculus for formulas in Λ_o^-. That is, we only consider formulas that use the logical constants $\rightarrow$ and $\forall^\sigma$. Such calculi were introduced by Gentzen in 1935 [10] and studied further by Prawitz [17].

A *context* Γ is a finite subset of Λ_o^-. $\Gamma \vdash_{\mathcal{N}} s$ holds when derivable using the rules in Figure 2.

Note that if for some context Γ and some formula s we are given a derivation of $\Gamma \vdash_{\mathcal{N}} s$ that uses the wk rule, we can construct a derivation of $\Gamma \vdash_{\mathcal{N}} s$ that does not use the wk rule. This follows from how the hy rule is stated. We only add the wk rule for convenience.

$$hy\ \frac{t \in \Gamma}{\Gamma \vdash_{\mathcal{N}} t} \qquad \beta\ \frac{\Gamma \vdash_{\mathcal{N}} s}{\Gamma \vdash_{\mathcal{N}} t}\ s \sim_\beta t \qquad wk\ \frac{\Gamma' \vdash_{\mathcal{N}} t}{\Gamma \vdash_{\mathcal{N}} t}\ \Gamma' \subseteq \Gamma$$

$$\forall I\ \frac{\Gamma \vdash_{\mathcal{N}} t}{\Gamma \vdash_{\mathcal{N}} \forall x.t}\ x \notin FV(\Gamma) \qquad \rightarrow I\ \frac{\Gamma, s \vdash_{\mathcal{N}} t}{\Gamma \vdash_{\mathcal{N}} s \rightarrow t}$$

$$\forall E\ \frac{\Gamma \vdash_{\mathcal{N}} \forall x.s}{\Gamma \vdash_{\mathcal{N}} s[x := t]} \qquad \rightarrow E\ \frac{\Gamma \vdash_{\mathcal{N}} s \rightarrow t \quad \Gamma \vdash_{\mathcal{N}} s}{\Gamma \vdash_{\mathcal{N}} t}$$

Figure 2: Rules in our ND calculus $\mathcal{N}$

We write $\vdash_{\mathcal{N}} s$ for $\emptyset \vdash_{\mathcal{N}} s$. We say a formula $s \in \Lambda_o^-$ is $\mathcal{N}$*-provable* if $\vdash_{\mathcal{N}} s$. We say a context Γ is $\mathcal{N}$*-refutable* if $\Gamma \vdash_{\mathcal{N}} \forall^o p.p$. Likewise, a formula $s \in \Lambda_o^-$ is $\mathcal{N}$*-refutable* if the context $\{s\}$ is $\mathcal{N}$-refutable.

Example 4.1. Let x, y, p and q be variables with $x, y \in \mathcal{V}_\iota$ and $p, q \in \mathcal{V}_{\iota o}$. We use the following diagram to show that the formula $(\forall p.px \rightarrow py) \rightarrow qy \rightarrow qx$ is $\mathcal{N}$-provable.

$$\rightarrow I\ \dfrac{\rightarrow E\ \dfrac{\beta\ \dfrac{\forall E\ \dfrac{hy\ \dfrac{}{\{\forall p.px \rightarrow py\} \vdash_{\mathcal{N}} \forall p.px \rightarrow py}}{\{\forall p.px \rightarrow py\} \vdash_{\mathcal{N}} (\lambda y.qy \rightarrow qx)x \rightarrow (\lambda y.qy \rightarrow qx)y}}{\{\forall p.px \rightarrow py\} \vdash_{\mathcal{N}} (qx \rightarrow qx) \rightarrow (qy \rightarrow qx)} \qquad wk\ \dfrac{\rightarrow I\ \dfrac{hy\ \dfrac{}{\{qx\} \vdash_{\mathcal{N}} qx}}{\vdash_{\mathcal{N}} qx \rightarrow qx}}{\{\forall p.px \rightarrow py\} \vdash_{\mathcal{N}} qx \rightarrow qx}}{\{\forall p.px \rightarrow py\} \vdash_{\mathcal{N}} qy \rightarrow qx}}{\vdash_{\mathcal{N}} (\forall p.px \rightarrow py) \rightarrow qy \rightarrow qx}$$

5 Translating Terms, Formulas and Branches

In this section we introduce a meaning preserving translation that maps tableau refutable formulas in Λ_o to $\mathcal{N}$-refutable formulas in Λ_o^-. Since a tableau calculus operates on branches instead of formulas, we will also need to define a branch translation Ψ^* that maps branches to contexts.

We begin by considering the most important issue: the translation of logical constants. We will associate with each logical constant $c \in \mathcal{C}_\sigma$ a term $\dot{c} \in \Lambda_\sigma^-$. For the propositional connectives we can use terms which are sometimes called the Russell-Prawitz definitions [17].

$$\begin{array}{rcl} \dot{\rightarrow} & := & \lambda x.\lambda y.x \rightarrow y \\ \dot{\bot} & := & \forall p : o.p \\ \dot{\top} & := & \forall p : o.p \rightarrow p \\ \dot{\neg} & := & \lambda x.x \rightarrow \forall p : o.p \\ \dot{\wedge} & := & \lambda x.\lambda y.\forall p : o.(x \rightarrow y \rightarrow p) \rightarrow p \\ \dot{\vee} & := & \lambda x.\lambda y.\forall p : o.(x \rightarrow p) \rightarrow (y \rightarrow p) \rightarrow p \end{array}$$

We also define $\dot{\equiv}$ to be $\lambda xy : o.(x \rightarrow y)\dot{\wedge}(y \rightarrow x)$, even though we do not have a logical constant $\equiv$, since it will be useful below.

We will define $\dot{=}^\sigma$ to be a term $\mathbf{R}^\sigma$ of type $\sigma\sigma o$. This term $\mathbf{R}^\sigma$ will also be used in the definitions of $\dot{\forall}^\sigma$ and $\dot{\exists}^\sigma$. Since $\dot{=}^\sigma$ will be defined as $\mathbf{R}^\sigma$, the meaning of $\mathbf{R}^\sigma$ in every (classical extensional) interpretation must be equality. A simple way to satisfy this constraint is to define $\mathbf{R}^\sigma$ to be Leibniz equality, i.e., $\lambda xy.\forall q : \sigma o.qx \rightarrow qy$, at each type σ. For each $\mathcal{T}$-provable formula the translation should be $\mathcal{N}$-provable. If $\mathbf{R}^{\iota\iota}$ and $\mathbf{R}^\iota$ were both Leibniz equality, then even though $\forall fg : \iota\iota.(\forall x : \iota.fx = gx) \rightarrow f = g$ is clearly $\mathcal{T}$-provable its translation would not be $\mathcal{N}$-provable.

This suggests that we should not define $\mathbf{R}^\sigma$ to be Leibniz equality for every type σ. Instead we will define $\mathbf{R}^\sigma$ to be Leibniz equality only when σ is the base type ι. We will define $\mathbf{R}^o$ to be logical equivalence (as expressed by $\dot{\equiv}$) and on function types we will use functional equivalence modified by a double negation.

Definition 5.1. For every type σ we define inductively a term $\mathbf{R}^\sigma$ in $\Lambda_{\sigma\sigma o}^-$ as follows:

$$\begin{array}{l} \mathbf{R}^o = \lambda x\, y.(x \rightarrow y)\dot{\wedge}(y \rightarrow x) \\ \mathbf{R}^\iota = \lambda x\, y.\forall q : \iota o.q\, x \rightarrow q\, y \\ \mathbf{R}^{\sigma\rightarrow\tau} = \lambda f\, g.\forall x\, y : \sigma.\mathbf{R}^\sigma\, x\, y \rightarrow \dot{\neg}\dot{\neg}\mathbf{R}^\tau(f\, x)(g\, y) \end{array}$$

The term $\mathbf{R}^\sigma$ corresponds to a binary relation on type σ. We sometimes speak of $\mathbf{R}^\sigma$ as a relation rather than as a term.

It will turn out that we cannot generally prove (in $\mathcal{N}$) that $\mathbf{R}^\sigma$ is reflexive. As a consequence, we must restrict the binders in the definitions of $\dot{\forall}^\sigma$ and $\dot{\exists}^\sigma$ to x satisfying $\mathbf{R}^\sigma\ x\ x$. In terms of Henkin semantics, this will not make a difference since we will prove that $\hat{\mathcal{I}}(\mathbf{R}^\sigma)$ is equality on $\mathcal{D}(\sigma)$ in every interpretation $(\mathcal{D}, \mathcal{I})$. When considering provability of formulas in $\mathcal{N}$, the restriction to x satisfying $\mathbf{R}^\sigma\ x\ x$ will be important.

Now we define $\dot{=}^\sigma$, $\dot{\forall}^\sigma$, and $\dot{\exists}^\sigma$ as follows:

$$\begin{array}{rcl} \dot{=}^\sigma & := & \mathbf{R}^\sigma \\ \dot{\forall}^\sigma & := & \lambda f.\forall^\sigma x.\mathbf{R}^\sigma xx \rightarrow \dot{\neg}\dot{\neg}fx \\ \dot{\exists}^\sigma & := & \lambda f.\forall^o p.(\forall^\sigma x.\mathbf{R}^\sigma xx \rightarrow fx \rightarrow p) \rightarrow p \end{array}$$

Definition 5.2. We define our translation $\Psi : \Lambda_\sigma \to \Lambda_\sigma^-$ by recursion as follows.

$$\begin{array}{rcll} \Psi x & := & x & \text{for variables } x \\ \Psi st & := & (\Psi s)(\Psi t) & \\ \Psi \lambda x.s & := & \lambda x.\Psi s & \\ \Psi c & := & \dot{c} & \text{for constants } c \end{array}$$

We call Ψ *compositional* because it respects application and λ-abstraction. As a simple consequence of compositionality, we know that Ψ preserves β-equivalence.

Lemma 5.3. If s and t are β-equivalent, then Ψs and Ψt are also β-equivalent.

We now prove $\Psi s \approx s$ for all $s \in \Lambda_\sigma$. We first prove that $\mathbf{R}^\sigma$ behaves like equality.

Lemma 5.4. $\forall \sigma \in \mathbb{T} : \forall (\mathcal{D}, \mathcal{I}) \in \mathit{Interp} : \forall a, b \in \mathcal{D}(\sigma) : (\hat{\mathcal{I}}(\mathbf{R}^\sigma)\; a\, b = 1) \iff a = b$

Proof. We prove this lemma by induction on types. Let $(\mathcal{D}, \mathcal{I})$ be an arbitrary interpretation and let a and b be arbitrary elements in $\mathcal{D}(\sigma)$.

- Case $\sigma = o$:
  It is easy to check that $\widehat{\mathcal{I}^{xy}_{ab}}((x \to y)\dot{\wedge}(y \to x)) = 1 \iff a = b$.
- Case $\sigma = \iota$:
  We know $\hat{\mathcal{I}}(\mathbf{R}^\iota)\; a\, b = 1 \iff a = b$ since $\mathbf{R}^\iota$ is Leibniz equality.
- Case $\sigma = \sigma_1\sigma_2$:
  We want to show $\hat{\mathcal{I}}(\mathbf{R}^{\sigma_1\sigma_2})\; a\, b = 1 \iff a = b$.
  - Assume $\hat{\mathcal{I}}(\mathbf{R}^{\sigma_1\sigma_2})\; a\, b = 1$. We need to show $a = b$. Let $c \in \mathcal{D}(\sigma_1)$ be given. We prove $a(c) = b(c)$ as follows.

$$\begin{array}{ll} \hat{\mathcal{I}}(\mathbf{R}^{\sigma_1\sigma_2})\; a\, b = 1 & \\ \iff \widehat{\mathcal{I}^{fg}_{ab}}(\forall x\, y.\mathbf{R}^{\sigma_1}\; x\, y \to \dot{\neg}\dot{\neg}\mathbf{R}^{\sigma_2}(f\; x)(g\; y)) = 1 & \\ \implies \widehat{\mathcal{I}^{fgxy}_{abcc}}(\mathbf{R}^{\sigma_1}\; x\, y \to \dot{\neg}\dot{\neg}\mathbf{R}^{\sigma_2}(f\; x)(g\; y)) = 1 & \\ \iff (\hat{\mathcal{I}}(\mathbf{R}^{\sigma_1})\; c\, c = 1 \implies \hat{\mathcal{I}}(\mathbf{R}^{\sigma_2})(a(c))(b(c)) = 1) & \\ \iff (c = c \implies a(c) = b(c)) & \text{(IH)} \\ \iff a(c) = b(c) & \end{array}$$

  - Assume $a = b$. Let $c, d \in \mathcal{D}(\sigma_1)$ be such that $\hat{\mathcal{I}}(\mathbf{R}^{\sigma_1})\; c\, d = 1$. We know $c = d$ by the inductive hypothesis and so $ac = bd$. By the inductive hypothesis we have $\hat{\mathcal{I}}(\mathbf{R}^{\sigma_2})\; (ac)\; (bd) = 1$. Hence $\hat{\mathcal{I}}(\mathbf{R}^{\sigma_1\sigma_2})\; a\, b = 1$.

□

Lemma 5.5. For every interpretation $(\mathcal{D}, \mathcal{I})$ and every a in $\mathcal{D}(\sigma)$ we have $\mathcal{I}(\mathbf{R}^\sigma)\; a\, a = 1$.

Proof. Follows directly from Lemma 5.4. □

Lemma 5.6. For every $c \in \mathcal{C}_\sigma$, $\dot{c} \approx c$.

Proof. Let $(\mathcal{D},\mathcal{I})$ be an interpretation. In order to prove $\hat{\mathcal{I}}(\dot{c}) = \hat{\mathcal{I}}(c)$ it is enough to prove $\hat{\mathcal{I}}(\dot{c})$ has the usual classical meaning of c in $\mathcal{I}$ since at most one element of $\mathcal{D}(\sigma)$ can have this property. It is easy to check this for $\hat{\mathcal{I}}(\dot{c})$ when $c \in \{\to, \bot, \top, \neg, \wedge, \vee\}$. We know $\hat{\mathcal{I}}(\dot{=}^\tau)$ behaves like equality by Lemma 5.4. It remains to prove $\hat{\mathcal{I}}(\dot{\forall}^\tau)$ and $\hat{\mathcal{I}}(\dot{\exists}^\tau)$ behave like universal and existential quantification. Let $q \in \mathcal{D}(\tau o)$ be given. We use Lemma 5.4 to obtain

$$\begin{aligned}
&\hat{\mathcal{I}}(\dot{\forall}^\tau)\, q = 1\\
\iff &\widehat{\mathcal{I}_q^f}(\forall^\tau x.(\mathbf{R}^\tau xx) \to \dot{\neg}\dot{\neg} fx) = 1\\
\iff &\widehat{\mathcal{I}_q^f}(\forall^\tau x.fx) = 1\\
\iff &qa = 1 \text{ for every } a \in \mathcal{D}(\tau)
\end{aligned}$$

and

$$\begin{aligned}
&\hat{\mathcal{I}}(\dot{\exists}^\tau)\, q = 1\\
\iff &\widehat{\mathcal{I}_q^f}(\forall^o p.(\forall^\sigma x.(\mathbf{R}^\sigma xx) \to fx \to p) \to p) = 1\\
\iff &\widehat{\mathcal{I}_q^f}(\forall^o p.(\forall^\sigma x.fx \to p) \to p) = 1\\
\iff &qa = 1 \text{ for some } a \in \mathcal{D}(\tau).
\end{aligned}$$

□

Theorem 5.7. For every $s \in \Lambda_\sigma$, $\Psi s \approx s$.

Proof. We argue by induction on s. If s is a variable, then the result is clear. If s is a logical constant, then the result follows by Lemma 5.6. Suppose s is tu. Let $(\mathcal{D},\mathcal{I})$ be an interpretation. By inductive hypothesis $\hat{\mathcal{I}}(\Psi t) = \hat{\mathcal{I}}(t)$ and $\hat{\mathcal{I}}(\Psi u) = \hat{\mathcal{I}}(u)$. Hence $\hat{\mathcal{I}}(\Psi(tu)) = \hat{\mathcal{I}}(tu)$.

Finally, suppose s is $\lambda x.t$ of type $\sigma_1\sigma_2$ and let $(\mathcal{D},\mathcal{I})$ be an interpretation. Let $a \in \mathcal{D}(\sigma_1)$ be given. By the inductive hypothesis $\widehat{\mathcal{I}_a^x}(\Psi t) = \widehat{\mathcal{I}_a^x}(t)$. Generalizing over a, we conclude $\hat{\mathcal{I}}(\Psi t) = \hat{\mathcal{I}}(t)$. □

Using Lemma 5.4 and Theorem 5.7 we easily obtain the following corollary.

Corollary 5.8. Let s be a formula such that $FV(s) = \{x_1, \ldots, x_n\}$. We know

$$s \approx (x_1 \dot{=} x_1 \to \cdots \to x_n \dot{=} x_n \to \dot{\neg}\dot{\neg}\Psi s)$$

We now consider which properties of $\mathbf{R}$ are provable in $\mathcal{N}$. In particular, $\mathbf{R}^\sigma$ is provably symmetric and transitive, i.e., a PER (partial equivalence relation). As we previously mentioned, we cannot generally prove $\mathbf{R}^\sigma$ is reflexive in $\mathcal{N}$ since $\mathcal{N}$ lacks extensionality, but we can prove that $\mathbf{R}^\sigma$ is reflexive when σ is o or ι. Proofs of the next two lemmas are straightforward and can be found as Coq proofs in [18, 19].

Lemma 5.9. For each type σ we have $\vdash_{\mathcal{N}} \forall^\sigma xyz.\mathbf{R}^\sigma xy \to \mathbf{R}^\sigma yz \to \mathbf{R}^\sigma xz$. We also have $\vdash_{\mathcal{N}} \forall^\sigma xy.\mathbf{R}^\sigma xy \to \mathbf{R}^\sigma yx$.

Definition 5.10. A type σ is *reflexive* if $\vdash_{\mathcal{N}} \forall^\sigma x.\mathbf{R}^\sigma xx$.

Lemma 5.11. The types ι and o are reflexive.

One can use the model constructed in Example 5.4 of [1] to demonstrate $\nvdash_{\mathcal{N}} \forall^{oo}x.\mathbf{R}^{oo}xx$ and $\nvdash_{\mathcal{N}} \forall^{o\iota}x.\mathbf{R}^{o\iota}xx$. For more details see [18].

When translating tableau refutations we will sometimes need to consider a term t of type σ and need to know that $\Gamma \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}^{\sigma}(\Psi t)(\Psi t)$. One might try to prove this by a simple induction on t, but such an attempt will fail at the variable case. In general, we cannot prove $\Gamma \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}^{\sigma}\ x\ x$. However, we are able to prove (see Theorem 5.15) $\Gamma \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}^{\sigma}(\Psi t)(\Psi t)$ under the assumption that $\Gamma \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}^{\sigma}\ x\ x$ for every $x \in FV(t)$. Establishing Theorem 5.15 requires generalizing the result using simultaneous substitutions (Lemma 5.14). In some sense, Lemma 5.14 and Theorem 5.15 encapsulate a central reason why the translation works.

Lemma 5.12. For each $c \in \mathcal{C}$ we have $\vdash_{\mathcal{N}} \mathbf{R}\dot{c}\dot{c}$.

Proof. One must consider each case. Details are in [18]. □

Lemma 5.13. $\vdash_{\mathcal{N}} \forall fgxy.\dot{\neg}\dot{\neg}\mathbf{R}^{\sigma\tau}fg \to \dot{\neg}\dot{\neg}\mathbf{R}^{\sigma}xy \to \dot{\neg}\dot{\neg}\mathbf{R}^{\tau}(fx)(gy)$

Proof. This is straightforward using the definition of $\mathbf{R}^{\sigma\tau}$. □

Lemma 5.14. For all terms t, for all substitutions θ_1, θ_2 and, for all contexts Γ:

$$\text{if } \forall x \in FV(t) : \Gamma \vdash_{\mathcal{N}} \mathbf{R}(\theta_1(x))(\theta_2(x)) \text{ then } \Gamma \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}(\theta_1(\Psi t))(\theta_2(\Psi t))$$

Proof. We prove this lemma by structural induction on t.

- If t is a variable, then the result holds by the assumption.
- If t is a logical constant, then the result hold by Lemma 5.12.
- Suppose t is t_1t_2. By the inductive hypothesis we know $\Gamma \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}(\theta_1(\Psi t_1))(\theta_2(\Psi t_1))$ and $\Gamma \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}(\theta_1(\Psi t_2))(\theta_2(\Psi t_2))$. Hence $\Gamma \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}(\theta_1(\Psi t_1 t_2))(\theta_2(\Psi t_1 t_2))$ by Lemma 5.13.
- Suppose t is $\lambda x.t'$. Renaming variables if necessary, we assume x is chosen to avoid capture so that $\theta_1(\Psi t) = \lambda x.(\theta_1(\Psi t'))$ and $\theta_2(\Psi t) = \lambda x.(\theta_2(\Psi t'))$. It suffices to prove

  $$\Gamma \vdash_{\mathcal{N}} \mathbf{R}(\theta_1(\Psi \lambda x.t'))(\theta_2(\Psi \lambda x.t')).$$

  Let x_1 and x_2 be distinct fresh variables. Let Γ' be $\Gamma, (\mathbf{R}x_1x_2)$. For each $i \in \{1,2\}$ let θ'_i be $\theta_i, [x := x_i]$. It is easy to see that we have

  $$\forall y \in FV(t') : \Gamma' \vdash_{\mathcal{N}} \mathbf{R}(\theta'_1(y))(\theta'_2(y)),$$

  because of the assumption about $FV(t)$ and the fact that $FV(t') \subseteq FV(t) \cup \{x\}$. We apply the inductive hypothesis to obtain $\Gamma' \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}\theta'_1(\Psi t')\theta'_2(\Psi t')$. □

Theorem 5.15. For all terms t, and for all contexts Γ:

$$\text{if } \forall x \in FV(t) : \Gamma \vdash_{\mathcal{N}} \mathbf{R}xx \text{ then } \Gamma \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}(\Psi t)(\Psi t)$$

Proof. Follows directly from Lemma 5.14 by using the identity substitution. □

We need to extend Ψ to map branches to contexts. The most obvious extension that just maps Ψ to all the formulas in the branch to obtain a context does not have the properties we want. Using the model $\mathcal{M}^{\beta f}$ constructed in Example 5.4 of [1] one can prove $\nvdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}^{oo}xx$. Consequently, the branch $x \neq^{oo} x$ is $\mathcal{T}$-refutable but $\dot{\neg}\Psi(x =^{oo} x)$ is not $\mathcal{N}$-refutable.

In Definition 5.16 below we define a branch translation Ψ^*A which includes $\mathbf{R}xx$ for each free variable x in A. Following the definition we give a detailed example to further illustrate why such extra formulas are desired.

Definition 5.16 (The Branch Translation Ψ^*)**.** The branch translation Ψ^* maps a branch to a context as follows:

$$\Psi^* A := \{\Psi s | s \in A\} \cup \{\mathbf{R}^\sigma x\, x | x : \sigma \text{ and } x \in FV(A)\}$$

Example 5.17. Consider the formula $x \neq^{oo} x$. A tableau refutation of this formula starts with the branch $\{x \neq^{oo} x\}$. This branch is directly $\mathcal{T}$-refutable using the $\mathsf{Closed}\neq$ rule. To mimic the $\mathsf{Closed}\neq$ step in $\mathcal{N}$ we need to prove that $\Psi^*\{x \neq^{oo} x\}$ is $\mathcal{N}$-refutable. If $\Psi^*\{x \neq^{oo} x\}$ were defined as simply $\{\Psi(x \neq^{oo} x)\}$, then it would not be $\mathcal{N}$-refutable as mentioned above. Our definition of Ψ^* maps the branch $\{x \neq^{oo} x\}$ to the context $\{\dot\neg\mathbf{R}yy, \mathbf{R}yy\}$ which is obviously $\mathcal{N}$-refutable.

Example 5.18. We consider the translation of the formula $(\lambda p.\neg\neg p) = (\lambda p.p)$ proven in Example 3.3.

$$\begin{array}{lll} \Psi((\lambda p.\neg\neg p) = (\lambda p.p)) & \text{is} & \mathbf{R}^{oo}(\lambda p.\dot\neg\dot\neg p)(\lambda p.p) \\ & \text{is} & \forall pq : o.(p \dot\equiv q \rightarrow \dot\neg\dot\neg((\dot\neg\dot\neg p) \dot\equiv q)). \end{array}$$

A consequence of the final corollary in the next section is that the formula

$$\dot\neg\dot\neg\forall pq : o.(p \dot\equiv q \rightarrow \dot\neg\dot\neg((\dot\neg\dot\neg p) \dot\equiv q))$$

is $\mathcal{N}$-provable.

6 Translating Proofs

We prove that every $\mathcal{T}$-refutable branch A maps to an $\mathcal{N}$-refutable context $\Psi^* A$ (see Theorem 6.14). Implicitly this gives a translation from tableau refutations to natural deduction refutations. One may attempt to prove this by an induction on the $\mathcal{T}$-refutability of A, but a problem arises. Namely, a rule such as Forall may introduce a term t with free variables that prevent us from applying Theorem 5.15. In order to avoid this problem, we define a restricted tableau calculus $\mathcal{T}_r$.[1] We prove that if A is $\mathcal{T}$-refutable, then it is also $\mathcal{T}_r$-refutable. We then prove that if a branch A is $\mathcal{T}_r$-refutable, then $\Psi^* A$ is $\mathcal{N}$-refutable.

Definition 6.1 (Admissible for a Branch)**.** A term t is *admissible for a branch* A if for each variable $x \in FV(t)$, either $x \in FV(A)$ or x is of type ι or o.

Definition 6.2 (Tableau Calculus $\mathcal{T}_r$)**.** The *tableau calculus* $\mathcal{T}_r$ contains all the tableau rules that are in $\mathcal{T}$ (see Definition 3.1) except for Forall, $\mathsf{Func}=$, and Cut, for which it contains restricted forms as shown in Figure 3. This also defines the corresponding notion of $\mathcal{T}_r$-refutability.

In Proposition 6.7 below we prove that every $\mathcal{T}$-refutable branch is $\mathcal{T}_r$-refutable. The proof depends on a few simple lemmas.

Lemma 6.3 (Weakening)**.** If a branch A is $\mathcal{T}_r$-refutable, then every branch A' such that $A \subseteq A'$ is $\mathcal{T}_r$-refutable.

Proof. This is proven by induction on $\mathcal{T}_r$-refutability, taking care to rename variables from the Exists and $\mathsf{FuncExt}$ rules so they remain fresh. □

[1] The proof of Lemma 6.11 will show how using $\mathcal{T}_r$ resolves the problem.

$$\mathsf{Forall}_r\,\frac{\forall s}{\lceil s\ t\rceil^{\beta}}\ t \text{ is admissible for the branch} \qquad \mathsf{Cut}_r\,\frac{}{s \mid \neg s}\ s \text{ is admissible for the branch}$$

$$\mathsf{Func} =_r \frac{s_1 =^{\sigma\tau} s_2}{\lceil s_1\ t =^{\tau} s_2\ t\rceil^{\beta}}\ t \text{ is admissible for the branch}$$

Figure 3: Restricted Forall, Func=, and Cut Rules

Lemma 6.4. If $\neg\exists^{\sigma}x.x = x$ is in a branch A, then A is $\mathcal{T}_r$-refutable.

Proof. Using DeMorgan$\exists$ it suffices to prove $A \cup \{\forall^{\sigma}x.x \neq x\}$ is $\mathcal{T}_r$-refutable. The type σ has the form $\sigma_1 \cdots \sigma_n\alpha$ where $\alpha \in \{o, \iota\}$. Choose a variable y of type α and for each $i \in \{1, \ldots, n\}$ choose a variable z_i of type σ_i. Let t be the term $\lambda z_1 \cdots z_n.y$. We know $A \cup \{(\forall^{\sigma}x.x \neq x), t \neq t\}$ is $\mathcal{T}_r$-refutable using the Closed $\neq$ rule. The term t is of type σ and is admissible for the branch $A \cup \{\forall^{\sigma}x.x \neq x\}$ since y has type ι or o. Hence the rule Forall$_r$ justifies that $A \cup \{\forall^{\sigma}x.x \neq x\}$ is $\mathcal{T}_r$-refutable. □

Definition 6.5. Let X be a finite set of variables. We define $\mathcal{E}^X$ to be the branch $\bigcup_{x\in X}\{(\exists x.x = x), (x = x)\}$.

Lemma 6.6. Let A be a branch and X be a finite set of variables such that $X \cap FV(A) = \emptyset$. If $A \cup \mathcal{E}^X$ is $\mathcal{T}_r$-refutable, then A is $\mathcal{T}_r$-refutable.

Proof. We prove this by induction on the number of variables in X. If X is empty, then the result is trivial since $\mathcal{E}^{\emptyset}$ is empty. Suppose X is $Y \cup \{x\}$ where $x \notin Y$. In this case $\mathcal{E}^X$ is $\mathcal{E}^Y \cup \{(\exists x.x = x), (x = x)\}$. Since x is not free in $A \cup \mathcal{E}^Y \cup \{(\exists x.x = x)\}$ and $A \cup \mathcal{E}^X$ is $\mathcal{T}_r$-refutable, we know $A \cup \mathcal{E}^Y \cup \{\exists x.x = x\}$ is $\mathcal{T}_r$-refutable via the Exists rule. By Lemma 6.4 we also know $A \cup \mathcal{E}^Y \cup \{\neg\exists x.x = x\}$ is $\mathcal{T}_r$-refutable. By Cut$_r$ we know $A \cup \mathcal{E}^Y$ is $\mathcal{T}_r$-refutable. Finally, we conclude A is $\mathcal{T}_r$-refutable using the inductive hypothesis. □

Now we are in a position to prove that if a branch is $\mathcal{T}$-refutable, then it is $\mathcal{T}_r$-refutable. The proof implicitly describes an algorithm for modifying a tableau refutation so that it only uses the restricted rules.

Proposition 6.7. If a branch A is $\mathcal{T}$-refutable, then A is $\mathcal{T}_r$-refutable.

Proof. The proof is by induction on $\mathcal{T}$-refutability. Suppose $\mathcal{T}$-refutability of A follows from the step $\langle A_1, \ldots, A_n, A\rangle$ in $\mathcal{T}$ where A_i is $\mathcal{T}$-refutable for each $i \in \{1, \ldots, n\}$. By the inductive hypothesis, we know A_i is $\mathcal{T}_r$-refutable for each $i \in \{1, \ldots, n\}$. If $\langle A_1, \ldots, A_n, A\rangle$ is a step in $\mathcal{T}_r$, then we are done. Otherwise, $\langle A_1, \ldots, A_n, A\rangle$ must be a step in one of the Forall, Func=, or Cut rules and the new term used in the rule contains free variables not in A. We consider the Forall rule; the others are similar. Suppose $\forall^{\sigma}s \in A$, $n = 1$ and A_1 is $A, \lceil st\rceil^{\beta}$. Let X be $FV(t) \setminus FV(A)$. Clearly X is finite and $X \cap FV(A) = \emptyset$. By weakening (Lemma 6.3) and $\mathcal{T}_r$-refutability of A_1, we know $A, \lceil st\rceil^{\beta} \cup \mathcal{E}^X$ is $\mathcal{T}_r$-refutable. Clearly every free variable of t is free in $A, \lceil st\rceil^{\beta} \cup \mathcal{E}^X$. Hence, we can use the Forall$_r$ rule to conclude $A \cup \mathcal{E}^X$ is $\mathcal{T}_r$-refutable. By Lemma 6.6 we know A is $\mathcal{T}_r$-refutable. □

Corollary 6.8. The tableau calculus $\mathcal{T}_r$ is complete.

Proof. This is a direct consequence of Proposition 6.7 and the completeness of $\mathcal{T}$. □

We can now turn to the main part of the translation from restricted tableau refutations to natural deduction refutations. We prove that if A is $\mathcal{T}_r$-refutable, then Ψ^*A is $\mathcal{N}$-refutable. We can reduce this to a local property – the property of a rule being respected.

Definition 6.9. We say a rule (a set of steps) is *respected* if the following holds for every step $\langle A_1, \ldots, A_n, A \rangle$ in the rule: If $\Psi^*A_i \vdash_{\mathcal{N}} \dot{\bot}$ holds for all $i \in \{1, \ldots, n\}$, then $\Psi^*A \vdash_{\mathcal{N}} \dot{\bot}$ holds.

We prove that every rule defining $\mathcal{T}_r$ is respected. The fact that the rules are respected follows from the $\mathcal{N}$-provability of certain formulas. We prove this for the rules Exists and Forall_r in some detail. For the remaining rules we mainly give corresponding $\mathcal{N}$-provable formulas.

Lemma 6.10. The Exists rule is respected.

Proof. Let s be a term, A be a branch containing $\exists^\sigma s$, and x be a fresh variable.
Assume that $\Psi^*(A \cup \{\lceil sx \rceil^\beta\}) \vdash_{\mathcal{N}} \dot{\bot}$. We want to show that $\Psi^*A \vdash_{\mathcal{N}} \dot{\bot}$. Note that if y occurs free in $\lceil sx \rceil^\beta$ then

$$\Psi^*(A \cup \{\lceil sx \rceil^\beta\}) = \Psi^*(A) \cup \{\Psi(\lceil sx \rceil^\beta)\} \cup \{\mathbf{R}xx\},$$

otherwise

$$\Psi^*(A \cup \{\lceil sx \rceil^\beta\}) = \Psi^*(A) \cup \{\Psi(\lceil sx \rceil^\beta)\}.$$

In either case we know $\Psi^*(A) \cup \{\Psi(\lceil sx \rceil^\beta)\} \cup \{\mathbf{R}xx\} \vdash_{\mathcal{N}} \dot{\bot}$ by assumption and possibly the *wk* rule. By Lemma 5.3 we know $\Psi(\lceil sx \rceil^\beta)$ is β-equivalent to $\Psi(sx)$, i.e., $(\Psi s)x$. Using $\to I$ and β we have $\Psi^*A \vdash_{\mathcal{N}} \mathbf{R}^\sigma xx \to (\Psi s)x \to \dot{\bot}$. Since $x \notin FV(A)$ we can use $\forall I$ to obtain

$$\Psi^*A \vdash_{\mathcal{N}} \forall^\sigma x. \mathbf{R}^\sigma xx \to (\Psi s)x \to \dot{\bot}.$$

Since $\exists s \in A$, we know $\Psi(\exists s) \in \Psi^*A$ and thus $\Psi^*A \vdash_{\mathcal{N}} \dot{\exists}^\sigma(\Psi s)$ by the *hy* rule. The following is easy to verify (for a Coq proof term see [18, 19]) :

$$\vdash_{\mathcal{N}} \forall^{\sigma o} f. (\forall^\sigma x. \mathbf{R}^\sigma xx \to fx \to \dot{\bot}) \to (\dot{\exists}^\sigma f) \to \dot{\bot}.$$

Hence we have $\Psi^*A \vdash_{\mathcal{N}} \dot{\bot}$ as desired. □

The argument for the Forall_r rule is similar. In this case we must make use of Theorem 5.15.

Lemma 6.11. The Forall_r rule is respected.

Proof. Let s be a term, A be a branch containing $\forall^\sigma s$, and t be a term admissible for A. Note that

$$\Psi^*(A \cup \{\lceil st \rceil^\beta\}) \subseteq \Psi^*(A) \cup \{\Psi(\lceil st \rceil^\beta)\} \cup \{\mathbf{R}xx | x \in FV(t)\}.$$

By assumption and possibly the *wk* rule we know

$$\Psi^*(A) \cup \{\Psi(\lceil st \rceil^\beta)\} \cup \{\mathbf{R}xx | x \in FV(t)\} \vdash_{\mathcal{N}} \dot{\bot}.$$

We know t is admissible. Thus for each $x \in FV(t)$ either $x \in FV(A)$ or x has the reflexive type ι or o. Hence for each $x \in FV(t)$ we know $\Psi^*A \vdash_{\mathcal{N}} \mathbf{R}xx$. Consequently, we have $\Psi^*(A) \cup \{\Psi(\lceil st \rceil^\beta)\} \vdash_{\mathcal{N}} \dot{\bot}$. Using Lemma 5.3 we know

$$\Psi^*A \vdash_{\mathcal{N}} (\Psi s)(\Psi t) \to \dot{\bot}.$$

Applying Theorem 5.15 we also have

$$\Psi^*A \vdash_{\mathcal{N}} \dot{\neg}\dot{\neg}\mathbf{R}^\sigma(\Psi t)(\Psi t).$$

Closed: $\forall^o p.p \to (\dot{\neg} p) \to \dot{\bot}$

Closed$\bot$: $\dot{\bot} \to \dot{\bot}$

Closed$\neg\top$: $(\dot{\neg}\dot{\top}) \to \dot{\bot}$

Closed$\neq$: $\forall^\sigma x.\dot{\neg}\dot{\neg}(\mathbf{R}^\sigma xx) \to \dot{\neg}(\mathbf{R}^\sigma xx) \to \dot{\bot}$

ClosedSym: $\forall^\sigma xy.(\mathbf{R}^\sigma xy) \to \dot{\neg}(\mathbf{R}^\sigma yx) \to \dot{\bot}$

Cut$_r$: $\forall^o p.(p \to \dot{\bot}) \to ((\dot{\neg} p) \to \dot{\bot}) \to \dot{\bot}$

DNeg: $\forall^o p.(p \to \dot{\bot}) \to (\dot{\neg}\dot{\neg} p) \to \dot{\bot}$

And: $\forall p\, q.(p \to q \to \dot{\bot}) \to (p \dot{\wedge} q) \to \dot{\bot}$

Or: $\forall p\, q.(p \to \dot{\bot}) \to (q \to \dot{\bot}) \to (p \dot{\vee} q) \to \dot{\bot}$

Imp: $\forall^o p\, q.((\dot{\neg} p) \to \dot{\bot}) \to (q \to \dot{\bot}) \to (p \to q) \to \dot{\bot}$

NegAnd: $\forall p\, q.((\dot{\neg} p) \to \dot{\bot}) \to ((\dot{\neg} q) \to \dot{\bot}) \to \dot{\neg}(p \dot{\wedge} q) \to \dot{\bot}$

NegOr: $\forall p\, q.((\dot{\neg} p) \to (\dot{\neg} q) \to \dot{\bot}) \to \dot{\neg}(p \dot{\vee} q) \to \dot{\bot}$

NegImp: $\forall^o p\, q.(p \to (\dot{\neg} q) \to \dot{\bot}) \to (\dot{\neg}(p \to q)) \to \dot{\bot}$

DeMorgan$\forall$: $\forall^{\sigma \to o} f.((\dot{\exists}^\sigma (\lambda x.\dot{\neg} fx)) \to \dot{\bot}) \to (\dot{\neg}(\dot{\forall}^\sigma f)) \to \dot{\bot}$

DeMorgan$\exists$: $\forall^{\sigma \to o} f.((\dot{\forall}^\sigma (\lambda x.\dot{\neg} fx)) \to \dot{\bot}) \to (\dot{\neg}(\dot{\exists}^\sigma f)) \to \dot{\bot}$

Bool$=$: $\forall^o pq.(p \to q \to \dot{\bot}) \to ((\dot{\neg} p) \to (\dot{\neg} q) \to \dot{\bot}) \to (\mathbf{R}^o pq) \to \dot{\bot}$

BoolExt: $\forall^o pq.(p \to (\dot{\neg} q) \to \dot{\bot}) \to (q \to (\dot{\neg} p) \to \dot{\bot}) \to \dot{\neg}(\mathbf{R}^o pq) \to \dot{\bot}$

FuncExt: $\forall^{\sigma\tau} kh.\dot{\neg}\dot{\neg}(\mathbf{R}^{\sigma\tau} hh) \to (\forall^\sigma x.(\mathbf{R}^\sigma xx) \to \dot{\neg}(\mathbf{R}^\tau (kx)(hx)) \to \dot{\bot}) \to \dot{\neg}(\mathbf{R}^{\sigma\tau} kh) \to \dot{\bot}$

Func$=_r$: $\forall^{\sigma\tau} kh.\forall^\sigma t.\dot{\neg}\dot{\neg}(\mathbf{R}^\sigma tt) \to ((\mathbf{R}^\tau (kt)(ht)) \to \dot{\bot}) \to (\mathbf{R}^{\sigma\tau} kh) \to \dot{\bot}$

Mat: $\forall^{\sigma_1\sigma_2\cdots\sigma_n o} p.\forall^{\sigma_1} x_1 y_1.\forall^{\sigma_2} x_2 y_2.\ldots.\forall^{\sigma_n} x_n y_n.\dot{\neg}\dot{\neg}(\mathbf{R}^{\sigma_1\sigma_2\cdots\sigma_n o} pp) \to$
$(\dot{\neg}(\mathbf{R}^{\sigma_1} x_1 y_1) \to \dot{\bot}) \to (\dot{\neg}(\mathbf{R}^{\sigma_2} x_2 y_2) \to \dot{\bot}) \to \cdots \to (\dot{\neg}(\mathbf{R}^{\sigma_n} x_n y_n) \to \dot{\bot}) \to$
$px_1 x_2 \ldots x_n \to \dot{\neg}(py_1 y_2 \ldots y_n) \to \dot{\bot}$

Dec: $\forall^{\sigma_1\sigma_2\cdots\sigma_n \iota} h.\forall^{\sigma_1} x_1 y_1.\forall^{\sigma_2} x_2 y_2.\ldots.\forall^{\sigma_n} x_n y_n.\dot{\neg}\dot{\neg}(\mathbf{R}^{\sigma_1\sigma_2\cdots\sigma_n \iota} hh) \to$
$(\dot{\neg}(\mathbf{R}^{\sigma_1} x_1 y_1) \to \dot{\bot}) \to (\dot{\neg}(\mathbf{R}^{\sigma_2} x_2 y_2) \to \dot{\bot}) \to \cdots \to (\dot{\neg}(\mathbf{R}^{\sigma_n} x_n y_n) \to \dot{\bot}) \to$
$\dot{\neg}(\mathbf{R}^\iota (hx_1 x_2 \ldots x_n)(hy_1 y_2 \ldots y_n)) \to \dot{\bot}$

Con: $\forall^\iota xyzw.(\dot{\neg}(\mathbf{R}^\iota xz) \to \dot{\neg}(\mathbf{R}^\iota yz) \to \dot{\bot}) \to (\dot{\neg}(\mathbf{R}^\iota xw) \to \dot{\neg}(\mathbf{R}^\iota yw) \to \dot{\bot}) \to$
$(\mathbf{R}^\iota xy) \to \dot{\neg}(\mathbf{R}^\iota zw) \to \dot{\bot}$

Figure 4: Formulas provable in $\mathcal{N}$ used in the proof of Lemma 6.12

Since $\forall s \in A$, we know $\Psi(\forall s) \in \Psi^* A$ thus $\Psi^* A \vdash_{\mathcal{N}} \dot{\forall}^\sigma(\Psi s)$ by the *hy* rule. The following is easy to verify (for a Coq proof term see [18, 19]) :

$$\vdash_{\mathcal{N}} \forall^{\sigma o} f.\forall^\sigma x.\dot{\neg}\dot{\neg}\mathbf{R}^\sigma xx \to (fx \to \dot{\bot}) \to (\dot{\forall}^\sigma f) \to \dot{\bot}$$

Hence we have $\Psi^* A \vdash_{\mathcal{N}} \dot{\bot}$ as desired. □

The proofs of Lemmas 6.10 and 6.11 illustrate how one proves that a rule is respected. For the remaining rules we will simply indicate the formulas whose $\mathcal{N}$-provability implies the rule is respected.

Lemma 6.12. All of the rules in $\mathcal{T}_r$ are respected.

Proof. We have already proven this for Exist in Lemma 6.10 and Forall$_r$ in Lemma 6.11. For the remaining rules one can argue similarly making use of formulas which are provable in $\mathcal{N}$ and correspond to the structure of the rule. We display formulas corresponding to the remaining rules in Figure 4. Proof terms (in Coq) for these formulas are available in [18, 19] with the exception of the lemmas for Mat and Dec. The lemmas for Mat and Dec have been formulated and proven in Coq for the cases with 1 and 2 arguments. □

Proposition 6.13. If A is $\mathcal{T}_r$-refutable, then Ψ^*A is $\mathcal{N}$-refutable.

Proof. The proof is by an easy induction on the $\mathcal{T}_r$-refutation using Lemma 6.12 at each step. □

We finally conclude similar results for $\mathcal{T}$-refutability.

Theorem 6.14. If A is $\mathcal{T}$-refutable, then Ψ^*A is $\mathcal{N}$-refutable. Also, if s is a ground formula and s is $\mathcal{T}$-refutable, then Ψs is $\mathcal{N}$-refutable.

Proof. This follows from Propositions 6.7 and 6.13. □

We can also conclude the following using Theorem 6.14 and Corollary 5.8.

Corollary 6.15. Let s be a formula such that $FV(s) = \{x_1, \ldots, x_n\}$. If s is $\mathcal{T}$-provable, then $s \approx (x_1 \dot{=} x_1 \rightarrow \cdots \rightarrow x_n \dot{=} x_n \rightarrow \dot{\neg}\dot{\neg}\Psi s)$ and $(x_1 \dot{=} x_1 \rightarrow \cdots \rightarrow x_n \dot{=} x_n \rightarrow \dot{\neg}\dot{\neg}\Psi s)$ is $\mathcal{N}$-provable.

7 Conclusion

Given a higher-order formula s and a classical extensional tableau proof of s, our aim was to find a formula s' that is semantically equivalent to s and construct an intuitionistic intentional natural deduction proof of s'. We defined two tableau calculi $\mathcal{T}$ and $\mathcal{T}_r$ and proved that whenever a branch is $\mathcal{T}$-refutable, it is also $\mathcal{T}_r$-refutable (Proposition 6.7). Moreover, we gave a translation Ψ and proved that it maps higher-order formulas to semantically equivalent formulas in the sense of Henkin semantics (Theorem 5.7). Furthermore, we defined a branch translation Ψ^* that maps branches to contexts and proved that for any $\mathcal{T}_r$-refutable branch A, Ψ^*A is $\mathcal{N}$-refutable (Proposition 6.13). We concluded that for any $\mathcal{T}$-provable formula s with free variables $x_1, \ldots, x_n$, the formula

$$\Psi(x_1 = x_1) \rightarrow \cdots \rightarrow \Psi(x_n = x_n) \rightarrow \Psi(\neg\neg s)$$

is semantically equivalent to s and is $\mathcal{N}$-provable (Corollary 6.15). Hence for any ground formula s that is $\mathcal{T}$-provable, $\Psi(\neg\neg s)$ is $\mathcal{N}$-provable.

Several issues are still open for future work. One may want to determine the precise relationship between our translation Ψ and the translation given by Gandy [9]. One could also investigate to what extent the translation can be extended to handle a choice operator.

A choice operator is a logical constant $\varepsilon^\sigma : (\sigma o)\sigma$ satisfying $\forall p : \sigma o.\forall x : \sigma.px \rightarrow p(\varepsilon^\sigma p)$. Such operators are supported by Satallax. To extend the translation in this paper to handle ε^σ one would need to find a term $\Psi\varepsilon^\sigma$ satisfying

$$\vdash_{\mathcal{N}} \Psi(\neg\neg(\varepsilon^\sigma = \varepsilon^\sigma))$$

and

$$\vdash_{\mathcal{N}} \Psi(\neg\neg\forall p : \sigma o.\forall x : \sigma.px \rightarrow p(\varepsilon^\sigma p)).$$

This will not generally be possible, but might be possible in special situations. For example, one might restrict to having the choice operator only at the base type ι and assume that the base type ι is finite.

On the more practical side, one can implement a mapping from tableau proofs to natural deduction proof terms. This would enable proof checking the tableau proofs that Satallax outputs using Coq. This implementation could make use of the Coq lemmas that are provided in [18, 19].

References

[1] C. Benzmüller, C. E. Brown, and M. Kohlhase. Higher-order semantics and extensionality. *Journal of Symbolic Logic*, 69:1027–1088, 2004.

[2] C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. LEO-II — A cooperative automatic theorem prover for classical higher-order logic. In *Fourth International Joint Conference on Automated Reasoning (IJCAR'08)*, volume 5195 of *LNCS (LNAI)*, pages 162–170. Springer, 2008.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[4] C. E. Brown. Satallax: An automated higher-order prover. In U. S. Bernhard Gramlich, Dale Miller, editor, *6th International Joint Conference on Automated Reasoning (IJCAR 2012)*, pages 111 – 117. Springer, 2012.

[5] C. E. Brown and C. Rizkallah. Glivenko and Kuroda for simple type theory. Technical report, Saarland University, Dec 2011. Article to be published in *Journal of Symbolic Logic*.

[6] C. E. Brown and G. Smolka. Analytic tableaux for simple type theory and its first-order fragment. *Logical Methods in Computer Science*, 6(2), Jun 2010.

[7] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[8] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 579–606. Academic Press, 1980.

[9] R. O. Gandy. On the axiom of extensionality–part I. *Journal of Symbolic Logic*, 21(1):36–48, 1956.

[10] G. Gentzen. Untersuchungen über das natürliche Schließen I, II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[11] V. Glivenko. Sur quelques points de la logique de M. Brouwer. *Bulletins de la classe des sciences*, 15:183–188, 1929.

[12] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, June 1950.

[13] W. A. Howard. The formula-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 480–490. Academic Press, 1980.

[14] S. Kuroda. Intuitionistische Untersuchungen der formalistischen Logik. *Nagoya Mathematical Journal*, 2:35–47, 1951.

[15] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[17] D. Prawitz. *Natural deduction: a proof-theoretical study.* PhD thesis, Almqvist & Wiksell, 1965.

[18] C. Rizkallah. Proof representations for higher-order logic. Master's thesis, Saarland University, Saarbruecken, Germany, Dec 2009.

[19] C. Rizkallah. Proof representations for higher-order logic: Coq proofs, 2009. http://www.mpi-inf.mpg.de/~crizkall/Full_Tableau_ND_Translation.v.

[20] A. Teucke. Translating a Satallax refutation to a tableau refutation encoded in Coq. Bachelor's thesis, Universität des Saarlandes, 2011.

A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo*

Guillaume Burel

ÉNSIIE/Cédric, 1 square de la rsistance, 91025 Évry cedex, France
guillaume.burel@ensiie.fr   http://www.ensiie.fr/~guillaume.burel/

Abstract

The $\lambda\Pi$-calculus modulo is a proof language that has been proposed as a proof standard for (re-)checking and interoperability. Resolution and superposition are proof-search methods that are used in state-of-the-art first-order automated theorem provers. We provide a shallow embedding of resolution and superposition proofs in the $\lambda\Pi$-calculus modulo, thus offering a way to check these proofs in a trusted setting, and to combine them with other proofs. We implement this embedding as a back-end of the prover iProver Modulo.

Introduction

Proof assistants have now achieved a quite high degree of maturity, and are able to certify rather big projects. One can for instance cite the certified compiler CompCert by Coq [14], or the seL4 micro-kernel specification in Isabelle/HOL [12]. Nevertheless, some of the current challenges concerning proof assistants are to overcome their lack of automation, and to help them cooperate better to share proof developments. A way of making proof assistants more automated is to delegate proof obligation to external automated theorem provers. This is for instance what the Sledgehammer [3] subsystem of Isabelle/HOL does, which passes on proof obligations to first-order automated theorem provers such as E or SPASS, or SMT solvers like CVC3, Yices or Z3. To keep confidence in the whole proof, the question arises of the combination of the proof found by the automated prover and the rest of the proof-assistant development. For Sledgehammer, this is done by reproving the proof obligation with an Isabelle/HOL tactic, namely Metis, only keeping the information of which lemmas were needed by the automated prover to find the proof and searching the proof again from scratch using only these lemmas. Of course, it would be more interesting to directly retrieve the proof of the automated prover and to translate it into an Isabelle/HOL proof. However, automated theorem do not often output proofs, and when they do, it is not trivial to translate them into a proof assistant format. Furthermore, such a translation would have to be performed for each pair automated prover/proof assistant.

Another solution would be to have a single, universal proof format in which every part of a big proof would be translated and combined. An analogy can be drawn with the interoperability of programming languages, that are translated into an assembly language in which the linking is performed. Ideally, this universal standard for proofs should have the following properties: It should be simple, so that it should be easy to write a proof checker in which one could therefore have a high degree of confidence. Moreover, it should be expressive enough to be able to embed the basic logics of all theorem provers and proof assistants available. To help proof recombination, these embeddings should also be shallow. Although there is to the author's

*This work is supported by the French ANR project BWare.

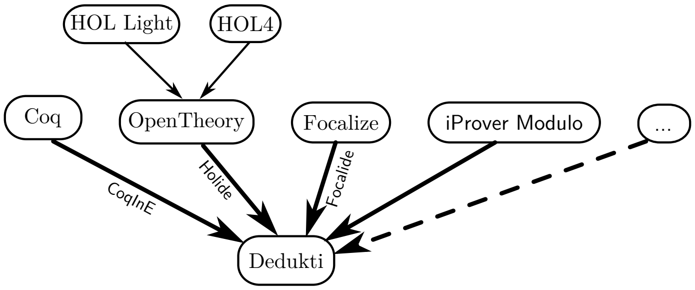


Figure 1: Dedukti as a universal proof language

knowledge no precise definition of what a shallow embedding is, it can be distinguished from a deep embedding by the fact that it reuse the features of the target language. For instance, connectives are translated as connectives, and not as constants, and the same for variables, binders, computations, etc. Now suppose that we have two input proof languages A and B, with respective embeddings $||\cdot||_A$ and $||\cdot||_B$ into our target standard. We would like to combine a proof of $P \Rightarrow Q$ in A and a proof of P in B to get a proof of Q. Using deep embeddings, it would be hard to relate the translation $||P \Rightarrow Q||_A$ and $||P||_B$, that could a priori have nothing in common. On the contrary, using a shallow embedding, $||P \Rightarrow Q||_A$ would be equal to $||P||_A \to ||Q||_A$, where $\to$ is the implication of the target language. Therefore, it only remains to relate $||P||_A$ and $||P||_B$ which should be easier.

The λΠ-calculus modulo [7, 5] is a proposed standard for proof interoperability. It is relatively simple, and an already efficient interpreter for it takes only a few hundred lines of code. The λΠ-calculus modulo is an extension of the λΠ-calculus, a proof language for minimal first-order logic also known as LF, λP, etc [11]. In the λΠ-calculus modulo, it is possible to have shallow embeddings of higher-order logics, which is not possible in pure λΠ-calculus. Cousineau and Dowek [7] have shown that any pure type system can be shallowly embedded into the λΠ-calculus modulo, including for instance the Calculus of Construction which serves as basis of the proof assistant Coq. Assaf [1] has proved that simple type theory (a.k.a. higher-order logic), that is the foundation of proof assistants of the HOL family, can also be translated in the λΠ-calculus modulo in a shallow way. The λΠ-calculus modulo seems therefore a good candidate for a universal standard for proofs.

Following this idea, a language called Dedukti[1] was designed to declare proofs of the λΠ-calculus modulo, and a proof checker for this language, namely dkparse, was implemented. dkparse is available at `https://www.rocq.inria.fr/deducteam/Dedukti/` . Tools related to Dedukti also include a translator of Coq proofs to Dedukti, namely CoqInE [4, `http://www.ensiie.fr/~guillaume.burel/blackandwhite_coqInE.html.en`], and a translator from OpenTheory proofs (a standard for proofs of the HOL family) to Dedukti, namely Holide [1, `https://www.rocq.inria.fr/deducteam/Holide/`]. There exists also a prototype of a back-end of the certifying programming environment FoCaLiZe to Dedukti, namely Focalide [`https://www.rocq.inria.fr/deducteam/Focalide/`]. Figure 1 summarizes the current tools available around Dedukti.

Current state-of-the-art automated theorem provers for first-order logic are based on the superposition calculus [2], which can be seen as an extension of the resolution method [17]. This includes for instance the provers Vampire [16], SPASS [20] or E [18]. To be able to

[1] "Dedukti" means "to deduce" in Esperanto.

$$\textsf{Empty}\ \frac{}{\emptyset\ \mathrm{WF}} \qquad \textsf{Declaration}\ \frac{\Gamma\ \mathrm{WF} \qquad \Gamma\vdash A : s \qquad x\notin\Gamma}{\Gamma, x : A\ \mathrm{WF}}\ s\in\{\textsf{Type},\textsf{Kind}\}$$

$$\textsf{Sort}\ \frac{\Gamma\ \mathrm{WF}}{\Gamma\vdash \textsf{Type} : \textsf{Kind}} \qquad \textsf{Variable}\ \frac{\Gamma\ \mathrm{WF} \qquad x : A\in\Gamma}{\Gamma\vdash x : A}$$

$$\textsf{Product}\ \frac{\Gamma\vdash A : \textsf{Type} \qquad \Gamma, x : A\vdash B : s}{\Gamma\vdash \Pi x : A.\ B : s}\ s\in\{\textsf{Type},\textsf{Kind}\}$$

$$\textsf{Application}\ \frac{\Gamma\vdash T : \Pi x : A.\ B \qquad \Gamma\vdash U : A}{\Gamma\vdash (T\ U) : \{U/x\}B}$$

$$\textsf{Abstraction}\ \frac{\Gamma\vdash A : \textsf{Type} \qquad \Gamma, x : A\vdash B : s \qquad \Gamma, x : A\vdash T : B}{\Gamma\vdash \lambda x : A.\ T : \Pi x : A.\ B}\ s\in\{\textsf{Type},\textsf{Kind}\}$$

$$\textsf{Conversion}\ \frac{\Gamma\vdash T : A \qquad \Gamma\vdash A : s \qquad \Gamma\vdash B : s}{\Gamma\vdash T : B}\ s\in\{\textsf{Type},\textsf{Kind}\}\text{ and }A\equiv_\beta B$$

Figure 2: Type System for the λΠ-calculus

combine proofs from these provers with the developments of a proof assistant, we therefore want to translate them in the λΠ-calculus in a shallow manner. In this paper, we show how this is possible. However, note that we only show how to translate resolution and superposition proofs, and not how the translate the transformation of the original problem into clausal normal form. As remarked in Section 3.3, this means that we only need intuitionistic logic. We also present an implementation of this translation in the prover iProver Modulo, which is therefore able to produce proofs in Dedukti's format.

In next section, we present formally the λΠ-calculus modulo. In Section 2, we describe the shallow embedding of first-order logic with equality in the λΠ-calculus modulo. Section 3 details the translation of resolution and superposition proofs. Its implementation in iProver Modulo is outlined in Section 4.

1 The λΠ-Calculus Modulo

The λΠ-calculus modulo [7, 5] is an extension of the λΠ-calculus, that can be seen as a proof language for minimal first-order logic and that is also known as LF, λP, etc [11]. The λΠ-calculus is based on the Curry-Howard-DeBruijn correspondence, which means that proofs are represented by λ-terms and formulas by their types, and it can be seen as one of the simplest coherent Pure Type System, which means that there is no syntactic distinction between terms and types.

Pre-terms in the λΠ-calculus are defined by the grammar

$$M, N, A, B ::= x \mid \lambda x : A.\ M \mid \Pi x : A.\ B \mid M\ N \mid \textsf{Type} \mid \textsf{Kind}$$

where x is an element of an infinite set of variables. A context is a set of couples of variables and pre-terms. A pre-term will be called a term when it is well-typed in the type system of Figure 2, where the judgment "Γ WF" means that a context Γ is well-formed, and the judgment $\Gamma\vdash T : A$ must be read as "T has type A in the context Γ". Remark that contrarily to other versions of LF, η-conversion is not considered.

In the Conversion rule of the $\lambda\Pi$-calculus, $A \equiv_\beta B$ means that A and B are β-convertible. In the $\lambda\Pi$-calculus *modulo*, this conversion rule is extended by well-typed rewriting rules:

Definition 1 (Rewriting rule)**.** A rewriting rule is a quadruple $\Delta :\cdot\ l \hookrightarrow^A r$ composed of a context Δ and three terms l, r and A. It is well typed in a context Γ if:

- the context Γ, Δ is well-formed;
- $\Gamma, \Delta \vdash l : A$ and $\Gamma, \Delta \vdash r : A$ are derivable judgments.

Intuitively, Δ contains the type of the free variables of l and r, and A ensures that l and r have the same type, which warrants the preservation of types through rewriting. In the rewriting rules that we use in the following, Δ and A can often be inferred from l and r, in which case we will omit them and simply write $l \hookrightarrow r$. As usual, a term s is rewritten by a rewriting rule $l \hookrightarrow r$ to a term t if there exists a substitution δ such that a subterm of s at a position $\mathfrak{p}$ is equal to $\delta(l)$ and t is equal to s where the subterm at position $\mathfrak{p}$ is replaced by $\delta(r)$. Note that the domain of δ is the set of variables in the context Δ of the rule.

In the $\lambda\Pi$-calculus modulo, contexts can contain rewriting rules, and the type system of the $\lambda\Pi$-calculus is therefore extended by a new rule adding a well-typed rewriting rule in a context:

$$\textsf{Rewrite}\ \frac{\Gamma, \Delta \vdash l : A \qquad \Gamma, \Delta \vdash r : A}{\Gamma, (\Delta :\cdot\ l \hookrightarrow^A r)\ \mathrm{WF}}$$

Given a context Γ, we let $\equiv_\Gamma$ be the smallest congruence generated by β-reduction and the rewriting rules of Γ. The conversion rule of the $\lambda\Pi$-calculus is then replaced by the following one:

$$\textsf{Conversion}\ \frac{\Gamma \vdash T : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash T : B}\ s \in \{\textsf{Type}, \textsf{Kind}\} \text{ and } A \equiv_\Gamma B$$

The case of the $\lambda\Pi$-calculus without modulo is regained when the contexts do not contain any rewriting rules.

A file in Dedukti's format is a declaration of a context of the $\lambda\Pi$-calculus modulo. Syntactically, $\lambda x : a.\ t$ and $\Pi x : a.\ b$ are respectively written `x : a => t` and `x : a -> b`, and a rewriting rule $\Delta :\cdot\ l \hookrightarrow r$ is declared as `[Δ] l --> r`. The tool dkparse checks that such a context is well-formed, in particular it checks that rewriting rules are well-typed. Dedukti's syntax also allows the declaration of constant definitions, with the syntax `c : a := t`. It can be seen as the combination of a declaration `c : a` and a rewriting rule `[] c --> t`. However, a definition is not expanded, and it is safe in the sense that it does not change the theory defined by the context. Contrarily, if a constant of type B is declared, but it is not rewritten, this can be seen as assuming the axiom B. For a function defined by means of rewriting rules, such as proof, only an exhaustiveness checker can tell us whether the theory changes or not.

Note that dkparse assumes that the given rewriting rules are strongly terminating and confluent. The philosophy behind Dedukti's proof environment is indeed to have several tools that are each specialized in a particular task. dkparse is only concerned with type checking, whereas other (presently nonexistent) tools should check the convergence of the rewriting rules or the exhaustiveness of rewriting-defined functions.

2 Translating First-Order Logic to $\lambda\Pi$-Calculus Modulo

2.1 Deep and Shallow Embedding of First-Order Logic

This section is based on Dorra's work [8], which itself borrows ideas from the embedding of pure type systems in the $\lambda\Pi$-calculus modulo [7].

We use standard definitions for terms, predicates, first-order propositions (with connectives $\bot, \neg, \Rightarrow, \wedge, \vee$ and quantifiers $\forall, \exists$) as can be found in [10].

The translation of first-order logic in the $\lambda\Pi$-calculus modulo consists of two embeddings, one deep $|\cdot|$ and one shallow $||\cdot||$, that are linked by a decoding function proof that is defined by means of rewriting rules.

To define the deep embedding, we first define two constants ι and o of type Type that contain respectively the translation of terms and propositions. We add constants $\dot{\bot} : o$, $\dot{\neg} : o \to o$, $\dot{\Rightarrow} : o \to o \to o$, $\dot{\vee} : o \to o \to o$, $\dot{\wedge} : o \to o \to o$, $\dot{\forall} : (\iota \to o) \to o$, $\dot{\exists} : (\iota \to o) \to o$ for the translation of connectives and quantifiers. For each function symbol f of arity n we add a constant $f : \underbrace{\iota \to \cdots \to \iota \to}_{n \text{ times}} \iota$, and for each predicate symbol p of arity n we add a constant $\dot{p} : \underbrace{\iota \to \cdots \to \iota \to}_{n \text{ times}} o$. In a context $X_1 : \iota, \dots, X_m : \iota$ where $X_1, \dots, X_m$ are the free variables of a formula A, we can then translate formulas by induction:

$$\begin{array}{rclrcl}
|X| &=& X \quad \text{(if } X \text{ is a variable)} & |f(t1, \dots, t_n)| &=& f\ |t_1|\ \cdots\ |t_n| \\
|p(t1, \dots, t_n)| &=& \dot{p}\ |t_1|\ \cdots\ |t_n| & |\bot| &=& \dot{\bot} \\
|\neg A| &=& \dot{\neg}\ |A| & |A \Rightarrow B| &=& \dot{\Rightarrow}\ |A|\ |B| \\
|A \vee B| &=& \dot{\vee}\ |A|\ |B| & |A \wedge B| &=& \dot{\wedge}\ |A|\ |B| \\
|\forall X.A| &=& \dot{\forall}\ (\lambda X : \iota.\ |A|) & |\exists X.A| &=& \dot{\exists}\ (\lambda X : \iota.\ |A|)
\end{array}$$

The shallow embedding is defined by $||A|| = \mathsf{proof}\ |A|$ where proof is a decoding function of type $o \to \mathsf{Type}$. What makes this translation shallow is the definition of the decoding function by means of rewriting rules, that relates the deep embedding of connectives with their counterparts in $\lambda\Pi$-calculus modulo. $\dot{\Rightarrow}$ is for instance related with $\to$, $\dot{\forall}$ with Π, whereas the other connectives are related with their impredicative encoding in $\lambda\Pi$, to use the connectors of the $\lambda\Pi$-calculus; this makes them more shallow than using a translation to a constant. We can add a constant p of type $\underbrace{\iota \to \cdots \to \iota \to}_{n \text{ times}} \mathsf{Type}$ to get a shallow embedding of each predicate symbol p whose arity is n. The rules defining proof are therefore:

$$\begin{array}{rcl}
\mathsf{proof}\ (\dot{p}\ t_1\ \cdots\ t_n) &\hookrightarrow& p\ t_1\ \cdots\ t_n \\
\mathsf{proof}\ \dot{\bot} &\hookrightarrow& \Pi\ \flat : o.\ \mathsf{proof}\ \flat \\
\mathsf{proof}\ (\dot{\neg}\ A) &\hookrightarrow& \Pi\ \flat : o.\ \mathsf{proof}\ A \to \mathsf{proof}\ \flat \\
\mathsf{proof}\ (\dot{\Rightarrow}\ A\ B) &\hookrightarrow& \mathsf{proof}\ A \to \mathsf{proof}\ B \\
\mathsf{proof}\ (\dot{\vee}\ A\ B) &\hookrightarrow& \Pi\ \flat : o.\ (\mathsf{proof} A \to \mathsf{proof}\ \flat) \to (\mathsf{proof} B \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat \\
\mathsf{proof}\ (\dot{\wedge}\ A\ B) &\hookrightarrow& \Pi\ \flat : o.\ (\mathsf{proof} A \to \mathsf{proof} B \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat \\
\mathsf{proof}\ (\dot{\forall}\ f) &\hookrightarrow& \Pi X : \iota.\ \mathsf{proof}\ (f\ X) \\
\mathsf{proof}\ (\dot{\exists}\ f) &\hookrightarrow& \Pi\ \flat : o.\ (\Pi X : \iota.\ \mathsf{proof}\ (f\ X) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat
\end{array}$$

where $\flat$ is a variable that does not appear in any first-order formula to avoid capture. Note that the rules for $\mathsf{proof}\ (\dot{\forall}\ f)$ and $\mathsf{proof}\ (\dot{\exists}\ f)$ do not introduce a fresh variable, since X is bound by Π. Of course, when applying such rule to a term t containing the variable X, substituting f by t in the right-hand side should not capture the X bound by Π.

It can be proved that this translation is sound, that is that if a formula A is provable in intuitionistic first-order logic, then there exists a term of type $||A||$ in the $\lambda\Pi$-calculus modulo with the environment described above. It is also a conservative extension of intuitionistic first-order logic, in the sense that for all first-order formula A, if the type $||A||$ is inhabited in the environment defined above, then A is provable in intuitionistic first-order logic.

Resolution and superposition are proof-search methods for first-order logic. They manipulate clauses. A literal is either an atomic formula (i.e. a predicate symbol applied to as many terms as its arity) or the negation of an atomic formula. A clause is a list of literals $L_1; \cdots ; L_m$. It corresponds to the formula $\forall X_1. \ \ldots \forall X_n. \ L_1 \vee \cdots \vee L_m$ where $X_1, \ldots, X_n$ are the free variables of $L_1, \ldots, L_m$. To ease the translation of resolution and superposition proofs, we translate clauses directly into a shallow embedding: A clause $L_1; \cdots ; L_m$ is translated as

$$||L_1; \cdots ; L_m|| = \Pi X_1 : \iota. \ \ldots \Pi X_n : \iota. \ \Pi \ \flat : o. \ [\![L_1]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof} \ \flat$$

where $X_1, \ldots, X_n$ are the free variables in the clause and $[\![P]\!]_\flat = ||P|| \to \mathsf{proof} \ \flat$ for a positive literal P and $[\![\neg P]\!]_\flat = (||P|| \to \mathsf{proof} \ \flat) \to \mathsf{proof} \ \flat$ for a negative literal $\neg P$. The empty clause is therefore translated as $\Pi \ \flat : o. \ \mathsf{proof} \ \flat$, which is also the translation of $\bot$ as expected. It can be shown that the translation of a clause $L_1; \cdots ; L_m$ is implied by the translation of the corresponding formula $\forall X_1. \ \ldots \forall X_n. \ L_1 \vee \cdots \vee L_m$. To get the other direction, one needs a classical axiom, for instance in the case of a clause containing only one literal.

2.2 Equality

The equality predicate $\simeq$ is so pervasive that it is often useful to have a specific treatment of it. For instance, the resolution method was extended into the superposition method to handle the equality better. To have a shallower translation of first-order logic with equality in the λΠ-calculus modulo, it is possible to *define* the equality predicate using Leibniz law.

$$\simeq : \iota \to \iota \to \mathsf{Type} := \lambda x : \iota. \ \lambda y : \iota. \ \Pi p : (\iota \to o). \ \mathsf{proof} \ (p \ x) \to \mathsf{proof} \ (p \ y)$$

Usual properties of equality can then be proved, so that we do not need to add them as axioms. For instance, reflexivity is proved by:

$$\mathsf{refl} : \Pi x : \iota. \ \simeq \ x \ x := \lambda x : \iota. \ \lambda p : \iota \to o. \ \lambda t : \mathsf{proof} \ (p \ x). \ t$$

Commutativity has the following proof:

$$\begin{aligned} \mathsf{comm} &: \Pi x : \iota. \ \Pi y : \iota. \ \simeq \ x \ y \to \ \simeq \ y \ x \\ &:= \lambda x : \iota. \ \lambda y : \iota. \ \lambda e : \simeq \ x \ y. \ \lambda p : \iota \to o. \ e \ (\lambda z : \iota. \ \dot{\Rightarrow} \ (p \ z) \ (p \ x)) \ (\lambda t : \mathsf{proof} \ (p \ x). \ t) \end{aligned}$$

3 Translating resolution and superposition proofs

3.1 Resolution

A derivation in resolution [17] tries to refute a set of clauses by inferring new clauses by means of the following two inference rules, until the empty clause is derived.

$$\textsf{Resolution} \ \frac{P; C \qquad \neg Q; D}{\sigma(C; D)} \ \sigma = mgu(P, Q) \qquad\qquad \textsf{Factoring} \ \frac{L; K; C}{\sigma(L; C)} \ \sigma = mgu(L, K)$$

To translate resolution proofs, we decompose these rules into two steps: one instantiation step and one propositional step:

$$\textsf{Instantiation}\ \frac{C}{\sigma(C)} \qquad \textsf{Identical Resolution}\ \frac{P; C \qquad \neg P; D}{C; D} \qquad \textsf{Identical Factoring}\ \frac{L; L; C}{L; C}$$

Of course these rules are applied modulo commutativity of ;, which means that P or L is not necessarily the first literal of the clauses.

Given some input clauses $C_1, \dots, C_k$, an identical-resolution derivation is a sequence of clauses $C_1, \dots, C_k, C_{k+1}, \dots, C_n$ such that each clauses C_i for $i > k$ is inferred from clauses among $C_1, \dots, C_{i-1}$ using one the three rules above. The input set of clauses is shown unsatisfiable if C_n is the empty clause. To translate such a derivation in the λΠ-calculus modulo, we first declare a constant c_i of type $||C_i||$ for each $1 \leq i \leq k$. Then, for each $k < j \leq n$, we define a constant c_j in terms of the previously declared or defined constants c_l where $1 \leq l < j$. The definitions depend on the rule used to infer C_j, and they use the constants corresponding to the clauses from which C_j is inferred. As mentioned above, definitions do not change the logical context of the proof. At the end, since all other constants are defined, the only axioms are $||C_i||$ for $1 \leq i \leq k$, and the translation of the empty clause, that is $\forall \flat.\ \textsf{proof}\ \flat$ is proved from these axioms. This shows that the set of input clauses is indeed refuted.

In contrast to other encodings of logical calculi in the λΠ-calculus, such as Pfenning sequent calculus [15] or some developments available on Logosphere (`http://www.logosphere.org/`), our embedding is shallow in the sense that a constant is not added for each inference rules, but resolution proofs are translated directly as terms of the λΠ-calculus modulo.

To understand the translation of the inference rules, one needs to look at the computational content of terms whose type is the translation of a clause $L_1; \cdots; L_m$: intuitively, they are functions that take as arguments n first-order terms to instantiate the free variables of the clause, a proposition $\flat$ to be proved, m functions that given a term of type $||L_i||$ return a proof of $\flat$, and that return a proof of $\flat$.

The translation of the instantiation rule is relatively easy, since one just needs to apply the image of the variable to the original clause, and to abstract over the new free variables:

$$\textsf{Instantiation}\ \frac{L_1; \cdots; L_m}{\sigma(L_1); \cdots; \sigma(L_m)}$$

$$\begin{aligned} c &: \Pi x_1 : \iota.\ \dots \Pi x_n : \iota.\ \Pi\ \flat : o.\ \llbracket L_1 \rrbracket_\flat \to \cdots \to \llbracket L_m \rrbracket_\flat \to \textsf{proof}\ \flat \\ d &: \Pi y_1 : \iota.\ \dots \Pi y_k : \iota.\ \Pi\ \flat : o.\ \llbracket \sigma(L_1) \rrbracket_\flat \to \cdots \to \llbracket \sigma(L_m) \rrbracket_\flat \to \textsf{proof}\ \flat \\ &:= \lambda y_1 : \iota.\ \dots \lambda y_k : \iota.\ c\ (\sigma(x_1))\ \cdots\ (\sigma(x_n)) \end{aligned}$$

The translation of factoring is also rather simple, since we just need to merge two literals:

$$\textsf{Identical Factoring}\ \frac{L_1; \cdots; L_i; L_i; \cdots; L_m}{L_1; \cdots; L_i; \cdots; L_m}$$

$$\begin{aligned} c &: \Pi x_1 : \iota.\ \dots \Pi x_n : \iota.\ \Pi\ \flat : o.\ \llbracket L_1 \rrbracket_\flat \to \cdots \to \llbracket L_i \rrbracket_\flat \to \llbracket L_i \rrbracket_\flat \to \cdots \to \llbracket L_m \rrbracket_\flat \to \textsf{proof}\ \flat \\ d &: \Pi x_1 : \iota.\ \dots \Pi x_n : \iota.\ \Pi\ \flat : o.\ \llbracket L_1 \rrbracket_\flat \to \cdots \to \llbracket L_i \rrbracket_\flat \to \cdots \to \llbracket L_m \rrbracket_\flat \to \textsf{proof}\ \flat \\ &:= \lambda x_1 : \iota.\ \dots \lambda x_n : \iota.\ \lambda \flat : o.\ \lambda l_1 : \llbracket L_1 \rrbracket_\flat.\ \cdots\ \lambda l_m : \llbracket L_m \rrbracket_\flat.\ c\ x_1\ \cdots\ x_n\ \flat\ l_1\ \cdots\ l_i\ l_i\ \cdots\ l_m \end{aligned}$$

To translate a resolution step, we can use the atom P and its negation to get the proof of $\flat$. More precisely, we can use as term of type $\llbracket P \rrbracket_\flat = ||P|| \to \textsf{proof}\ \flat$ in the translation of the clause $L_1; \cdots; P; \cdots; L_m$ the function that take a term tp of type $||P||$ and that returns the clause $M_1; \cdots; \neg P \cdots; M_l$ where the term for type $\llbracket \neg P \rrbracket_\flat = (||P|| \to \textsf{proof}\ \flat) \to \textsf{proof}\ \flat$ is the function that take a term tnp of type $||P|| \to \textsf{proof}\ \flat$ and return $tnp\ tp$, which is of type $\textsf{proof}\ \flat$.

$$\textsf{Identical Resolution}\ \frac{L_1; \cdots; L_{i-1}; P; L_i; \cdots; L_m \qquad M_1; \cdots; M_{h-1}; \neg P; M_h; \cdots; M_l}{L_1; \cdots; L_m; M_1; \cdots; M_l}$$

$$\begin{aligned}
c1 &: \Pi x_1:\iota.\ \ldots \Pi x_n:\iota.\ \Pi\,\flat:o.\ \llbracket L_1\rrbracket_\flat \to \cdots \to \llbracket P\rrbracket_\flat \to \cdots \to \llbracket L_m\rrbracket_\flat \to \mathsf{proof}\ \flat \\
c2 &: \Pi y_1:\iota.\ \ldots \Pi y_k:\iota.\ \Pi\,\flat:o.\ \llbracket M_1\rrbracket_\flat \to \cdots \to \llbracket \neg P\rrbracket_\flat \to \cdots \to \llbracket M_l\rrbracket_\flat \to \mathsf{proof}\ \flat \\
d &: \Pi z_1:\iota.\ \ldots \Pi z_j:\iota.\ \Pi\,\flat:o.\ \llbracket L_1\rrbracket_\flat \to \cdots \to \llbracket L_m\rrbracket_\flat \to \llbracket M_1\rrbracket_\flat \to \cdots \to \llbracket M_l\rrbracket_\flat \to \mathsf{proof}\ \flat \\
&:= \lambda z_1:\iota.\ \ldots \lambda z_j:\iota.\ \lambda\flat:o.\ \lambda l_1:\llbracket L_1\rrbracket_\flat.\ \cdots\ \lambda l_m:\llbracket L_m\rrbracket_\flat. \\
&\qquad\qquad\qquad\qquad\qquad\quad \lambda m_1:\llbracket M_1\rrbracket_\flat.\ \cdots\ \lambda m_l:\llbracket M_l\rrbracket_\flat. \\
&\qquad\qquad\qquad\qquad\qquad\quad c1\ x_1\ \cdots\ x_n\ \flat\ l_1\ \cdots\ l_{i-1} \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda tp:||P||.\ c2\ y_1\ \cdots\ y_k\ \flat\ m_1\ \cdots\ m_{h-1} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (\lambda tnp:(||P||\to\mathsf{proof}\ \flat).\ tnp\ tp)\ m_h\ \cdots\ m_l)\ l_i\ \cdots\ l_m
\end{aligned}$$

Example 1. We want to refute the set of two clauses $p(X,Y);p(X,a)$ and $\neg p(b,Y)$. A possible derivation of the empty clause in resolution is the following:

1	$p(X,Y);p(X,a)$	
2	$\neg p(b,Y)$	
3	$p(X,a)$	applying Factoring on 1
4	$\Box$	applying Resolution on 2 and 3

If we decompose the instantiations from the inferences, we get

1	$p(X,Y);p(X,a)$	
2	$\neg p(b,Y)$	
3	$p(X,a);p(X,a)$	applying Instantiation on 1 with $\sigma=\{Y\mapsto a\}$
4	$p(X,a)$	applying Identical Factoring on 3
5	$p(b,a)$	applying Instantiation on 4 with $\sigma=\{X\mapsto b\}$
6	$\neg p(b,a)$	applying Instantiation on 2 with $\sigma=\{Y\mapsto a\}$
7	$\Box$	applying Identical Resolution on 5 and 6

We have a binary predicate symbol p and two constants a and b. The context of the translation in the λΠ-calculus modulo is therefore

$$\begin{aligned}
&\iota:\mathsf{Type} \\
&o:\mathsf{Type} \\
&\mathsf{proof}:o\to\mathsf{Type} \\
&\dot{p}:\iota\to\iota\to o \\
&p:\iota\to\iota\to\mathsf{Type} \\
&\mathsf{proof}\ (\dot{p}\ x\ y)\hookrightarrow p\ x\ y \\
&a:\iota \\
&b:\iota
\end{aligned}$$

We first declare the two input clauses:

$$\begin{aligned}
c1 &: \Pi X:\iota.\ \Pi Y:\iota.\ \Pi\,\flat:o.\ (p\ X\ Y\to\mathsf{proof}\ \flat)\to(p\ X\ a\to\mathsf{proof}\ \flat)\to\mathsf{proof}\ \flat \\
c2 &: \Pi Y:\iota.\ \Pi\,\flat:o.\ ((p\ b\ Y\to\mathsf{proof}\ \flat)\to\mathsf{proof}\ \flat)\to\mathsf{proof}\ \flat
\end{aligned}$$

We then declare the inferred clauses and define them as explained above:

$$\begin{aligned}
c3 &: \Pi X:\iota.\ \Pi\,\flat:o.\ (p\ X\ a\to\mathsf{proof}\ \flat)\to(p\ X\ a\to\mathsf{proof}\ \flat)\to\mathsf{proof}\ \flat := \lambda X:\iota.\ c1\ X\ a \\
c4 &: \Pi X:\iota.\ \Pi\,\flat:o.\ (p\ X\ a\to\mathsf{proof}\ \flat)\to\mathsf{proof}\ \flat
\end{aligned}$$

$$
\begin{aligned}
&\quad := \lambda X : \iota.\ \lambda \flat : o.\ \lambda l : (p\ X\ a \to \mathsf{proof}\ \flat).\ c3\ X\ \flat\ l\ l\\
&c5 : \Pi\ \flat : o.\ (p\ b\ a \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat := c4\ b\\
&c6 : \Pi\ \flat : o.\ ((p\ b\ a \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat := c2\ a\\
&c7 : \Pi\ \flat : o.\ \mathsf{proof}\ \flat := \lambda \flat : o.\ c5\ \flat\ (\lambda tp : p\ b\ a.\ c6\ \flat\ (\lambda tnp : p\ b\ a \to \mathsf{proof}\ \flat.\ tnp\ tp))
\end{aligned}
$$

3.2 Superposition

Superposition can be seen as an extension of resolution to handle equality better. Superposition primarily uses four inference rules ($u \not\simeq v$ denotes $\neg(u \simeq v)$):

$$\textsf{Equality Resolution}\ \frac{u \not\simeq v; R}{\sigma(R)}\ \sigma = mgu(u, v) \qquad \textsf{Negative Superposition}\ \frac{s \simeq t; S \qquad u \not\simeq v; R}{\sigma(u[t]_{\mathfrak{p}} \not\simeq v; S; R)}\ ^{a}$$

$$\textsf{Positive Superposition}\ \frac{s \simeq t; S \qquad u \simeq v; R}{\sigma(u[t]_{\mathfrak{p}} \simeq v; S; R)}\ ^{a} \qquad \textsf{Equality Factoring}\ \frac{s \simeq t; u \simeq v; R}{\sigma(t \not\simeq v; u \simeq v; R)}\ \sigma = mgu(s, u)$$

[a] $\sigma = mgu(u_{|\mathfrak{p}}, s)$

These rules are given with many conditions that restrict the cases when they can be applied. That makes the superposition calculus usable in practice in contrast to former paramodulation-based methods. Since we are only concerned in translating a proof, not finding one, these restrictions do not concern us.

Also, superposition-based provers use simplification rules, in which a set of clauses is replaced by another set of clauses. This too is not problematic for us since these simplification rules can in most of the cases be decomposed into the application of the four basic inference rules followed by the elimination of redundant clauses. Notable exceptions are the rules introducing and applying definitions in for instance the prover E, that we will not consider here.

Here again, to ease the translation, we will consider an explicit instantiation step and propositional rules:

$$\textsf{Identical Equality Resolution}\ \frac{u \not\simeq u; R}{R} \qquad \textsf{Negative Replacement}\ \frac{s \simeq t; S \qquad u[s]_{\mathfrak{p}} \not\simeq v; R}{u[t]_{\mathfrak{p}} \not\simeq v; S; R}$$

$$\textsf{Positive Replacement}\ \frac{s \simeq t; S \qquad u[s]_{\mathfrak{p}} \simeq v; R}{u[t]_{\mathfrak{p}} \simeq v; S; R} \qquad \textsf{Identical Equality Factoring}\ \frac{s \simeq t; s \simeq v; R}{t \not\simeq v; s \simeq v; R}$$

Once more, these rules can be applied modulo commutativity of ; and $\simeq$. For $\simeq$, it can be taken into account using the comm term (see Section 2.2). For simplicity, we assume in the following that equalities are oriented appropriately.

Since reflexivity is provable thanks to our encoding of equality, Identical Equality Resolution is rather easy to translate. Indeed, a term of type $[\![u \not\simeq u]\!]_\flat = (\simeq\ \ u\ u \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$ can be $\lambda p : (||u \simeq u|| \to \mathsf{proof}\ \flat).\ p\ (\mathsf{refl}\ u)$.

$$\textsf{Identical Equality Resolution}\ \frac{L_1; \cdots ; L_{i-1}; u \not\simeq u; L_i \cdots ; L_m}{L_1; \cdots ; L_m}$$

$$
\begin{aligned}
&c : \Pi x_1 : \iota.\ \ldots \Pi x_n : \iota.\ \Pi\ \flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![u \not\simeq u]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof}\ \flat\\
&d : \Pi y_1 : \iota.\ \ldots \Pi y_k : \iota.\ \Pi\ \flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof}\ \flat\\
&\quad := \lambda y_1 : \iota.\ \ldots \lambda y_k : \iota.\ \lambda \flat : o.\ \lambda l_1 : [\![L_1]\!]_\flat.\ \cdots\ \lambda l_m : [\![L_m]\!]_\flat.\\
&\qquad\qquad c\ x_1\ \ldots\ x_n\ \flat\ l_1\ \ldots\ l_{i-1}\ (\lambda p : (||u \simeq u|| \to \mathsf{proof}\ \flat).\ p\ (\mathsf{refl}\ u))\ l_i\ \ldots\ l_m
\end{aligned}
$$

For Identical Equality Factoring, we somehow need to refute $s \simeq t$ from $s \simeq v$ and $t \not\simeq v$. If we consider a term p of type $[\![t \not\simeq v]\!]_\flat = (\simeq\ \ t\ v \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$, a term q of type $\llbracket s \simeq v \rrbracket_\flat =\simeq\ \ s\ v \to \mathsf{proof}\ \flat$ and a term r of type $||s \simeq t|| =\simeq\ \ s\ t$, the term $p\ (r\ (\lambda z : \iota. \dot{\Rightarrow}\ (\dot{\simeq}\ z\ v)\ \flat)\ q)$ has type $\mathsf{proof}\ \flat$.

$$\textsf{Identical Equality Factoring}\ \frac{L_1; \cdots ; L_{h-1}; s \simeq t; L_h; \cdots ; L_{i-1}; s \simeq v; L_i; \cdots ; L_m}{t \not\simeq v; s \simeq v; L_1; \cdots ; L_m}$$

$$\begin{aligned}
c &: \Pi x_1 : \iota \ldots \Pi x_n : \iota.\ \Pi\ \flat : o.\ \llbracket L_1 \rrbracket_\flat \to \cdots \to \llbracket s \simeq t \rrbracket_\flat \to \cdots \to \llbracket s \simeq v \rrbracket_\flat \to \cdots \to \llbracket L_m \rrbracket_\flat \to \mathsf{proof}\ \flat \\
d &: \Pi y_1 : \iota \ldots \Pi y_k : \iota.\ \Pi\ \flat : o.\ \llbracket t \not\simeq v \rrbracket_\flat \to \llbracket s \simeq v \rrbracket_\flat \to \llbracket L_1 \rrbracket_\flat \to \cdots \to \llbracket L_m \rrbracket_\flat \to \mathsf{proof}\ \flat \\
&:= \lambda y_1 : \iota.\ \ldots \lambda y_k : \iota.\ \lambda \flat : o.\ \lambda p : \llbracket t \not\simeq v \rrbracket_\flat.\ \lambda q : \llbracket s \simeq v \rrbracket_\flat.\ \lambda l_1 : \llbracket L_1 \rrbracket_\flat.\ \cdots\ \lambda l_m : \llbracket L_m \rrbracket_\flat. \\
&\quad\ c\ x_1\ \ldots\ x_n\ \flat\ l_1\ \ldots\ l_{h-1}\ (\lambda r : ||s \simeq t||.\ p\ (r\ (\lambda z : \iota. \dot{\Rightarrow}\ (\dot{\simeq}\ z\ v)\ \flat)\ q))\ l_h \ldots l_{i-1}\ q\ l_i \ldots l_m
\end{aligned}$$

For Positive Replacement, we can use the following idea: given a term p of type $\llbracket u[t]_\mathfrak{p} \simeq v \rrbracket_\flat = ||u[t]_\mathfrak{p} \simeq v|| \to \mathsf{proof}\ \flat$, a term q of type $||u[s]_\mathfrak{p} \simeq v||$ and a term r of type $||s \simeq t||$, the term $p\ (r\ (\lambda z.\ \dot{\simeq}\ |u[z]_\mathfrak{p}|\ |v|)\ q)$ has type $\mathsf{proof}\ \flat$.

$$\textsf{Positive Replacement}\ \frac{L_1; \cdots ; L_{i-1}; s \simeq t; L_i; \cdots ; L_m \qquad M_1; \cdots ; M_{h-1}; u[s]_\mathfrak{p} \simeq v; M_h; \cdots ; M_l}{u[t]_\mathfrak{p} \simeq v; L_1; \cdots ; L_m; M_1; \cdots ; M_l}$$

$$\begin{aligned}
c1 &: \Pi x_1 : \iota \ldots \Pi x_n : \iota.\ \Pi\ \flat : o.\ \llbracket L_1 \rrbracket_\flat \to \cdots \to \llbracket s \simeq t \rrbracket_\flat \to \cdots \to \llbracket L_m \rrbracket_\flat \to \mathsf{proof}\ \flat \\
c2 &: \Pi y_1 : \iota \ldots \Pi y_k : \iota.\ \Pi\ \flat : o.\ \llbracket M_1 \rrbracket_\flat \to \cdots \to \llbracket u[s]_\mathfrak{p} \simeq v \rrbracket_\flat \to \cdots \to \llbracket M_l \rrbracket_\flat \to \mathsf{proof}\ \flat \\
d &: \Pi z_1 : \iota \ldots \Pi z_j : \iota.\ \Pi\ \flat : o.\ \llbracket u[t]_\mathfrak{p} \simeq v \rrbracket_\flat \to \llbracket L_1 \rrbracket_\flat \to \cdots \llbracket L_m \rrbracket_\flat \to \llbracket M_1 \rrbracket_\flat \to \cdots \llbracket M_l \rrbracket_\flat \to \mathsf{proof}\ \flat \\
&:= \lambda z_1 : \iota.\ \ldots \lambda z_j : \iota.\ \lambda \flat : o.\ \lambda p : \llbracket u[t]_\mathfrak{p} \simeq v \rrbracket_\flat.\ \lambda l_1 : \llbracket L_1 \rrbracket_\flat.\ \cdots\ \lambda l_m : \llbracket L_m \rrbracket_\flat. \\
&\quad\ \lambda m_1 : \llbracket M_1 \rrbracket_\flat.\ \cdots\ \lambda m_l : \llbracket M_l \rrbracket_\flat.c2\ y_1\ \cdots\ y_k\ \flat\ m_1\ \cdots\ m_{h-1}\ (\lambda q : ||u[s]_\mathfrak{p} \simeq v||. \\
&\quad\quad c1\ x_1 \cdots x_n\ \flat\ l_1 \cdots l_{i-1}\ (\lambda r : ||s \simeq t||.\ p\ (r\ (\lambda z.\ \dot{\simeq}\ |u[z]_\mathfrak{p}|\ |v|)\ q))\ l_i\ \cdots\ l_m)\ m_h\ \cdots\ m_l
\end{aligned}$$

Negative Replacement is almost the same, except that p has type $\llbracket u[t]_\mathfrak{p} \not\simeq v \rrbracket_\flat$ instead of $\llbracket u[t]_\mathfrak{p} \simeq v \rrbracket_\flat$ and q has type $||u[s]_\mathfrak{p} \simeq v|| \to \mathsf{proof}\ \flat$ instead of $||u[s]_\mathfrak{p} \simeq v||$, so that the term $p\ (r\ (\lambda z. \dot{\Rightarrow}\ (\dot{\simeq}\ |u[z]_\mathfrak{p}|\ |v|)\ \flat)\ q)$ has type $\mathsf{proof}\ \flat$.

$$\textsf{Negative Replacement}\ \frac{L_1; \cdots ; L_{i-1}; s \simeq t; L_i; \cdots ; L_m \qquad M_1; \cdots ; M_{h-1}; u[s]_\mathfrak{p} \not\simeq v; M_h; \cdots ; M_l}{u[t]_\mathfrak{p} \not\simeq v; L_1; \cdots ; L_m; M_1; \cdots ; M_l}$$

$$\begin{aligned}
c1 &: \Pi x_1 : \iota \ldots \Pi x_n : \iota.\ \Pi\ \flat : o.\ \llbracket L_1 \rrbracket_\flat \to \cdots \to \llbracket s \simeq t \rrbracket_\flat \to \cdots \to \llbracket L_m \rrbracket_\flat \to \mathsf{proof}\ \flat \\
c2 &: \Pi y_1 : \iota \ldots \Pi y_k : \iota.\ \Pi\ \flat : o.\ \llbracket M_1 \rrbracket_\flat \to \cdots \to \llbracket u[s]_\mathfrak{p} \not\simeq v \rrbracket_\flat \to \cdots \to \llbracket M_l \rrbracket_\flat \to \mathsf{proof}\ \flat \\
d &: \Pi z_1 : \iota \ldots \Pi z_j : \iota.\ \Pi\ \flat : o.\ \llbracket u[t]_\mathfrak{p} \not\simeq v \rrbracket_\flat \to \llbracket L_1 \rrbracket_\flat \to \cdots \llbracket L_m \rrbracket_\flat \to \llbracket M_1 \rrbracket_\flat \to \cdots \llbracket M_l \rrbracket_\flat \to \mathsf{proof}\ \flat \\
&:= \lambda z_1 : \iota.\ \ldots \lambda z_j : \iota.\ \lambda \flat : o.\ \lambda p : \llbracket u[t]_\mathfrak{p} \not\simeq v \rrbracket_\flat. \\
&\quad\ \lambda l_1 : \llbracket L_1 \rrbracket_\flat.\ \cdots\ \lambda l_m : \llbracket L_m \rrbracket_\flat.\lambda m_1 : \llbracket M_1 \rrbracket_\flat.\ \cdots\ \lambda m_l : \llbracket M_l \rrbracket_\flat. \\
&\quad\ c2\ y_1\ \cdots\ y_k\ \flat\ m_1\ \cdots\ m_{h-1}\ (\lambda q : (||u[s]_\mathfrak{p} \simeq v|| \to \mathsf{proof}\ \flat).\ c1\ x_1\ \cdots\ x_n\ \flat\ l_1\ \cdots\ l_{i-1} \\
&\quad\quad (\lambda r : ||s \simeq t||.\ p\ (r\ (\lambda z. \dot{\Rightarrow}\ (\dot{\simeq}\ |u[z]_\mathfrak{p}|\ |v|)\ \flat)\ q))\ l_i\ \cdots\ l_m)\ m_h\ \cdots\ m_l
\end{aligned}$$

Note that the shallowness of the translation of the equality predicate is heavily used in the translation of inference rules.

Example 2. We want to refute the three clauses $c \simeq g(a); X \simeq f(b, Y)$ and $g(Z) \simeq f(X, Z)$ and $g(c) \not\simeq g(f(X, Y))$. A possible derivation of the empty clause in superposition (without considering ordering restrictions) is the following:

1	$c \simeq g(a); X \simeq f(b,Y)$	
2	$g(X) \simeq f(Z,X)$	
3	$g(c) \not\simeq g(f(X,Y))$	
4	$c \simeq f(Z,a); X \simeq f(b,Y)$	applying Positive Superposition on 2 and 1
5	$f(Z,a) \not\simeq f(b,Y); c \simeq f(b,Y)$	applying Equality Factoring on 4
6	$c \simeq f(b,a)$	applying Equality Resolution on 5
7	$g(f(b,a)) \not\simeq g(f(X,Y))$	applying Negative Superposition on 6 and 3
8	$\Box$	applying Equality Resolution on 7

If we decompose the instantiations from the inferences, we get

1	$c \simeq g(a); X \simeq f(b,Y)$	
2	$g(X) \simeq f(Z,X)$	
3	$g(c) \not\simeq g(f(X,Y))$	
4	$g(a) \simeq f(Z,a)$	applying Instantiation on 2 with $\sigma = \{X \mapsto a\}$
5	$c \simeq f(Z,a); X \simeq f(b,Y)$	applying Positive Replacement on 4 and 1
6	$c \simeq f(Z,a); c \simeq f(b,Y)$	applying Instantiation on 5 with $\sigma = \{X \mapsto c\}$
7	$f(Z,a) \not\simeq f(b,Y); c \simeq f(b,Y)$	applying Identical Equality Factoring on 6
8	$f(b,a) \not\simeq f(b,a); c \simeq f(b,a)$	applying Instantiation on 7 with $\sigma = \{Y \mapsto a; Z \mapsto b\}$
9	$c \simeq f(b,a)$	applying Identical Equality Resolution on 8
10	$g(f(b,a)) \not\simeq g(f(X,Y))$	applying Negative Replacement on 9 and 3
11	$g(f(b,a)) \not\simeq g(f(b,a))$	applying Instantiation on 10 with $\sigma = \{X \mapsto b; Y \mapsto a\}$
12	$\Box$	applying Identical Equality Resolution on 11

We have a unary function symbol g, a binary function symbol f and three constants a, b and c. The context of the translation in the λΠ-calculus modulo is therefore

$$\iota : \mathsf{Type}$$
$$o : \mathsf{Type}$$
$$\mathsf{proof} : o \to \mathsf{Type}$$
$$\dot{\simeq} : \iota \to \iota \to o$$
$$\simeq : \iota \to \iota \to \mathsf{Type} := \lambda x : \iota.\ \lambda y : \iota.\ \Pi p : (\iota \to o).\ \mathsf{proof}\ (p\ x) \to \mathsf{proof}\ (p\ y)$$
$$\dot{\Rightarrow} : o \to o \to o$$
$$\mathsf{proof}\ (\dot{\simeq}\ x\ y) \hookrightarrow \simeq\ x\ y$$
$$\mathsf{proof}\ (\dot{\Rightarrow}\ A\ B) \hookrightarrow \mathsf{proof}\ A \to \mathsf{proof}\ B$$
$$\mathsf{refl} : \Pi x : \iota.\ \simeq\ x\ x := \lambda x : \iota.\ \lambda p : \iota \to o.\ \lambda t : \mathsf{proof}\ (p\ x).\ t$$
$$g : \iota \to \iota$$
$$f : \iota \to \iota \to \iota$$
$$a : \iota$$
$$b : \iota$$
$$c : \iota$$

We first declare the three input clauses:

$$c1 : \Pi X : \iota.\ \Pi Y : \iota.\ \Pi \flat : o.\ (\simeq\ c\ (g\ a) \to \mathsf{proof}\ \flat) \to (\simeq\ X\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$c2 : \Pi X : \iota.\ \Pi Z : \iota.\ \Pi \flat : o.\ (\simeq\ (g\ X)\ (f\ Z\ X) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$c3 : \Pi X : \iota.\ \Pi Y : \iota.\ \Pi \flat : o.\ ((\simeq\ (g\ c)\ (g\ (f\ X\ Y)) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$

We then declare the inferred clauses and define them as explained above:

$$\begin{array}{l}
c4 : \Pi Z : \iota.\ \Pi\, \flat : o.\ (\simeq\ (g\ a)\ (f\ Z\ a) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat := \lambda Z : \iota.\ c2\ a\ Z \\
c5 : \Pi X : \iota.\Pi Y : \iota.\Pi Z : \iota.\Pi\, \flat : o.(\simeq c\ (f\ Z\ a) \to \mathsf{proof}\ \flat) \to (\simeq X\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat \\
\quad := \lambda X : \iota.\lambda Y : \iota.\lambda Z : \iota.\lambda \flat : o.\lambda p : (\simeq c\ (f\ Z\ a) \to \mathsf{proof}\ \flat).\ \lambda l : (\simeq X\ (f\ b\ Y) \to \mathsf{proof}\ \flat). \\
\qquad\quad c1\ X\ Y\ \flat\ (\lambda q :\simeq\ c\ (g\ a).\ c4\ Z\ \flat\ (\lambda r :\simeq\ (g\ a)\ (f\ Z\ a).\ p\ (r\ (\lambda u : \iota.\ \dot{\simeq}\ c\ u)\ q)))\ l \\
c6 : \Pi Y : \iota.\ \Pi Z : \iota.\ \Pi\, \flat : o.\ (\simeq\ c\ (f\ Z\ a) \to \mathsf{proof}\ \flat) \to (\simeq\ c\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat \\
\quad := \lambda Y : \iota.\ \lambda Z : \iota.\ c5\ c\ Y\ Z \\
c7 : \Pi Y : \iota.\Pi Z : \iota.\Pi\, \flat : o.((\simeq (f\ Z\ a)\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to (\simeq c\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \\
\quad \to \mathsf{proof}\ \flat\ :=\ \lambda Y : \iota.\ \lambda Z : \iota.\ \lambda \flat : o.\ \lambda p : ((\simeq\ (f\ Z\ a)\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat). \\
\quad\ \lambda q : (\simeq\ c\ (f\ b\ Y) \to \mathsf{proof}\ \flat). \\
\qquad\quad c6\ Y\ Z\ \flat\ (\lambda r :\simeq\ c\ (f\ Z\ a).\ p\ (r\ (\lambda u : \iota.\ \dot{\Rightarrow}\ (\dot{\simeq}\ u\ (f\ b\ Y))\ \flat)\ q))\ q \\
c8 : \Pi\, \flat : o.\ ((\simeq\ (f\ b\ a)\ (f\ b\ a) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to (\simeq\ c\ (f\ b\ a) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat \\
\quad := c7\ a\ b \\
c9 : \Pi\, \flat : o.\ (\simeq\ c\ (f\ b\ a) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat\ :=\ \lambda \flat : o.\ \lambda l : (\simeq\ c\ (f\ b\ a) \to \mathsf{proof}\ \flat). \\
\qquad c8\ \flat\ (\lambda p : (\simeq (f\ b\ a)\ (f\ b\ a) \to \mathsf{proof}\ \flat).\ p\ (\mathsf{refl}\ (f\ b\ a)))\ l \\
c10 : \Pi X : \iota.\ \Pi Y : \iota.\ \Pi\, \flat : o.\ ((\simeq\ (g\ (f\ b\ a))\ (g\ (f\ X\ Y)) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat \\
\quad :=\ \lambda X : \iota.\ \lambda Y : \iota.\ \lambda \flat : o.\ \lambda p : ((\simeq\ (g\ (f\ b\ a))\ (g\ (f\ X\ Y)) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat). \\
\qquad\quad c3\ X\ Y\ \flat\ (\ \lambda q : (\simeq\ (g\ c)\ (g\ (f\ X\ Y)) \to \mathsf{proof}\ \flat). \\
\qquad\qquad\qquad\qquad c9\ \flat\ (\lambda r :\simeq\ c\ (f\ b\ a).\ p\ (r\ (\lambda u.\ \dot{\Rightarrow}\ (\dot{\simeq}\ (g\ u)\ (g\ (f\ X\ Y)))\ \flat)\ q))) \\
c11 : \Pi\, \flat : o.\ ((\simeq\ (g\ (f\ b\ a))\ (g\ (f\ b\ a)) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat := c10\ b\ a \\
c12 : \Pi\, \flat : o.\ \mathsf{proof}\ \flat \\
\quad := \lambda \flat : o.\ c11\ \flat\ (\lambda p : (\simeq\ (g\ (f\ b\ a))\ (g\ (f\ b\ a)) \to \mathsf{proof}\ \flat).\ p\ (\mathsf{refl}\ (g\ (f\ b\ a))))
\end{array}$$

3.3 Resolution Proofs Are Constructive Proofs

In the translation of resolution and superposition proofs above, we do not need any axiom for classical logic, which means that we have an intuitionistic proof. Furthermore, since the translation of a clause is intuitionistically implied by the translation of its corresponding formula, that means that the proof of unsatisfiability of a set of clauses by the resolution method is intuitionistic. However, the resolution method is in general used to refute the negation of a formula: to prove A, one proves that the clausal normal form of $\neg A$ is unsatisfiable. To go to the proof of unsatisfiability of $\neg A$ to a proof of A, one needs a classical axiom (even without considering the clausification of $\neg A$).

This remark about constructiveness of resolution proofs is not so surprising. Indeed, given the clauses $C_1, \dots, C_m$ with correspond formulas $A_1, \dots A_m$, proving the unsatisfiability of $C_1, \dots, C_m$ amounts to proving the sequent $A_1, \dots, A_m \vdash$ in the sequent calculus. But for this particular class of sequents, intuitionistic and classical logics coincide. Indeed, since there are only atomic formulas under negations, and there are no implications, there can only be atomic formulas in the right-hand side of sequents in a proof of $A_1, \dots, A_m \vdash$. Since only one of them can be used in each axiom rule closing a branch of the proof, we can restrict ourselves to sequents containing at most one formula in the right-hand side, as in the intuitionistic fragment.

4 Implementation in iProver Modulo

We have successfully implemented the technique above in iProver Modulo. iProver [13] is a prover for first-order logic based the combination of two proof-search methods, namely instantiation-generation and resolution. iProver Modulo [6] is a patch to iProver to integrate Polarized Resolution Modulo [9]. iProver Modulo is available at `http://www.ensiie.fr/~guillaume.burel/blackandwhite_iProverModulo.html.en`. When iProver Modulo finds a pure resolution proof (for instance, when the instantiation-generation method is switched off), we are able to translate it to the $\lambda\Pi$-calculus modulo using the technique presented in this paper.

As said above, resolution- and superposition-based provers do not only use the inference rules presented above, but uses also simplification rules that can be used to replace a set of clause by another. For instance, iProver Modulo uses

$$\text{Subsumption Resolution}\ \frac{L;C \qquad \sigma(\overline{L});\sigma(C);D}{L;C \qquad \sigma(C);D}$$

where $\overline{P} = \neg P$ and $\overline{\neg P} = P$. After the simplification is performed, $\sigma(\overline{L});\sigma(C);D$ is no longer in the working space of the prover to search for a proof, but it can be used to translate a proof once it has been found. It is possible to infer the clause $\sigma(C);D$ but using the derivation

$$\text{Identical Factoring}\ \cfrac{\text{Identical Resolution}\ \cfrac{\text{Instantiation}\ \cfrac{L;C}{\sigma(L);\sigma(C)} \qquad \sigma(\overline{L});\sigma(C);D}{\sigma(C);\sigma(C);D}}{\sigma(C);D}\ *$$

and this derivation can be translated as usual.

In practice, when iProver Modulo is run with the option `--dedukti-out-proof true`, if a proof using only the resolution method is found, it is output in Dedukti's syntax (in which $\lambda x : a.\ t$ and $\Pi x : a.\ b$ are respectively written `x : a => t` and `x : a -> b`) and can be checked by the dkparse tool. For instance, for the unsatisfiability of the two clauses in Example 1, iProver Modulo outputs:

```
o : Type.
proof : (o -> Type).
i : Type.
a : i.
b : i.
p : (i -> (i -> Type)).
clause3 : X1 : i -> X0 : i -> bot_var : o -> (p X0 a -> proof bot_var)
  -> (p X0 X1 -> proof bot_var) -> proof bot_var.
clause2 : X0 : i -> bot_var : o -> (p X0 a -> proof bot_var) -> proof bot_var
  := X0 : i => bot_var : o => lit1 : (p X0 a -> proof bot_var)
     => clause3 a X0 bot_var lit1 lit1.
clause4 : X0 : i -> bot_var : o ->
  ((p b X0 -> proof bot_var) -> proof bot_var) -> proof bot_var.
clause1 : bot_var : o -> proof bot_var
  := bot_var : o => clause2 b bot_var (tp : p b a =>
    clause4 a bot_var (tnp : (p b a -> proof bot_var) => tnp tp)).
```

The input clauses are `clause3` and `clause4`, and the false formula $\Pi\flat : o.\ \mathsf{proof}\ \flat$ is proved by `clause1`. Note that contrarily to what is detailed above to ease the comprehension, instantiations are integrated in the inference rules: `clause2` is inferred from `clause3` by Factoring (with

$\sigma = \{\mathtt{X1} \mapsto \mathtt{a}\}$), and `clause1` from `clause2` and `clause4` by Resolution (with σ mapping the `X0` of `clause2` to `b` and the `X0` of `clause4` to `a`).

Note that in iProver Modulo, clauses can be normalized w.r.t. a term rewriting system given as input as part of the theory in which the proof is searched for. To handle this in the translation to Dedukti, one just need to add term rewriting rules in the context, the translation of the proof remaining unchanged. Indeed, proof checking will normalize clauses appropriately (provided the rewriting system is convergent).

Conclusion

We have presented a shallow embedding of the proofs found by state-of-art first-order automated theorem provers into the λΠ-calculus modulo (however without the transformation in clausal normal form). We also have described its implementation in iProver Modulo. This work is a first step towards the interoperability of automated theorem provers and proof assistants. We can now envisage to combine proofs coming from Coq, HOL, and iProver Modulo, by linking their translations in Dedukti. To that purpose, as explained in the introduction, the fact that the embeddings are shallow will be extremely useful. We now consider further works.

Note that we do not claim any adequacy theorem, in the sense that we could relate a proof in the λΠ-calculus modulo to a resolution or superposition proof. We only claim the correctness of the translation. Since we have a shallow embedding, the only adequacy that we need is that of the translation of first-order logic. If a malicious user changes a proof, and it is checked by dkparse, it will still be a valid proof (but perhaps not of the same theorem) even if it does not correspond to a resolution proof.

An implementation of the translation of superposition proofs would let us see if such an embedding can really be used in practice. A good candidate for integrating this translation is Zipperposition, a first-order theorem prover based on superposition, written in OCaml and developed as a experimental platform to test ideas around the superposition calculus. Zipperposition is available at `https://www.rocq.inria.fr/deducteam/Zipperposition/`.

Moreover, first-order theorem provers generally do not take as inputs only set of clauses to be proved unsatisfiable, but they also can handle full first-order formulas. To be able to translates these proofs into the λΠ-calculus modulo, we should be able to express in the λΠ-calculus modulo the transformation of formulas into clausal normal form. This raises two issues: first, some transformations need classical logic. To handle them, a possibility is to add a classical axiom, for instance $nnpp : \Pi p : o.\ \Pi \flat : o.\ ((\mathsf{proof}\ p \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ p$. A more difficult point is that for some transformations, the resulting set of formulas is not logically equivalent to the first one, but is only equisatisfiable. This is the case for instance of the elimination of an existential quantifier using a Skolem symbol. To solve this, one should probably transform the proof back to reintegrate the existential variables introduced by Skolemization.

A remaining challenge is to be able to obtain Dedukti proof from other automated theorem provers than iProver Modulo. Instead of implementing the idea of this paper to other provers, a solution could be to use iProver Modulo to output a Dedukti proof for each inference step of a proof found by another prover, as could be described in the TSTP format [19], supported by many provers nowadays. Then, by recombining each of these steps, we would obtain a whole proof of the original formula, at least if only inference rules that are really logical implications are used. Nevertheless, this is not immediate, because for the moment we only translate proofs of unsatisfiability of set of clauses, and the combination of such proofs would require to link clauses with the clausal normal form of their negation: a proof that C_1 and C_2 leads to C_3 will indeed be a proof that C_1, C_2 and the clausal normal form of $\neg C_3$ is unsatisfiable.

References

[1] Ali Assaf and Guillaume Burel. Translating HOL to Dedukti. Submitted, 2013.

[2] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):1–31, 1994.

[3] Jasmin Christian Blanchette, Sascha Bhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 116–130. Springer, 2011.

[4] Mathieu Boespflug and Guillaume Burel. CoqInE: Translating the calculus of inductive constructions into the λΠ-calculus modulo. In David Pichardie and Tjark Weber, editors, *Second International Workshop on Proof Exchange for Theorem Proving*, 2012.

[5] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *Second International Workshop on Proof Exchange for Theorem Proving*, 2012.

[6] Guillaume Burel. Experimenting with deduction modulo. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 162–176. Springer, 2011.

[7] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[8] Alexis Dorra. Équivalence de Curry-Howard entre le lambda-Pi-calcul et la logique intuitionniste. Rapport de stage de L3, École Polytechnique, 2011.

[9] Gilles Dowek. Polarized resolution modulo. In Cristian S. Calude and Vladimiro Sassone, editors, *IFIP TCS*, volume 323 of *IFIP AICT*, pages 182–196. Springer, 2010.

[10] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*, volume 5 of *Computer Science and Technology Series.* Harper & Row, New York, 1986.

[11] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[12] Gerwin Klein et al. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[13] Konstantin Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando and Peter Baumgartner, editors, *IJCAR*, volume 5195 of *LNAI*, pages 292–298. Springer, 2008.

[14] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[15] Frank Pfenning. Structural cut elimination I. Intuitionistic and classical logic. *Information and Computation*, 157:84–141, 2000.

[16] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1. In Rajeev Gor, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning*, volume 2083 of *LNCS*, pages 376–380. Springer, 2001.

[17] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[18] Stephen Schultz. System description: E 0.81. In David A. Basin and Michaël Rusinowitch, editors, *IJCAR*, volume 3097 of *LNCS*, pages 223–228. Springer, 2004.

[19] G. Sutcliffe. The TPTP world - infrastructure for automated reasoning. In E. Clarke and A. Voronkov, editors, *LPAR*, number 6355 in LNAI, pages 1–12. Springer, 2010.

[20] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.

Checking foundational proof certificates for first-order logic (extended abstract)

Zakaria Chihani, Dale Miller and Fabien Renaud

INRIA and LIX, École Polytechnique, Palaiseau, France

Abstract

We present the design philosophy of a proof checker based on a notion of *foundational proof certificates*. At the heart of this design is a semantics of proof evidence that arises from recent advances in the theory of proofs for classical and intuitionistic logic. That semantics is then performed by a (higher-order) logic program: successful performance means that a formal proof of a theorem has been found. We describe how the λProlog programming language provides several features that help guarantee such a soundness claim. Some of these features (such as strong typing, abstract datatypes, and higher-order programming) were features of the ML programming language when it was first proposed as a proof checker for LCF. Other features of λProlog (such as support for bindings, substitution, and backtracking search) turn out to be equally important for describing and checking the proof evidence encoded in proof certificates. Since trusting our proof checker requires trusting a programming language implementation, we discuss various avenues for enhancing one's trust of such a checker.

1 Introduction

A range of computational systems that prove that certain formulas are, in fact, theorems are in regular use today. Such systems range from interactive theorem provers to model checkers, type checkers, and static analyzers. Generally, these systems produce proof evidence in many and varying formats for classical and/or intuitionistic logics.

It is increasingly recognized that such proof systems need to be able to communicate proofs and to find some means to trust each other. One approach to making such communications possible is to build a particular technological bridge between two specific provers so that proofs from one system can be exported to the other system in such a way it can be checked and trusted. See, for example, [8] where an SMT prover was modified to output its proof evidence as proof scripts that Isabelle could then execute and trust. A similar approach is done with SMTCoq [2] for the type-theory based proof assistant Coq. Other approaches exist: for example, the OpenTheory project [13] attempts to provide a framework where various HOL theorem provers can share proofs.

In this extended abstract, we report on a multi-year effort to design, define, and check *foundational proof certificates*. In this setting, we emphasize a technology-free description of proof evidence making use of basic results and designs taken from the proof theory of sequent calculus *à la* Gentzen and Girard. Recent results in the theory of *focused sequent calculus proofs* are used to build the elements of our framework. We also discuss the design of our certificate checker that exploits higher-order logic programming (specifically λProlog [18]).

2 Trustworthy communication of theorems and proofs

Our main concern here is to communicate the validity of theorems by the transmission of proof evidence from one machine to another machine in such a way that the human operators of these machines can check and trust the transmitted proof. Thus, we remove the human from any interesting involvement with the transmitted proof itself: for example, we are not concerned with whether or not a human can understand the structure of such proofs. While having humans read and learn from machine proofs is an interesting and important topic, we set aside that topic for the more narrow and, hopefully, solvable problem of having machines read and check each other's proofs.

It seems difficult to develop trust in theorem provers and model checkers by completing large scale formal proofs of them. It also seems undesirable to do so given that such systems are often evolving and incorporate experimental concepts for which formal correctness may not be established. Instead, one can turn towards checking their individual outputs, *i.e.*, their claims that certain "proof evidence" exists for a given proposed theorem. One solution could be to have, for every kind of output (and, possibly, for every software version of a prover), a specific proof checker. For example, Coq makes use of a small, trusted kernel for checking proof evidence generated by other parts of that system. Similarly, one can increase confidence in a SAT solver by checking the proofs that they output [6]. While it is probably desirable for every theorem prover to contain a trusted kernel that always checks its proof claims, one still has the problem that there are many checkers to trust and that one is still not addressing the need to *communicate* proof evidence between provers.

Instead of living with a proliferation of proof formats and proof checkers, we propose to explore to what extent we can take a foundational—instead of technological—approach to proof checking. After all, logic and proof have been studied for a long time (longer than, say, context-free grammars and parsing) and that literature is mature and contains a number of deep results backed-up with a lively research community. Furthermore, symbolic logic and proof theory have always purported to deal with the eternal and universal structures behind reasoning.

2.1 Size of the proofs

An important aspect of proofs that makes communicating and trusting them difficult is their size. While the size of formal proofs can vary a great deal among various provers and application domains, formal proofs will almost always be too large for humans to check with confidence. Thus, machines will need to do such checking. There is also evidence that in some settings, the size of proofs can be a challenge for their transmission and storage: the literature on proof carrying code (*e.g.*, [22]) is shaped partly by the need to address large proof objects.

One way to address the problem of communicating and checking such large objects involves allowing a trade-off between explicit proofs and proof reconstruction. A common principle involved with reducing the size of proofs is the Poincaré principle (formulated by Barendregt and Barendsen in [3]): traces of computation should not be included in a proof. One expects, instead, the checker to redo computations which implies that the checker must incorporate into its trusted base a programming language implementation with all its associated components (parsers, printers, compilers, garbage collectors, *etc*). The Poincaré principle is not, however, without problems: for example, if the elided computation comes from a complex algorithm then either that algorithm is implemented outside the checker (thus augmenting the trusted base for each such algorithm) or it is coded within the proof checker as a naive computation that will likely run for too long.

A more interesting approach might be to find ways to handle huge proofs efficiently, which can be done both in a practical way using, for instance, incremental checking as proposed in [24], and in a theoretical way by designing certificates that can be recognized in deterministic log space (*e.g.*, the RUP format for proofs of unsatisfiability [27]).

2.2 Proof reconstruction

Besides leaving out computation, a proof can be further reduced by replacing details, such as "shallow" subproofs or (potentially large) witnesses, with "holes". Filling in those holes can be done by the checker using features such as unification and (bounded) backtracking proof search. For example, instantiating a quantifier in a proof is a small step but describing the substitution term might involve a lot of space. On the other hand, a proof checker using unification might easily determine an appropriate term from context. Similarly, if the checker also involves (bounded) backtracking search, then many small subproofs might well be reconstructed from context, thereby removing the need to insert those subproofs explicitly into the proof certificate.

Given that proof checking will involve performing general computations and proof reconstruction, it seems natural to use logic programming—where unification and backtracking search are central—to build sound and flexible proof checkers. Such a conclusion is certainly not surprising given that relational programming can easily be seen as a generalization of functional programming and given that many key concepts of proof systems are relational, the central one being the most basic relationship $M : A$ between a term (proof) and a type (formula).

3 Proof checking as (logic) programming

While the first automated proof checker was Automath [7], the ML programming language was the first programming language designed to provide a flexible framework for writing proof checkers [10]. This functional programming language has lead to the implementation of the LCF-family of tactics and tacticals that are at the core of many interactive theorem provers. We argue here that logic programming, as opposed to functional programming, is a good choice for doing the kind of proof checking we have described: this is particularly true when the logic programming language chosen is λProlog [18].

3.1 Important programming language features

Below we list various aspects of programming languages that are important for the construction of trusted proof checkers. The first three of these features are present in LCF/ML while all five are present in λProlog.

Strong static typing. In ML, it is possible to describe a type `thm` of theorems. This type can be built using constants representing axioms (of type `thm`) and functions of type, say, `thm -> thm -> thm` denoting a binary inferences rule (such as modus ponens). Thus modeling proofs on the familiar Hilbert-Frege style of proof is easy to capture in ML via its static type checker and type preservation property. While the role of types is rather different in logic programming (see [18] for a discussion of these differences), simple types are also important in λProlog since they are used to denote "syntactic categories" such as formulas, terms, and certificates as well as categories such as "a term-level abstract over formulas" by a type such as `term -> form`. In this setting, capturing the notion that Ξ is a proof of formula B or the notion that a formula is a theorem is done not by types but by predicates (central to all logic programming).

Abstract datatypes. In the LCF/ML approach to proof checking, the type `thm` needs to be protected in the sense that only authorized primitive functions can construct members of type `thm`. In order to enforce this, ML allows this type to be an abstract datatype, which means that the constructors of that type are available to only certain, privileged functions. For example, functions that compute directly with axioms and inference rules can be placed into this abstract type and can be given access to the constructors of `thm`. Any other function that can build theorems must use these privileged functions. In a similar way, the abstract datatypes of λProlog allow one to define constructors for sequents, to allow certain clauses to describe how provability of some sequents are able to infer provability of other sequents (encodings of inference rules), and then to forbid any other clauses from using this sequent constructor. In this way, the rules of inference are sealed from “code injection attacks”.

Higher-order programming. The ability to manipulate functions (in ML) and relations (in λProlog) as first-class objects is not only a powerful programming feature but also makes abstract datatypes far more useful. For example, if a multiset is an abstract datatype, then a natural way to manipulate such a structure is via higher-order programs for, say, applying a certain operation to all elements of a multiset or for selecting elements from a multiset depending on a given predicate.

Backtracking search and unification. While functional programming languages can accommodate these features, they are an essential and central aspect of logic programming. The implementation of these two features has always been a part of the trusted core of logic programming implementations. While historically some Prolog implementations did not provide sound unification (since they did not implement the occurs-check within unification), most modern Prolog systems provide a way to turn this check on. Implementations of λProlog have always implemented sound unification.

Bindings. A proof checker for first-order (quantificational) logic needs to treat syntax with binders. Thus it must handle operations such as checking for equality modulo λ-conversion, instantiating quantifiers, treating eigenvariables and their associated restrictions, and unifying terms and formulas. In particular, λProlog implements the λ-tree approach to higher-order abstract syntax [18].

3.2 Logic foundations for these features

All the features described above are present in *one* logical system, namely a fragment of the intuitionistic version of Church’s 1940 Simple Theory of Types (STT). In order to understand (and implement) what entailment involving λProlog programs should be, one simply needs to understand intuitionistic reasoning in STT. A large literature also exists that describes that logic and various ways to implement it, *e.g.*, goal-directed search [19], higher-order unification [12, 16], backtracking search [20], term representation to support efficient λ-reduction and unification [21], *etc.* While a particular implementation of logic programming is a particular piece of technology, that logic programming language and the checker implemented in it are not tied to that technology. Anyone familiar with the above mentioned literature of logic and algorithms for implementing logic can build their own foundational proof checker. Other proof systems, such as Isabelle and Twelf, make use of a similar intuitionistic foundation. As we shall argue in Section 5, such a declarative and mature foundation can be a great asset in establishing trust of a proof checker.

4 The checker's architecture

Our approach to certificate checking is based on three components—the *kernel*, the *client*, and the *clerks and experts*—described in more details below.

The *kernel* is a λProlog implementation of the LKU focused proof system [15]. Formulas in LKU contain a mix of classical and linear logic connectives and (first-order) quantifiers. Unrestricted, LKU is essentially a verbose presentation of the focused classical logic LKF [14]. The LKU proof system allows for various restrictions to be placed on its structural rules. One set of restrictions (reminiscent of Gentzen's restricting of classical inference rules to only single-conclusion sequents) gives rise to the LJF [14] focused proof systems for intuitionistic logic. Another set of restrictions (reminiscent of Girard's restricting of classical inference rules so that weakening and contraction are not available) gives rise to a focused proof system for (multiplicative-additive) linear logic [1]. Thus this one kernel can capture focused proof systems in these three logics. Even if one uses LKU only to capture classical and intuitionistic proofs, aspects of linear logic still play an important role in LKU. For example, Gentzen's characterization of an intuitionistic sequent as having only a single conclusion is captured, in part, by forbidding contraction of formulas on the right of the encoded sequent: LKU treats this restriction by mixing classical and linear logic connectives. Furthermore, formulas whose introduction rules are invertible (in both classical and intuitionistic proof systems) are treated as *purely linear* within LKU: that is, they are never contracted nor weakened. The linear logic aspects of LKU allow a simple treatment of this important aspect of invertible formulas.

The *client* of the checker is the programmer of a theorem prover who would like to export a proof for checking. The client will not need to know the specifics of our checker: that is, she will not need to know that it is based on a focused sequent calculus or that it is implemented in λProlog. The hope is, instead, that the client will be able to "pretty-print" her proof evidence into a document that can then be checked. Significant effort by the client should not be necessary to transform internal justifications for provability into some strikingly different format. For example, if the client is a resolution refutation prover, the document output for checking should be something familiar, such as a list of numbered clauses as well as a list of triples describing which two clauses resolve to yield a third.

In order to translate the information in the client's proof certificate into instructions to drive the kernel's inference rules, we use a third component composed of *clerks and experts* [4]. An analogy might succinctly convey the spirit of this component of checking. Imagine an accounting office that needs to check that a certain mound of financial documents (provided by the client) represents a legal transaction (as judged by the kernel). The office workers called experts are given the responsibility of looking into the mound and extracting information: they must *decide* into which series of transactions to dig and they need to know when to *release* their findings for storage and later reconsideration. On the other hand, the clerks are responsible for taking information released by the experts and performing some computations on them, including their *indexing* and *storing*. The justification of this division of effort between clerks and experts comes from the structure of focused sequent proof systems [1, 14, 15]: experts operate during the *synchronous* phase of proof construction while the clerks operation during the *asynchronous* phase. Furthermore, the vocabulary of *decide*, *release*, and *store* is reused as structural rules within the focused proof system. The actual definition of a proof certificate format essentially amount to describing a flow of work between the experts and the clerks. Such a work-flow is defined using small λProlog programs that define predicates that interface with the kernel. That interface has been designed so that no matter how badly coded the clerks and experts are, the soundness of the kernel is never compromised.

The current implementation of our proof checker can be found at `https://team.inria.fr/parsifal/proofcert/`.

5 Avenues to trust

We would like to be able to claim that our checker is sound: that is, if our checker succeeds in checking a proof certificate for a formula B from some client, then B is a theorem. Trust in such a claim is built around many elements, so we break this claim into smaller elements.

First, we must trust the logic programming language implementation and the many things on which it depends. In our current setting, we rely on the correctness of the Teyjus implementation [26] of λProlog (which is itself implemented in OCaml and C) and a host of associated computing subsystems: printers, parsers, compilers, garbage collectors, hardware processors, etc. Pollack's paper [23] explores many of the trust aspects of such components when applied to proof checking.

Second, one must trust that the LKU proof system [15] is, in fact, correct in its claim of representing proofs for classical, intuitionistic, and linear logics. Since developing trust in a mathematical text is a familiar problem to academics, we do not elaborate on this here.

Third, one must trust that our logic programming implementation of LKU is correct and that there is no way for "malicious" experts and clerks to "attack" our kernel. The λProlog programming language provides many features that make it easy to trust this aspect of correctness: (*i*) Bindings, eigenvariables, and substitutions are implemented within the language and with great care for their correct treatment. (*ii*) Sequent calculus inference rules (such as those in LKU) naturally correspond to Horn clauses so it is easy to examine whether or not a set of Horn clauses correctly captures a proof system. (*iii*) By exploiting abstract datatypes, it is possible to close the set of inference rules so that no attacker can add new inference rules to the kernel. Finally, the interface between the kernel and the clerks and experts is via a set of predicates which are defined via λProlog clauses. Furthermore, these predicates are used only as premises to the clauses specifying the various LKU inference rules. As a result, it is a relatively easy matter to verify that our λProlog source files do indeed implement our LKU kernel in a sound fashion.

We would also like to insist that by employing *declarative* techniques in specifying a proof checking kernel, we are adhering to a proven approach to writing trustworthy software. Consider, by analogy, writing a parser for some programming language. Often one tries to construct such a parser in two distinct steps. First, one *declares* the lexical structure (using techniques from finite state machines) and grammar (using techniques from context-free language theory). Given these specifications, one then uses tools—such as `lex` and `yacc`—that generate code that actually tokenizes and parses input strings. Of course, these tools are complex but since they are so commonly used, since their formal foundation is well understood and documented, and since a number of people depend on them to be correct, parsers achieved by this route are often considered more trustworthy than if one implements a parser by hand. The architecture behind our proof checking system similarly relies on declarative specifications (of inference rules and of the clerks and experts) since their semantics is clear (by being founded on logic). The many other tools on which we rely (such as Teyjus) are used by a number of other people for different tasks and usually with similar concerns for correctness.

We have discussed only the soundness of proof checking. If one provides a resolution refutation as a proof certificate for some formula B (the paper [4] describes how this can be done), the only conclusion we can claim from a successful run of the kernel is that B is a theorem. The kernel makes no claim that the certificate is, in fact, a proper resolution refutation. Specific knowledge of how the clerks and experts work is needed to provide such a guarantee. The specific resolution checker that was presented in [4] will accept all resolution refutations (with factoring) but will accept more things than are officially defined as resolution steps. Of course, the additional items that are accepted are still sound proof evidence even if they are not proper resolution steps.

6 Related and Future Work

There are many projects trying to get different theorem provers to communicate proofs and a few of these have the goal of being universal. One of these is Dedukti [25] which aims at capturing all intuitionistic proofs using $\lambda\Pi$-calculus modulo as its foundation [5]. While Dedukti separates computation from deduction (the former is not part of a certificate but is executed by the checker), it does not support directly the possibility of doing (bounded) proof reconstruction. It is also not clear whether or not such an intuitionistic framework will treat classical logic well. One can always use the excluded middle as an assumption within intuitionistic logic but this means that instances of the excluded middle axiom must be part of the certificate. Also the use of axioms leads one away from truly *analytic* proof theory in which subformulas of a conjectured sequent are needed for consideration within (cut-free) proofs. Also in the general area of enhancing intuitionistic proof representations for checking, there is also recent work on extending the $\lambda\Pi$-calculus with side conditions [24] and with external predicates [11].

The proof checker described here is currently restricted to *first-order logic*: we are planning to extend this work to include proof checking in logics with least and greatest fixed points as well as higher-order quantification. We also hope to eventually extend this kind of checking to both partial proofs and counterexamples [17]. In order to increase confidence in various aspects of our existing checker, we plan to undertake some formal proofs involving our code in the Abella prover [9]. From the efficiency point of view, as of now, it is not yet clear how effective our current software architecture will handle large proof certificates or intensive checker-side computations. Teyjus is currently under development in anticipation of some of these potential problems.

Acknowledgments We thank the anonymous reviewers of an earlier version of this paper for their comments. This work was funded by the ERC Advanced Grant ProofCert.

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.

[2] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs (CPP 2011)*, LNCS 7086, pages 135–150, 2011.

[3] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *J. of Automated Reasoning*, 28(3):321–336, 2002.

[4] Zakaria Chihani, Dale Miller, and Fabien Renaud. Foundational proof certificates in first-order logic. In Maria Paola Bonacina, editor, *CADE 24: Conference on Automated Deduction 2013*, LNAI 7898, pages 162–177, 2013.

[5] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In S. Ronchi Della Rocca, ed, *TLCA: Typed Lambda Calculi and Applications*, LNCS 4583, pages 102–117, Springer, 2007.

[6] Ashish Darbari, Bernd Fischer, and Joao Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In *Theoretical Aspects of Computing–ICTAC 2010*, pages 260–274. Springer, 2010.

[7] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, New York, 1980.

[8] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In H. Hermanns and J. Palsberg, eds, *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 3920, pages 167–181. Springer, 2006.

[9] Andrew Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, *LNCS* 5195, pages 154–161. Springer, 2008.

[10] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, LNCS 78. Springer, 1979.

[11] Furio Honsell, Marina Lenisa, Luigi Liquori, Petar Maksimovic, and Ivan Scagnetto. LFP: a logical framework with external predicates. In *LFMTP'12: International workshop on Logical frameworks and meta-languages, theory and practice*, pages 13–22. ACM New York, 2012.

[12] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[13] Joe Hurd. The OpenTheory standard theory library. In *The Third International Symposium on NASA Formal Methods*, LNCS 6617, pages 177–191, 2011.

[14] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[15] Chuck Liang and Dale Miller. A focused approach to combining logics. *Annals of Pure and Applied Logic*, 162(9):679–697, 2011.

[16] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.

[17] Dale Miller. A proposal for broad spectrum proof certificates. In J.-P. Jouannaud and Z. Shao, editors, *CPP: First International Conference on Certified Programs and Proofs*, LNCS 7086, pages 54–69, 2011.

[18] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic.* Cambridge University Press, June 2012.

[19] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[20] Gopalan Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11(1):115–145, August 1993.

[21] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.

[22] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Logic in Computer Science*, pages 93–104, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[23] Robert Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory.* Oxford University Press, 1998.

[24] Aaron Stump. Proof checking technology for satisfiability modulo theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

[25] The Dedukti team. The Dedukti system and homepage. `https://www.rocq.inria.fr/deducteam/`

Dedukti/index.html, 2013.

[26] The Teyjus team. The Teyjus website. http://code.google.com/p/teyjus, 2013.

[27] Allen Van Gelder. Producing and verifying extremely large propositional refutations: Have your cake and eat it too. *Annals of Mathematics and Artificial Intelligence*, 65(4):329-372, 2012

Translating Higher-Order Specifications to Coq Libraries Supporting Hybrid Proofs

Nada Habli[1] and Amy P. Felty[1,2]

[1] Department of Mathematics and Statistics, University of Ottawa, Canada
[2] School of Electrical Engineering and Computer Science, University of Ottawa, Canada
nhabl094@uottawa.ca, afelty@eecs.uottawa.ca

Abstract

We describe ongoing work on building an environment to support reasoning in proof assistants that represent formal systems using higher-order abstract syntax (HOAS). We use a simple and general specification language whose syntax supports HOAS. Using this language, we can encode the syntax and inference rules of a variety of formal systems, such as programming languages and logics. We describe our tool, implemented in OCaml, which parses this syntax, and translates it to a Coq library that includes definitions and hints for aiding automated proof in the Hybrid system. Hybrid itself is implemented in Coq, and designed specifically to reason about such formal systems. Given an input specification, the library that is automatically generated by our tool imports the general Hybrid library and adds definitions and hints for aiding automated proof in Hybrid about the specific programming language or logic defined in the specification. This work is part of a larger project to compare reasoning in systems supporting HOAS. Our current work focuses on Hybrid, Abella, Twelf, and Beluga, and the specification language is designed to be general enough to allow the automatic generation of libraries for all of these systems from a single specification.

1 Introduction

The Hybrid system [4] provides support for reasoning about *object languages* (OLs) such as programming languages and other formal systems using *higher-order abstract syntax* (HOAS). In [4], two versions of Hybrid are described, one implemented in Isabelle [8], and one implemented later in the Coq Proof Assistant [1] by fairly directly porting the Isabelle version to Coq. We focus on the Coq version here. Hybrid provides support for encoding syntax, for representing the semantics via inference rules and axioms, and for reasoning about the properties of the OL. For example, reasoning about the metatheory of a programming language allows important properties, such as soundness, to be established formally. Such properties are important for providing assurance that a language can be used to build reliable and secure software systems.

The general Hybrid infrastructure is implemented as two Coq libraries. The first provides an underlying de Bruijn representation of λ-terms parameterized by a set of constants for a particular OL. This layer is hidden from the user. This library includes a set of definitions and lemmas that builds an HOAS layer from this lower level, which is used to encode the syntax of OLs. The only axiom used in the implementation is the law of excluded middle, included by importing Coq's library for classical logic.[1] The reasoning infrastructure has multiple levels also. The inference rules of an OL are defined at the lowest level as logic programming-like clauses (called *prog* clauses here) that are provided as a parameter to an intermediate logic,

[1] This library was originally imported in order to keep the Coq implementation close to the Isabelle one. It is in fact not necessary. See [2] for a constructive version of Hybrid in Coq.

called a *specification logic* (SL). The second Coq library implements the SL. At the highest level is the *reasoning logic*, which is Coq.

This paper presents our tool for supporting reasoning in Hybrid by translating high-level specifications of OLs to Coq libraries. Our specification language has three sections, `Syntax`, `Judgments`, and `Rules`. The first section contains the specification of new constants and their types, representing the basic syntax constructors of the OL. The allowed types are a subset of the types of the simply-typed λ-calculus. In HOAS, object-level binding is encoded directly using meta-level binding, and thus arguments to constructors are allowed to have function types. We restrict to second-order types, which means that the functions appearing as arguments must themselves take arguments of atomic types. Hybrid itself is currently restricted to second-order since the representation of many formal systems does not require more. The declarations in this section are used to generate the set of constants needed for the de Bruijn level, as well as a set of definitions for encoding syntax at the HOAS level.

The declarations in the second section introduce SL-level predicates. These are the predicates used to encode the judgments in the inference rules defining the semantics of the OL. The third section defines the OL inference rules, which are translated to prog clauses. Together, the predicates and clauses instantiate the required parameters of the SL.

We present the specification language and our translation tool informally via an example, which is described in Section 2. In Section 3, we describe the technical details of some of our algorithms and their implementation. This work is part of an ongoing larger project to compare reasoning in a variety of systems that reason using HOAS (see [3], for example). In Section 4, we discuss the current focus of our work on the translation tool in the context of this larger project. In Section 5, we conclude and discuss our longer term goals.

2 An Example: The Polymorphic λ-Calculus

As an example, we consider typing for the polymorphic λ-calculus as defined in [10]. The syntax is defined by the following grammars, and typing is defined by the rules below.

$$\begin{array}{lll} \text{Terms} & M, N & ::= \; x \mid \lambda x : T.M \mid M \; M \mid \lambda\alpha.M \mid M[T] \\ \text{Types} & S, T & ::= \; \alpha \mid T \to T \mid \forall\alpha.T \end{array}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\, ty_v \qquad \frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash MN : T}\, ty_a \qquad \frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x : S.M : S \to T}\, ty_l$$

$$\frac{\Gamma \vdash M : \forall\alpha.T}{\Gamma \vdash M[S] : [S/\alpha]T}\, ty_{ta} \qquad \frac{\Gamma, \alpha \vdash M : T}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.T}\, ty_{tl}$$

Figure 1 encodes the syntax and typing rules, and illustrates the use of our specification language. In the `Syntax` section, the keyword `type` introduces new atomic types for the different syntax classes, which are the polymorphic types (`tp`) and terms (`tm`) in this example. Abstraction in types and terms (defined by constants `all`, `lam`, and `tlam`) is defined using function types. Thus in the HOAS representation, abstraction in the OL will be represented using abstraction in the meta-language, which here is Coq's λ-abstraction.

The typing judgment for the polymorphic λ-calculus is expressed using the `typeof` predicate declared in the `Judgments` section. Here, the keyword `type` will map to the type of propositions of the target system, which for Hybrid is the type of formulas of the SL. Note that we use the

```
Syntax
tp: type.
arr: tp -> tp -> tp.             all: (tp -> tp) -> tp.

tm: type.
app: tm -> tm -> tm.             lam: (tm -> tm) -> tp -> tm.
tapp: tm -> tp -> tm.            tlam: (tp -> tm) -> tm.

Judgments
typeof: tm -> tp -> type.

Rules
ty_a: typeof M (arr S T) -> typeof N S -> typeof (app M N) T.
ty_l: (Pi x. typeof x S -> typeof (M x) T) -> typeof (lam (\x. M x) S) (arr S T).
ty_ta: typeof M (all (\a. T a)) -> typeof (tapp M S) (T S).
ty_tl: (Pi a. typeof (M a) (T a)) -> typeof (tlam (\a. M a)) (all (\a. T a)).
End
```

Figure 1: A Specification for the Polymorphic λ-Calculus

type keyword in both sections, borrowing from Twelf [11], where predicates and types are at the same level.

The inference rules appear in the last section, and here again the syntax resembles the syntax of Twelf to some extent. As in Twelf, the contexts that are explicit in the informal presentation of the rules are implicit in the judgment section of the specification. The rules are named, and the arrow is used to separate hypotheses from one another and from the conclusion, which is the last formula before the terminating dot. Binders in the polymorphic λ-calculus are represented using the binding operator of our specification language (backslash). We use tokens starting with uppercase letters for "schematic" variables (used to represent terms and types of the λ-calculus in this example) and tokens starting with lowercase letters for constructors, predicates, rule names, and bound variables.

From this fairly small specification, we generate a library that can be directly loaded into Coq, part of which is shown in Figures 2 and 3. As mentioned earlier, Hybrid is implemented in both Isabelle and Coq, and both implementations are described in [4]. As we present the Coq code in this section, we will often refer to results from [4] that are relevant. For the reader interested in looking up these results, we note that most of the formal definitions and statements in that paper use a pretty-printed version of code that can be viewed as either Isabelle or Coq syntax. (See pages 48–49 for a description of this notation.) In the text of [4], when the implementations diverge, it is explicitly stated. (For example, see Section 2.2.)

The set Econ in Figure 2 is the set of constants that serve as a parameter to the de Bruijn representation of terms. Note that there is one for each constructor in the Syntax section. The next 3 lines perform this parameter instantiation and are the same for any Hybrid OL library. The last 6 lines of the Constants section fill in the Hybrid definitions for the HOAS representation of the 6 constructors. They are defined in terms of their underlying de Bruijn representation. The constant lambda is a binding operator defined on top of the de Bruijn representation, and its definition is part of the infrastructure hidden from the user. Types of bound variables in Coq are not explicitly added since they can be inferred. In these Coq definitions, there is no distinction between the types tp and tm found in the specification. All terms have Coq type uexp (the type of de Bruijn terms parameterized by the set ECon) and all

```
Require Import sl.

Section encoding.
(************************************************************************
        Constants
*************************************************************************)
Inductive ECon: Set :=  Carr: ECon | Call: ECon | Capp: ECon |
                        Clam: ECon | Ctapp: ECon | Ctlam: ECon.

Definition uexp: Set := expr ECon.
Definition Var: var -> uexp := (VAR ECon).
Definition Bnd: bnd -> uexp := (BND ECon).

Definition arr:= fun T1 => fun T2 => (APP (APP (CON Carr) T1) T2).
Definition all:= fun T1 => (APP (CON Call) (lambda T1)).
Definition app:= fun T1 => fun T2 => (APP (APP (CON Capp) T1) T2).
Definition lam:= fun T1 => fun T2 => (APP (APP (CON Clam) (lambda T1)) T2).
Definition tapp:= fun T1 => fun T2 => (APP (APP (CON Ctapp) T1) T2).
Definition tlam:= fun T1 => (APP (CON Ctlam) (lambda T1)).
(**************************************************************************
        The atm type and instantiation of oo.
***************************************************************************)
Inductive atm : Set :=
| is_tp : uexp -> atm
| is_tm : uexp -> atm
| typeof : uexp -> uexp -> atm.
Definition oo_ := oo atm ECon.
...
```

Figure 2: A Hybrid Library for Reasoning about the Polymorphic λ-Calculus Part 1

arguments to `lambda` have type (`uexp -> uexp`).

Note that not all Coq functions of type (`uexp -> uexp`) encode object-level λ-terms. Those that do not are often called *exotic* terms. Only functions that behave *uniformly* or *parametrically* on their arguments represent λ-terms. Hybrid includes a predicate `abstr` that rules out exotic terms and identifies exactly those terms that represent OL terms. (See Section 2, pages 52–54 in [4] for a definition of `abstr` as well as other definitions it depends on.) This predicate appears in the Coq code obtained from translating the OL inference rules to prog clauses. (See Figure 3.)

The types `atm` and `oo_` are the Hybrid types of atomic and general formulas, respectively, of the SL. A sequent calculus for the SL is implemented in Hybrid as an inductive predicate defining the type `oo_`. In [4], two sample SLs are given, one for a fragment of second-order intuitionistic logic, and another for an ordered linear logic. The former is used in the work described here. (See Figure 5 on page 68 of [4].) The sequent calculus is analogous to a logic programming interpreter, where the prog clauses can be viewed as a second-order logic program. We note that contexts in the SL are explicitly represented in the inductive definition, while they are implicit in the prog clauses. For readers familiar with Twelf, the prog clauses correspond to a Twelf program, while the SL corresponds to Twelf's meta-level, where OL contexts are represented as meta-level contexts. In Hybrid, both levels are formalized, and thus contexts are explicitly represented and reasoned about at the SL level.

```
(************************************************************************
        Definition of prog
*************************************************************************)
Inductive prog : atm -> oo_ -> Prop :=
| tp_arr: forall T1, forall T2,
     prog (is_tp (arr T1 T2)) (Conj (atom_ (is_tp T1)) (atom_ (is_tp T2)))
| tm_lam: forall T1, abstr T1 -> forall T2,
     prog (is_tm (lam T1 T2))
          (Conj (All (fun x1 => (Imp (is_tm x1) (atom_ (is_tm (T1 x1))))))
                (atom_ (is_tp T2)))
...
| ty_a : forall M, forall S, forall T, forall N,
    prog (typeof (app M N) T) (Conj (atom_ (typeof M (arr S T))) (atom_ (typeof N S)))
| ty_l : forall S, forall M, abstr M -> forall T,
    prog (typeof (lam (fun x=>  (M x)) S) (arr S T))
         (All (fun x => (Imp (typeof x S) (atom_ (typeof (M x) T)))))
| ty_ta : forall M, forall T, abstr T -> forall S,
    prog (typeof (tapp M S) (T S)) (atom_ (typeof M (all (fun a => (T a)))))
| ty_tl : forall M, abstr M -> forall T, abstr T ->
    prog (typeof (tlam (fun a => (M a))) (all (fun a => (T a))))
         (All (fun a =>  (atom_ (typeof (M a) (T a))))).
Hint Resolve tp_arr tp_all  tm_app tm_lam tm_tapp tm_tlam  ty_a ty_l
  ty_ta ty_tl : hybrid.
```

Figure 3: A Hybrid Library for Reasoning about the Polymorphic λ-Calculus Part 2

Figure 2 defines `atm` as an inductive set of predicates. Continuing the logic programming analogy, these can be viewed as predicates of a logic program. The binary predicate `typeof` comes directly from the specification. Again, the types of the arguments of this predicate (`tm` and `tp`) in the specification are mapped to `uexp` in the Coq library. As a result, we need to introduce a predicate corresponding to each type in the specification to be used to identify well-formed types and terms of the polymorphic λ-calculus. Here, the `is_tp` and `is_tm` predicates are introduced for this purpose. The last definition in the figure instantiates the `oo_` type, which must be done after the `atm` parameter is defined. The elided part includes other definitions involved in instantiating this type, as well as hints to Coq to help with automating proofs.

Figure 3 defines the prog clauses (or logic program) which serve as the final parameter to the SL. The last 4 clauses are direct translations of the rules in the specification. The `prog` predicate takes two arguments: the conclusion of an inference rule (the head of a logic programming clause) followed by the premise or premises (the body of the logic programming clause). The constructors `Conj`, `Imp`, and `All` are the connectives of the SL, and `atom_` coerces `atm` to `oo_`. `Pi` in the specification language maps to `All`, embedded implication maps to `Imp`, and multiple hypotheses are separated by `Conj`. Note that schematic variables in the specification are implicitly quantified at the outermost level. Explicit quantifiers (`forall` in Coq) are added to each clause of the definition of `prog` as part of the translation.

The other clauses, including the elided ones, are the rules for determining well-formed terms and well-formed types. These are automatically generated from the type declarations in the `Syntax` section, and represent the most complex part of the translation implemented so far.

In Hybrid, as in other logical frameworks such as Twelf, we must be sure that the syntax and inference rules are *adequately* encoded in the meta-language. Proving adequacy involves proving that there is a one-to-one correspondence between the syntax of the OL and its representation in the meta-language, and that an OL judgment has a proof using the inference rules if and only if the encoded version of the judgment is provable in the logical framework. For an example of how adequacy is proved in Hybrid, see Section 3.2 of [4]. The rules for well-formed terms of the OL are an important component of adequacy proofs in Hybrid. In our example OL, for instance, we must prove that whenever there is a proof in Hybrid that a term of the polymorphic λ-calculus has a particular type, then both the term and the type are well-formed (as defined by the clauses for `is_tp` and `is_tm`).

3 Implementation of the Translation Tool

In this section, we describe the overall structure of the implementation of the translation, which as mentioned, is in OCaml. We have not given a formal definition of the syntax of the specification language, so this description is informal. Using `mllex` and `mlyacc`, the 3 sections of a specification are each parsed to a list of type `(string * exp) list`. In each pair in the list, the first argument is the constructor, predicate, or rule name, and the second argument is an element of the following type:

```
type exp = Type | Id of string | Arrow of exp * exp | App of string * exp list |
           Lambda of string * exp * exp | Pi of string * exp * exp
```

There is a direct mapping of each operator in the specification to a constructor of `exp`. For example, the combination of `\` and the dot separating the bound variable from the term maps to `Lambda`. The `type` keyword maps to `Type`, and all other identifiers to `Id`. We unify the syntax of all three sections of the specification, even though the first two do not use `Lambda` and `Pi`. The translation function has the following overall structure, divided here into 4 steps, with details of step 3(e) filled in a bit further in Figure 4.

1. Parse the input file into three lists of declarations: `ds1`, `ds2`, and `ds3` where each one is the parsed input of `Syntax`, `Judgments`, and `Rules`, respectively.

2. Call functions to isolate the following variables:

   (a) `Typelist`: from `ds1`, obtain the list of identifiers (strings) from declarations of the form "`id:type.`" in the specification.

   (b) `Syntax_listName_aux`: from `ds1` and `Typelist`, obtain a list of lists of the remaining identifiers (OL syntax constructor names); use `Typelist` to group them into sublists according to the target types of the constructors (the types just before the terminating dots).

   (c) `Syntax_listExpr_aux`: from `ds1` and `Typelist`, form the list of lists of types of the constructors (expressed as elements of type `exp`), using the same groupings into sublists as above in (b).

   (d) `Rules_listName`: from `ds3`, get the list of the identifiers corresponding to rule names.

3. Call functions to create the following strings using the variables from step 2.

   (a) `string1`: from `Syntax_listName_aux` construct the string for the inductive definition of `ECon`, with one case of the Coq definition for each constructor in `Syntax_listName_aux` with names prepended by `C`. (See `ECon` in Figure 2.)

(b) `string2`: from `Syntax_listName_aux` and `Syntax_listExpr_aux`, construct a string with one line for each constructor, containing a Coq definition for the encoding of syntax for that constructor. Two cases must be considered, depending on whether the constructor's type is first- or second-order. If there is a functional argument, the `lambda` operator is used. (See the 6 definitions at the end of the `Constants` section of Figure 2.)

(c) `string3`: from `Typelist` construct a string for the inductive definition of `atm` containing all the clauses for the well-formedness predicates (those of type `uexp -> atm`, see Figure 2).

(d) `string4`: from `ds2` construct a string containing clauses of the inductive definition of `atm`, one for each predicate in the `Judgments` section. Judgments cannot have function arguments; their types are first-order. We simply count the number of argument types and write "`uexp ->`" for each one, ending the clause with `atm`. (See the last clause of the definition of `atm` in Figure 2.)

(e) `string5`: from `Syntax_listName_aux` and `Syntax_listExpr_aux`, construct a string containing all the prog clauses for well-formedness of OL terms. See Figure 4 for some details of the implementation. (See also Figure 3, which contains 2 of 6 such clauses, with the rest elided.)

(f) `string6`: from `ds3` construct a string containing one prog clause corresponding to each rule in the `Rules` section. We omit the details. (See the last 4 prog clauses in Figure 3.)

(g) `string7`: From `Syntax_listName_aux` and `Rules_listName` it is straightforward to construct the `Hint` string. (See the last line of Figure 3.)

4. Write the following strings to the output file in the appropriate order: strings representing Coq comments, fixed strings (library elements that are the same for all specifications), and the strings obtained from step 3.

4 Extensions

In this section, we discuss the extensions of our tool that we are currently working on, as well as some other near-term goals.

There are a variety of standard lemmas that are useful for reasoning about OLs that can be directly generated from the specification. The next step in our current work is to add capabilities to our tool to automatically generate the statements of these lemmas from the specification. In addition, their proofs are mostly easily automated. Part of our work involves improving Hybrid to include better tactics for automating such proofs. In addition, we envision augmenting the translation to automatically insert parts of a proof script into the Coq libraries. Our current work involves studying the most effective way to combine these two techniques for automating proofs. This approach is used in a variety of other tools such as Krakatoa [6], which automatically generates Coq libraries for Hoare-style reasoning about correctness of Java programs, and uses tactics designed specifically for automating proofs in this domain.

We also have done some preliminary work on extending the specification language to include a declaration section for contexts, used to represent a set of hypotheses. Many theorems require reasoning about contexts, and our previous work on comparing systems [3] focused particularly on this aspect.

Examples of the kinds of "standard lemmas" that we would like to generate and prove partially or fully automatically include lemmas for adequacy and lemmas for dealing with

Loop 1 (outermost): For each type name tname in Typelist, each corresponding list of constructors cnames in Syntax_listName_aux and list of expressions representing types ctypes in Syntax_listExpr_aux, execute Loop 2. Using the first 3 declarations in Figure 1 as an example, the following data is used the first time through Loop 2:

```
tname = "tp"     cnames = ["arr";"all"]
ctypes = [Arrow (Id "tp", Arrow (Id "tp", Id "tp"));
          Arrow (Arrow (Id "tp", Id "tp"), Id "tp")]
```

Loop 2 1. From tname and cnames, build a list rnames of rule names for prog clauses. (In the example, the result is ["tp_arr";"tp_all"].)

2. For each element rname of rnames and the corresponding element ctype of ctypes, execute Loop 3.

Loop 3 1. Count the number of arguments in ctype by finding the number of external arrows (all those except arrows in function types of arguments). Create a list args of pairs containing variables T1, T2, ..., Tn, one for each argument and arity of the argument (0 for non-functional arguments). For the first elements of ctypes, we get [("T1",0);("T2",0)] and for the second element, we get [("T1",1)].

2. Using rname, ctype, and the corresponding element of args, build a string by concatenating the following substrings:

- "| " ^ rname ^ ":"
- For each ("Ti",m) in args, add "forall Ti,". If m $>$ 0, add "abstr Ti ->".
- "prog (is_" ^ tname ^ "(" ^ cname
- For each pair in args write the first element followed by a space. At the end, add "))".
- Create a string of the form (Conj s_1 (Conj s_2 ... (Conj s_{n-1} s_n)$\cdots$)) where n is the number of elements of args. If $n = 1$, the string is just s_1 with no Conj.
- If the i^{th} element of args is ("Ti",0), s_i is "(atom_ (is_" ^ t ^ " Ti))" where t is the corresponding identifier in ctype (always "tp" in this example).
- If the i^{th} element of args is ("Ti",m) where m $>$ 0, then create variables x1, ..., xm. Form s_i as follows: for each xj, add the substring "(All (fun xj => (Imp (is_" ^ tj ^ "xj)"; end s_i with "(atom_ (is_" ^ t ^ " (Ti x1...xm)...))" where tj and t are the appropriate types in ctype.

For our example, from tname, the first elements of cnames and ctypes, and the first list args above, the output string we obtain is the first clause of the inductive definition of prog in Figure 3.

Figure 4: Building string5 from Step 3(e).

explicit contexts, as well as a variety of others. For example, the adequacy lemma mentioned in Section 2 can be automatically generated. Many proofs in Hybrid proceed by induction over the SL with inversion over both the definitions of the SL and the prog clauses of the OL. In addition to the inversion lemmas automatically generated from a Coq inductive definition, we state and prove specialized inversion lemmas that can greatly simplify Hybrid proofs. The first lemma in Figure 5 is an example of such a lemma, one whose statement and proof can be automatically generated. Note that seq_ is Hybrid's predicate for SL sequents. It takes 3 arguments: the height of a proof, a context of assumptions, and the formula to be proved. The proper predicate appearing in the lemma is important for adequacy (see [4]).

Context weakening is a general lemma that follows from the definition of the SL, and it

```
Lemma ty_l_inv : forall (i:nat) (Psi:list atm) (M:uexp->uexp) (T1 T2:uexp),
  (forall x : uexp, proper x ->
    seq_ i Psi (Imp (typeof x T1) (atom_ (typeof (M x) T2)))) ->
  exists j:nat, (i=j+1 /\
    forall x : uexp, proper x ->
      seq_ j (typeof x T1::Psi) (atom_ (typeof (M x) T2))).

Lemma simple_strengthen : forall (i:nat) (T x y:uexp) (Gamma:list atm),
  seq_ i (is_tm x::is_tp y::Gamma) (atom_ (is_tp T)) ->
  seq_ i (is_tp y::Gamma) (atom_ (is_tp T)).
```

Figure 5: Example OL Lemmas

is stated and proved in the SL library. Context "strengthening" on the other hand, where assumptions that are irrelevant to the proof of a particular judgment are removed, depends on the OL. A simple example of a strengthening lemma is given in Figure 5. For several examples that occur in the context of a case study, see [3] (e.g., the occurrence of strengthening in the proof of Theorem 2). Part of our extension to the specification language will include the capability to specify at a high level what kinds of strengthening lemmas are desired, and then automatically generate and prove or partially prove them as part of the translation.

Our specifications can be translated to libraries for other systems supporting reasoning with HOAS, although we have not yet done so. Much of the work done here, however, can be directly reused, including, of course, the parsing of a specification to its internal representation in OCaml, as well as much of the overall structure of the OCaml functions we have defined to perform the translation. The systems we are currently targeting are Abella [5], Twelf [11], and Beluga [9]. A common characteristic of all of these systems is multi-level reasoning. A straightforward modification of the translation is all that should be required to obtain a basic input library for each of these systems. We mentioned earlier that our specification language adopts several features of Twelf directly. In fact, translation to Twelf will be the most straightforward to implement. Adding more significant support for each of these systems, such as including specialized lemmas, will require further effort.

5 Conclusion and Future Work

We have described our translation tool, which provides support for reasoning in Hybrid about object languages expressed using HOAS. We presented the specification language, described the translation to a Hybrid library, and discussed the implementation as well as several extensions planned for the near term.

In the longer term, we would also like to examine translations to more systems, in order to facilitate a more fuller comparison of reasoning in systems supporting HOAS. Such work may require extending the specification language. The work presented here can be considered as a variant of the Ott project [12] tailored specifically to the needs of HOAS. Ott contains constructs for specifying binders, and one of our longer term goals is to integrate our tool with Ott. In the present work, we chose to start with a smaller simpler language targeted to the needs of Hybrid, Abella, Twelf, and Beluga. Another approach is to consider using the higher-order logic programming language λProlog [7] as both the specification language as well as the implementation language for the translation.

Finally, we mentioned in Section 1 that both Hybrid and the specification language restrict the definition of syntax of OLs to second-order types. Another long-term goal is to generalize Hybrid to higher-order, which will then require extending our specification language and translation.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

[2] Venanzio Capretta and Amy P. Felty. Combining de Bruijn indices and higher-order abstract syntax in Coq. In *TYPES*, pages 63–77, 2006.

[3] Amy Felty and Brigitte Pientka. Reasoning with higher-order abstract syntax and contexts: A comparison. In *International Conference on Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2010.

[4] Amy P. Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

[5] Andrew Gacek. The Abella interactive theorem prover (system description). In *4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008.

[6] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):86–106, 2004.

[7] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic.* Cambridge University Press, 2012.

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer Verlag, 2002.

[9] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *5th International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer, 2010.

[10] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[11] Carsten Schürmann. The Twelf proof assistant. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*, pages 79–83. Springer, 2009.

[12] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool suppor for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.

Initial experiments on deriving a complete HOL simplification set

Cezary Kaliszyk and Thomas Sternagel
{cezary.kaliszyk,thomas.sternagel}@uibk.ac.at

University of Innsbruck, Innsbruck, Austria

Abstract

Rewriting is a common functionality in proof assistants, that allows to simplify theorems and goals. The set of equations to use in a rewrite step has to be manually specified, and therefore often includes rules which may lead to non-termination. Even in the case of termination another desirable property of a simplification set would be confluence. A well-known technique from rewriting to transform a terminating system into a terminating and confluent one is completion. But the sets of equations we find in the context of proof assistants are typically huge and most state-of-the-art completion tools only work on relatively small problems. In this paper we describe our initial experiments with the aim to close the gap and use rewriting to compute a complete first-order simplification set for a HOL-based proof assistant fully automatically.

1 Introduction

Proof assistants are computer programs that aid the user in building a proof that can be mechanically checked. A typical proof assistant includes a number of algorithms that perform common proof operations in an automated way. One of such mechanisms is rewriting and conditional rewriting (often called *simplification* in the context of formal proof). There are two ways of performing rewriting in a proof assistant, depending on the working style: rewriting provided as a forward derivation rule that lets one rewrite a theorem using (possibly conditional) equalities and rewriting as a tactic, that uses equations to rewrite the current goal to an equivalent goal.

To perform a rewriting step, the user needs to choose the equations to rewrite with. Typically choosing this set is done completely manually. In certain cases, however, there exist defaults (for example Isabelle [17] simplification set) or lists of theorems to use (for example `ARITH` in HOL Light [8] or hint databases in Coq [1]). Such default sets are also defined manually, and developers try hard to avoid creating simplification sets that are non-terminating. Unfortunately this problem is quite hard, and for example the usual simplification set defined for many theories in Isabelle includes rules that lead to non-termination of the `simp` tactic.

In order to obtain an even stronger proof technique with the help of rewriting, one can consider normal forms of expressions with relation to some theory. Completion of rewriting is a technique that lets us derive rewrite systems that follow given sets of equations, but are terminating and confluent. A terminating and confluent rewrite system for a theory would give a complete decision procedure for establishing equalities in this theory, giving a strong proof technique.

Termination of term rewrite systems (TRSs) is an undecidable property. Nevertheless a vast number of methods have been developed to determine termination, many of which are suitable for implementation. For example the tools $\mathsf{T_TT_2}$ [12] or AProVE [6] implement techniques for automatically proving the termination of a first-order rewrite system. The termination methods implemented by these systems are sound, but the tools may produce unsound proofs (for example due to implementation bugs). Therefore a number of proof certification techniques have been developed, for example CeTA [22] which can certify termination (or non-termination) proofs provided by a termination tool.

Likewise confluence of TRSs is undecidable in general. In the presence of termination, however, confluence and local confluence coincide according to Newman's Lemma [16]. Based on this information most automatic completion tools follow a common strategy: They maintain termination of an oriented version of an initial set of equations and try to make it locally confluent by means of deducing and adding new consequences. If this succeeds, at some point all critical pairs will be joinable and the tool yields a terminating and confluent term rewrite system, which has the same equational theory as the initial set of equations. There is very little work that bridges the gap between proof assistants and termination, the most notable being [13].

The aim of this paper is to use rewriting techniques to automatically derive a terminating and confluent first-order rewrite system for a proof assistant. In this research we considered HOL Light [8] and its Multivariate [9] library, as together with the Flyspeck project (developing a formal proof of the Kepler conjecture [7]) they form a large library of mathematical knowledge. The completion tool we used is KBCV [21], which also features an interactive mode and the generated completion proves may be certified by CeTA.

In the setting of proof assistants we work on higher-order terms with types whereas in the rewrite community most tools work on first-order term rewriting systems. A commonly used input format to termination as well as completion tools is the TPDB format[1] and its XML-based successor.[2]

We have implemented a translation mechanism from HOL Light to the TPDB format in order to give the theorems present in HOL Light/Multivariate to the available rewriting tools. To give the computed results back to HOL Light we implemented the converse translation from the TPDB format to typed λ-terms. We proposed an interaction model between HOL Light and KBCV and did initial experiments on deriving new theorems from the critical pairs. The initial set of 3,267 orientable equations has been passed in one go to KBCV and 305 thousand critical pairs were found which gave rise to 167 thousand HOL equations.

The rest of this paper is organized as follows. In Section 2 we describe how the theorems of HOL Light may be translated to first-order equations in the TPDB format. Following this we briefly recall completion in Section 3. Our main idea — the interaction between HOL Light and KBCV — is presented in Section 4. Next we present our experiments in Section 5. Finally we conclude in Section 6.

2 Translation from logic to rewriting

The first issue in implementing a translation mechanism from theorem statements to rewriting is to choose which theorem statements can be used in a rewriting setting. Modern completion tools only support unconditional equations, so we choose to work only with unconditional orientable equations that can be encoded in a first-order format.

We start with all the theorems available in HOL Light/Multivariate. To obtain a list of all these, we use the `update_database` functionality of HOL Light, which can produce a list of name–theorem pairs accessible from the top level by analyzing OCaml's internal data structures. We proceed by eliminating repetitions (theorems that have the same statement, but have been

[1] `https://www.lri.fr/~marche/tpdb/format.html`

[2] `http://www.termination-portal.org/wiki/XTC_Format_Specification`

assigned different names), and selecting only the theorems that have the form of unconditional equations (possibly universally quantified).

The two most common approaches for eliminating λ-expressions from higher-order logic formulas are to use λ-lifting or to encode them as combinators. Such approaches are often used to translate HOL to first-order logic [15, 10]. Lambda lifting does not seem to be possible, as in rewriting we are not able to quantify inside a term. As the theory of SKI combinators is equational, it might be possible to obtain a translation of λ-expressions to combinators. In our first experiments we chose to translate the λ-expressions which can be expressed as rewrite rules using simple transformations (β-reduce all redexes in the theorem statements and η-expand equalities that have a λ-expression on one of the sides using extensionality) and to ignore the rest of the equations.

Next we define the weight function, as a partial function on terms that returns polynomials over variables (we will ignore the equations for which the function is undefined). The function that we present ignores the types of the terms completely. In case of type-aware encodings the function would need to include the type variables in the resulting polynomial.

Definition 2.1 (weight of a higher-order logic term)**.**

$$w(t) := \begin{cases} 1 & \text{if } t \text{ is a constant} \\ x & \text{if } t \text{ is a variable } x \\ w(l) + w(r) & \text{if } t = l\,r \\ \textit{undefined} & \text{if } t = \lambda y.s \end{cases}$$

Given such a weight function, we can filter the orientable equations. We consider a HOL theorem as an orientable equation, if after specializing all the top level universally quantified variables the weights of the left- and right-hand sides of the equation are defined and one of them is strictly greater than the other in the usual polynomial order sense. Not all equations are orientable, for example the usual associativity or commutativity theorems have the same polynomials returned by w for both sides, so they are not orientable.

In order to eliminate the higher-order applications, we use the `apply` functor. We employ the algorithm introduced by Meng and Paulson [15]. For each higher-order constant c we compute the minimum arity n_c with which it appears in a problem, and the first n_c arguments are passed to c directly. If the constant is also used with more arguments in the problem, `apply` is used. Blanchette [4, p. 105–106] gives simple examples when this encoding introduces incompleteness in the encoding to ATP formats. Due to the lack of general quantifiers, however, this works quite well for rewriting.

We can now proceed to encode the equations in the TPDB format. A file in this format starts with a number of variable declarations, followed by a number of equations. To synchronize the symbols appearing in the theorems, the TPDB export declares the signature of the constants, functions and variables (for polymorphic constants or functions only one symbol for all occurrences; this could be strengthened with monomorphisation). The variables present in all the equations are written to the file, followed by the equations oriented in the direction implied by the weights. In order to verify our implementation of the polynomial ordering we proved the following theorem.

Theorem 2.1. *The theorems of HOL Light/Multivariate orientable by $w(t)$ and translated to first-order rewriting form a terminating TRS.*

Proof. We have used TTT2 to find a proof that the system is terminating. The automatic strategy is quite slow. Limiting TTT2 to polynomial interpretations (matrix interpretations of dimension

$$\begin{array}{llll}
\textsc{Deduce}\ \dfrac{(\mathcal{E},\mathcal{R})}{(\mathcal{E}\cup\{s\approx t\},\mathcal{R})}\ \text{if } s\ {}_{\mathcal{R}}\!\leftarrow u \rightarrow_{\mathcal{R}} t & & \textsc{Orient}\ \dfrac{(\mathcal{E}\cup\{s\ \dot{\approx}\ t\},\mathcal{R})}{(\mathcal{E},\mathcal{R}\cup\{s\rightarrow t\})}\ \text{if } s>t \\[2ex]
\textsc{Compose}\ \dfrac{(\mathcal{E},\mathcal{R}\cup\{s\rightarrow t\})}{(\mathcal{E},\mathcal{R}\cup\{s\rightarrow u\})}\ \text{if } t\rightarrow_{\mathcal{R}} u & & \textsc{Delete}\ \dfrac{(\mathcal{E}\cup\{s\approx s\},\mathcal{R})}{(\mathcal{E},\mathcal{R})} \\[2ex]
\textsc{Collapse}\ \dfrac{(\mathcal{E},\mathcal{R}\cup\{s\rightarrow t\})}{(\mathcal{E}\cup\{u\approx t\},\mathcal{R})}\ \text{if } s\stackrel{\sqsupset}{\rightarrow}_{\mathcal{R}} u & & \textsc{Simplify}\ \dfrac{(\mathcal{E}\cup\{s\ \dot{\approx}\ t\},\mathcal{R})}{(\mathcal{E}\cup\{u\ \dot{\approx}\ t\},\mathcal{R})}\ \text{if } s\rightarrow_{\mathcal{R}} u
\end{array}$$

Figure 1: The inference rules of *completion.*

1) over the naturals, however, is able to find a proof in a reasonable time (about 25 minutes). In particular the SMT prover invoked by TTT2 does not find a satisfiable assignment for a bit-width of 5, but finds one for a bit-width of 6. Parsing the system (consisting of a few thousand rewrite rules) takes the biggest part of the running time of TTT2. We used CeTA to certify that the proof found by TTT2 is indeed a valid termination proof of the system. CeTA requires about 10 minutes to check the proof. □

We have also implemented a combinator parser, which is able to read files generated by termination and confluence tools and output HOL Light preterms. When reading the TPDB file, applications give rise to `Combp` preterms and constants or variables give rise to `Varp` preterms. When exporting the HOL theorems as a TPDB file, we have declared a signature which is used to map the TPDB concepts (names of functions, constants, variables) back to their HOL counterparts. Such preterms can later be type-checked by the standard HOL Light term parser. Due to an encoding that does not preserve types, some of the preterms may fail to type-check (and as we will see in Section 5, some will fail). The rewrites performed on the KBCV side are not type-valid in the HOL setting, therefore such equations do not give rise to valid HOL critical pairs and can be forgotten.

Given a terminating rewrite system, we can proceed to completing the system.

3 Completion

We briefly recall the basics of completion. See for example [2] for a comprehensive introduction to completion and term rewriting.

Completion is a procedure which takes as input a (finite) set of equations $\mathcal{E}$ and optionally a reduction order $>$ (older tools need the reduction order in advance whereas modern tools try to construct the reduction order dynamically with the help of external termination tools, see [23]) and attempts to construct a terminating and confluent TRS $\mathcal{R}$ with the same equational theory as $\mathcal{E}$. Provided the completion procedure succeeds, two terms are equivalent with respect to $\mathcal{E}$ if and only if they reduce to the same normal form with respect to $\mathcal{R}$, that is, $\mathcal{R}$ represents a decision procedure for the word problem of $\mathcal{E}$.

The procedure generates a finite sequence of intermediate TRSs which constitute approximations of the equational theory of $\mathcal{E}$. Following Bachmair and Dershowitz [3] the completion procedure may be modeled as a system of inference rules (see Figure 1). These inference rules work on pairs $(\mathcal{E},\mathcal{R})$ where $\mathcal{E}$ constitutes a finite set of equations and $\mathcal{R}$ is a finite set of rewrite rules. The goal of the procedure is to transform an initial pair $(\mathcal{E},\varnothing)$ into a pair $(\varnothing,\mathcal{R})$ such that $\mathcal{R}$ is terminating, confluent and equivalent to $\mathcal{E}$. A completion procedure based on these rules may either succeed (find $\mathcal{R}$ after finitely many steps), loop indefinitely, or fail. In Figure 1

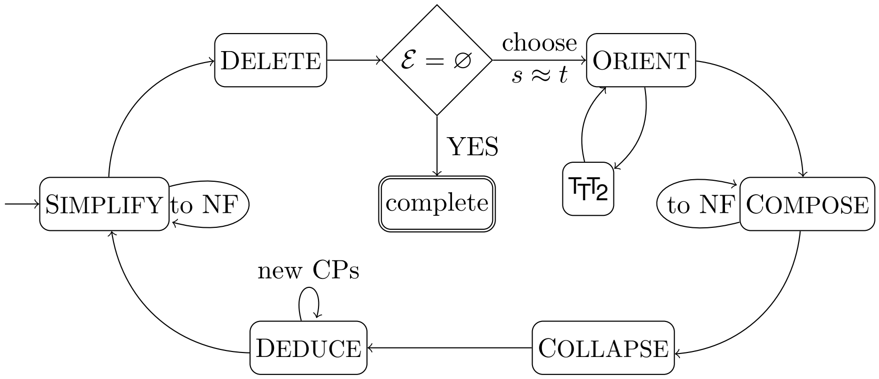


Figure 2: KBCV's automatic completion procedure.

a reduction order $>$ is provided as part of the input (whereas most modern completion tools construct this ordering on the fly as described above). We write $s \overset{\sqsupset}{\to}_{\mathcal{R}} u$ to express that s is reduced by a rule $\ell \to r \in \mathcal{R}$ such that ℓ cannot be reduced by $s \to t$. The notation $s \dot{\approx} t$ denotes either of $s \approx t$ and $t \approx s$.

In order to make the simplification set more complete we want to use an automatic completion tool. Most modern completion tools use some variant of the above inference system and then commit to a specific strategy to implement a completion procedure. There are several such tools available today, e.g., Slothrop [23], MKBTT [18], Maxcomp [11], and KBCV [21]. One of the main issues was that most of these tools have not been designed to handle problems with a magnitude counting thousands of equations. In the end we decided to use KBCV for several reasons:

- It has both an automatic and an interactive mode, where the completion inference rules may be applied freely.
- It records a history [19] of how rules were applied, which is useful to reprove the corresponding equations in HOL Light.
- Its parser is fast enough to load the thousands of equations of the problem at hand in reasonable time.
- One of the authors is the main developer of KBCV so it was relatively easy to adapt the tool and optimize it [20].

Tools like KBCV typically work on small problems, e.g., those which can be found in the TPDB problem database.[3] When given a problem KBCV tries to complete it by issuing the inference rules of completion in the order depicted in Figure 2.

First SIMPLIFY is used on all equations as long as a normal form with respect to the current TRS $\mathcal{R}$ is reached. Next all trivial equations, i.e., equations where the left- and right-hand sides are the same, are deleted. Now the tool checks whether $\mathcal{E}$ is empty, if this is the case $\mathcal{R}$ is complete and the procedure finishes. Otherwise KBCV chooses an equation which it will try to orient. The heuristic here is to select an equation with minimal left- and right-hand sides. The depicted procedure actually runs in two threads in parallel. The first of those always tries to orient equations from left to right and only if this does not succeed the other way round. The

[3] http://termination-portal.org/wiki/TPDB

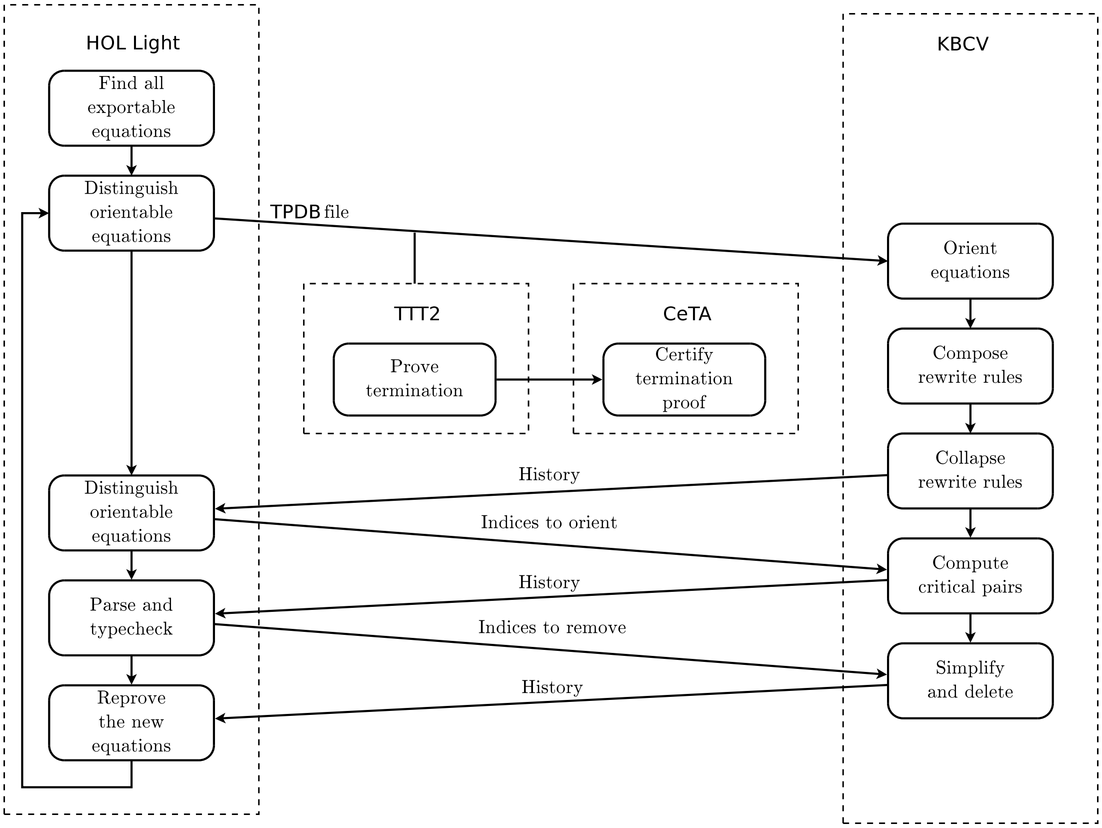


Figure 3: Control and data flow between HOL Light, KBCV, and external termination tools.

second thread behaves dually. KBCV implements a fast version of a lexicographic path ordering to orient equations. Only if this does not succeed it gives the problem to $\mathsf{T_TT_2}$ to establish a terminating system which also comprises the newly oriented equation. To keep the resulting system as small as possible KBCV now applies COMPOSE until all right-hand sides of rules are in normal form with respect to the current TRS $\mathcal{R}$. Next it also simplifies the left-hand sides of rules using COLLAPSE. Finally DEDUCE computes critical pairs between left-hand sides of rules and adds those to the set of equations.

4 Interaction between HOL Light and KBCV

In this section we describe how the steps performed in the completion procedure (described in the previous section) correspond to operations in HOL. For this we run KBCV in its interactive mode. The interaction is depicted schematically in Fig. 3.

Exporting the HOL Light simplification set. In Section 2 we have already described the export of HOL Light equations to the TPDB format. We export all the equations that can be written in the first-order rewriting format. We separate the orientable ones from the non-orientable ones but write the latter as well (as they may become orientable after simplification).

Importing the rewrite system to KBCV. KBCV can directly read the TPDB file. We issue the `orient` command to orient the part of the system which could be oriented in HOL. Because we already know that our chosen orientation will be terminating we set KBCV to use an external termination tool that will always output "YES".

Composing the rewrite rules. Given that the right-hand side of a rewrite rule can be simplified with another rule the same operation can be performed in HOL assuming that the types are correct. This means that we will later have to check that the derived rules correspond to provable equations in HOL Light.

Collapse of rewrite rules. This operation works in a similar way as compose, only that it produces equations which we will have to give back to HOL Light to orient and prove terminating.

Deducing critical pairs. A critical pair is a derivable equation which arises from the overlap of left-hand sides of two rules. Some of those pairs found by KBCV will not be well-typed. To improve the performance of SIMPLIFY we can remove such ill-typed pairs by parsing and type-checking them in HOL Light. In our experiments we decided to use type-checking rather than encoding of types in terms for two reasons: First, an ill-typed pair may give rise to a well-typed equation using unification. For example an equation where we have $list(\alpha)$ on one side and $list(num)$ on the other side will give a well-typed equation by instantiating α to num. Second, by throwing away not well-typed pairs we may remove equations that are needed to preserve confluence. We have already discussed at the end of Section 2, why such pairs can be forgotten. completion procedure.

Simplify and Delete. Simplify rewrites the left- and right-hand sides of equations to normal form in order to eliminate joinable critical pairs. At this point the information about newly obtained equations to orient can be send back to HOL together with their recorded history which allows to reprove them.

Now we have a somehow more complete approximation of the initial theorems in HOL Light for which we want to repeat the loop until we arrive at a complete system.

5 Experiments

We did our experiments with HOL Light revision 153 from December 2012 and Flyspeck revision 3130 from March 2013. The experiments were performed on a 48-core server with AMD Opteron 6174 2.2 GHz CPUs, 320 GB RAM, and 0.5 MB L2 cache per CPU. HOL Light uses only one process, whereas KBCV uses threads to exploit all the available cores.

The number of all theorems as obtained using the `update_database` mechanism is 17,807. After removing repetitions and considering only the universally quantified equations there are 6,273 theorems. Our weight function returns a defined value for 4,186 theorems. Our reduction order divides these in two parts: the 3,267 equations that are orientable and 919 that are not.

To verify our heuristic order, we used $\mathsf{T_TT_2}$. We have exported the orientable equations in the TPDB format and having used $\mathsf{T_TT_2}$ with polynomial interpretations, we have found a proof. The proof is 2.6MB in size and we have used CeTA to certify it.

We next proceeded by loading the TPDB file in KBCV. Since we already have established termination of these rules beforehand orienting them only takes a view seconds. Deducing all critical pairs between the left-hand sides of the oriented equations proved to be the first hurdle for KBCV. The previous version of KBCV did not finish the computation of critical pairs in a few days. To speed it up we introduced parallelization. Now KBCV is able to compute all 305,708 critical pairs in about two hours.

The critical pairs can be exported together with their history. As described in Section 2, we have implemented a combinator parser to read back the KBCV history output as HOL Light terms. Only the well-typed ones can be properly parsed. Trying to parse the terms takes about 4 hours and 137,888 of the newly generated equations are ill-typed. The indices of the ill-typed equations are passed back to KBCV, which can in turn remove them.

The next bottleneck was SIMPLIFY. Using the previous version of KBCV, simplification of this big number of equations was infeasible. With the help of caching techniques and parallelization we are able to delete the joinable critical pairs; this, however, takes a few days. The remaining step to close the cycle is to reprove the equations obtained using rewriting techniques in HOL Light. With the history of performed simplifications, this is straightforward but has not been done yet. So far we have performed only one iteration, so the TRS is not a confluent one.

Two example critical pairs found by KBCV are (the numbers are the internal indices used to reference equations in KBCV, e.g., 309551 is a consequence of theorems 49 and 882 from the initial set):

```
309551 : i(i(realu_lt(), i(i(realu_add(), x), y)), i(realu_ofu_num(), u_0())) =
  i(i(realu_lt(), x), i(realu_neg(), y)) : 309551 : 49, 882,
```

which corresponds to the equation $x + y < 0 \iff x < -y$ and

```
309569 : i(i(realu_lt(), i(i(realu_add(), x), y)), i(Re(), ii())) =
  i(i(realu_lt(), x), i(realu_neg(), y)) : 309569 : 139, 882,
```

which corresponds to $x + y < \mathsf{Re}(i) \iff x < -y$. The latter of the two theorems will be simplified using the rule that rewrites $\mathsf{Re}(i)$ to 0 and will be discarded.

6 Conclusion

This paper presents initial experiments in automatically deriving a terminating and confluent simplification set for HOL Light using tools coming from termination and completion research. We started with all the (unconditional) equations present in HOL Light/Multivariate and using a manually defined order we proved termination for a large subset of the rules. We have presented a possible loop for deriving confluence and we have done some experiments with the first loop of the confluence derivation.

The simplification set that we derive, includes a number of equations translated to first-order logic. Because higher-order matching is used for rewriting in most proof assistants, the properties of the simplification set (like termination or confluence) derived for first-order translations do not immediately give rise to the same properties for the original rules. There are at least two ways to proceed in order to preserve the termination and confluence for the obtained HOL simplification set:

- Use first-order rewriting in the proof assistant;
- Further restrict the initial simplification set to the first-order theorems.

In the future, the first thing we intend to do is to automatically prove the equations derived by KBCV in HOL. Thanks to recording completion we know the equations used to derive the new ones, so we have all the necessary components to use the existing HOL Light decision procedures.

This will complete the cycle presented in Section 4 and will allow executing more iterations of the completion procedure. Further, the remaining efficiency bottlenecks (on both HOL Light and KBCV sides) need to be taken care of to make the procedure practical.

The approach that we presented is applicable not only to the large HOL theorem set, but also to an arbitrary subset of it. For example, one might consider rewrite rules concerning one specific domain of mathematics. In case completion for the whole set turns out to be unachievable (for such a big set it might not be possible to iterate the loop until the result is stable), the approach can be applied to the particular area, automatically deriving a decision procedure.

We intend to try out different encodings to rewriting, that would take types into account. Certain approaches presented for example in [5] could be directly applicable. Next, extensions to rewriting, like higher-order rewriting [14], conditional rewriting, or AC rewriting can be considered. Finally, we intend to investigate, how useful the automatically derived simplification set is for proving real HOL Light problems.

References

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, et al. *The Coq Proof Assistant Reference Manual Version 8.4*. LogiCal project, 2012. `http://coq.inria.fr/refman/`.

[2] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[3] Leo Bachmair and Nachum Dershowitz. Equational inference, canonical proofs, and proof orderings. *Journal of the ACM*, 41:236–276, 1994.

[4] Jasmin Christian Blanchette. *Automatic Proofs and Refutations for Higher-Order Logic*. PhD thesis, Fakultät für Informatik, Technische Universität München, 2012. `http://www21.in.tum.de/~blanchet/phdthesis.pdf`.

[5] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. Encoding Monomorphic and Polymorphic Types. In *TACAS*, 2013. To appear, `http://www21.in.tum.de/~blanchet/enc_types_paper.pdf`.

[6] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with AProVE. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004.

[7] Thomas C. Hales. Introduction to the Flyspeck project. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[8] John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *FMCAD*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[9] John Harrison. The HOL Light theory of euclidean space. *J. Autom. Reasoning*, 50(2):173–190, 2013.

[10] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *CoRR*, abs/1211.7012, 2012.

[11] Dominik Klein and Nao Hirokawa. Maximal completion. In Manfred Schmidt-Schauß, editor, *Proc. of the 22nd International Conference on Rewriting Techniques and Applications (RTA 2011)*, volume 10 of *Leibniz International Proceedings in Informatics*, pages 71–80. Schloss Dagstuhl, Dagstuhl, 2011.

[12] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In Ralf Treinen, editor, *RTA*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009.

[13] Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle functions via termination of rewriting. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *ITP*, volume 6898 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2011.

[14] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theor. Comput. Sci.*, 192(1):3–29, 1998.

[15] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[16] Maxwell H. A. Newman. On theories with a combinatorial definition on “equivalence”. In *The Annals of Mathematics*, pages 223–243. Annals of Mathematics, 1942.

[17] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[18] Haruhiko Sato, Sarah Winkler, Masahito Kurihara, and Aart Middeldorp. Multi-completion with termination tools (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proc. of the 4th International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 306–312. Springer, 2008.

[19] Thomas Sternagel. Automatic proofs in equational logic. Master’s thesis, University of Innsbruck, 2012. `http://cl-informatik.uibk.ac.at/teaching/master/theses/TS.pdf`.

[20] Thomas Sternagel. KBCV 2.0 – Automatic Completion Experiments, 2013. Accepted for publication at IWC 2013, `http://cl-informatik.uibk.ac.at/users/csag9384/publications/iwc_2013_paper.pdf`.

[21] Thomas Sternagel and Harald Zankl. KBCV - Knuth-Bendix completion visualizer. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 530–536. Springer, 2012. `http://cl-informatik.uibk.ac.at/users/csag9384/publications/ST12_02.pdf`.

[22] René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009.

[23] Ian Wehrman, Aaron Stump, and Edwin Westbrook. Slothrop: Knuth-Bendix completion with a modern termination checker. In *Proc. of the 17th International Conference on Rewriting Techniques and Applications (RTA 2006)*, pages 287–296. Springer, 2006.

Stronger Automation for Flyspeck by Feature Weighting and Strategy Evolution

Cezary Kaliszyk and Josef Urban

[1] University of Innsbruck, Austria
[2] Radboud University, Nijmegen

Abstract

Two complementary AI methods are used to improve the strength of the AI/ATP service for proving conjectures over the HOL Light and Flyspeck corpora. First, several schemes for frequency-based feature weighting are explored in combination with distance-weighted k-nearest-neighbor classifier. This results in 16% improvement (39.0% to 45.5% Flyspeck problems solved) of the overall strength of the service when using 14 CPUs and 30 seconds. The best premise-selection/ATP combination is improved from 24.2% to 31.4%, i.e. by 30%. A smaller improvement is obtained by evolving targetted E prover strategies on two particular premise selections, using the Blind Strategymaker (BliStr) system. This raises the performance of the best AI/ATP method from 31.4% to 34.9%, i.e. by 11%, and raises the current 14-CPU power of the service to 46.9%.

1 Introduction

Methods for automated reasoning in large theories (ARLT) have started to develop in the recent years [8,14,20,24]. The primary driving force behind this development is the growing use of such methods for assisting ITPs like Isabelle [15] and Mizar [22, 23]. Recently, we have added HOL Light [7] to the pool of systems linked to the large-theory ATP methods [11], and experimented with the strongest and most orthogonal combinations of the premise-selection methods and various ATPs over the Flyspeck corpus [6]. The experimental work, described in [10], has shown that 39% of the 14185 Flyspeck theorems could be proved in a push-button mode (without any high-level advice and user interaction) in 30 seconds of real time on a fourteen-CPU workstation.

The work described in this paper improves two aspects of large-theory reasoning done on the Flyspeck corpus: (i) the premise selection methods, i.e., selecting from a large repository the lemmas, theorems, and definitions that are most relevant for a new conjecture, and (ii) the ATP strategies used to solve the problems after premise selection. The techniques used to achieve these improvements are quite straightforward. In the first case (Section 2), the improvement is obtained by better weighting (scaling) of the large number of features that are used as an input to the machine-learning algorithms that learn premise selection from previous proofs. Such feature weighting seems to be quite important particularly for the k-nearest-neighbor algorithm that we initially started to try on Flyspeck in [10]. Feature weighting seems to be a reasonably studied subject in the machine-learning community, in particular in the information-retrieval domain, and in some sense this work is quite a straightforward application of those studies. The result is however quite a surprising improvement over the best AI/ATP method used so far for Flyspeck, and also quite high improvement of the joint power of all AI/ATP methods used. An important practical advantage of using k-nearest-neighbor over the more sophisticated kernel methods explored recently on smaller corpora [1,13] is that k-nearest-neighbor works quite fast with the number of features and training examples that the Flyspeck corpus provides, while scaling up the kernel methods to the Flyspeck sizes is still work in progress.

The second improvement is obtained by a straightforward running of the newly developed BliStr (Blind Strategymaker) strategy-evolving system [21] on two classes of ATP problems that are created by different premise-selection methods. We have already been using for Flyspeck a custom strategy-scheduling version (Epar) of the E [17] prover, consisting of strategies developed by BliStr for the Mizar@Turing competition [18] on the 1000 Mizar@Turing training problems. While that version turned out to be significantly stronger than standard E on Flyspeck, it was still optimized on problems coming from a different corpus (Mizar). These Mizar problems were additionally quite small in comparison to the sizes of Flyspeck problems produced by the premise-selection methods that are most useful for Flyspeck. Section 3 shows that several hundred runs of the BliStr's strategy-evolving loop (slightly extended in comparison to [21], to also evolve further SInE-based [8] premise-pruning parameters) on these Flyspeck problem classes can again raise the performance of E quite considerably. The additional improvement of the overall power of the system is smaller than for the first method, but it is still quite significant, and more power can likely be added in the future by using strategy evolution also for the remaining important classes of Flyspeck problems.

2 Better Feature Weights for Nearest Neighbor

Premise selection is an essential AI component that has in the past decade allowed the usage of automated theorem provers (ATPs) over large corpora built with ITPs such as Mizar, Isabelle, and HOL Light. The premise-selection methods select from the large repositories the lemmas, theorems, and definitions (i.e., the *premises*) that are most relevant for a new conjecture. This is a hard AI problem, for which various heuristics taking into account the semantics and syntax of mathematical formulas can be considered. Such heuristics can involve (possibly approximative) deductive reasoning components. For example, the MoMM system [19] generalizes hundreds of thousands of existing mathematical lemmas, and using (deductively correct) type-aware ATP indexing methods combined with limited deductive reasoning tries to find suitable lemmas for a new conjecture. The SInE [8] heuristic is based on approximative reasoning about formulas, using only the symbols contained in the formulas. In some sense it carves out the set of formulas that (particularly in Horn-like ontologies) may be transitively related to the conjecture. A much less deductive way of selecting premises is to learn (use machine learning) what is relevant for what from the proofs contained already in the large repositories [1,13,20]. For this, the formulas are characterized by suitably chosen features that can be purely syntactic, such as the symbols and terms occuring in the formulas, or by more semantic/deductive features, such as models and abstractions of the formulas and relations between them. The machine learners then try to learn the association from such features to the premises that were most useful for proving the theorems. The key parts of such methods are therefore the learning (generalization) algorithms and the features used by the algorithms.

In order to get additional premise-selection power for the first large-scale AI/ATP experiments done over Flyspeck, we had quickly added in [10] a custom implementation of the k-nearest neighbor (k-NN) machine-learning method, which computes for a new example (conjecture) the k nearest (in a given feature distance) previous examples and ranks premises by their frequency in these examples. The motivation was that this fast ("lazy" and trivially incremental) learning method can be easily parametrized and might for some parameters behave quite differently from the naive Bayes learner, which we had been using exclusively until then, because the newly developed kernel-based methods [1, 13] so far do not scale to such large corpora. Our (distance-weighted [4]) implementation weighs the contribution of the k nearest neighbors by their feature-based similarity to the current conjecture, and computes the overall ranking of the available premises as a sum of the premises' weighted contributions from these k neighbors.

The performance of this (multi-class, distance-weighted) k-NN seemed reasonable (the best k-NN method solved 21.7% of the Flyspeck problems when combined with E) and quite complementary to premise selection based on naive Bayes. However, k-NN was weaker than the best naive-Bayes classifier (24.1% of problems solved when combined with E). The features that we use for characterizing Flyspeck formulas and measuring their similarity with k-NN consist of the symbols and normalized shared terms contained in the Flyspeck formulas. To give a concrete example (taken from Section 4.1 of [10]), the set of features characterizing the HOL theorem `DISCRETE_IMP_CLOSED`:[1]

```
∀s:real^N→bool e.
        &0 < e ∧ (∀x y. x IN s ∧ y IN s ∧ norm(y − x) < e ⟹ y = x)
        ⟹ closed s
```

is the following set of strings:

```
"real", "num", "fun", "cart", "bool", "vector_sub", "vector_norm",
"real_of_num", "real_lt", "closed", "_0", "NUMERAL", "IN", "=", "&0",
"&0 < Areal", "0", "Areal", "Areal^A", "Areal^A - Areal^A",
"Areal^A IN Areal^A->bool", " Areal^A->bool", "_0", "closed Areal^A->bool",
"norm (Areal^A - Areal^A)", "norm (Areal^A - Areal^A) < Areal"
```

Here `real` is a type constant, `IN` is a term constructor, `Areal^A->bool` is a normalized type, `Areal^A` its component type, `norm (Areal^A - Areal^A) < Areal` is an atomic formula, and `Areal^A - Areal^A` is its normalized subterm.

The simplest way how to measure the similarity of formulas to the new conjecture is to compute the overlap of their (sparse) feature vectors. One known property [2] of the k-NN that was neglected by our first implementation is however the sensitivity of k-NN to feature frequencies. For example, without additional weighting, the most common symbol (e.g., equality) has in such similarity function the same weight as the most rare symbol, which is clearly not desirable: overlap on the rarest symbol is much more significant than overlap on a symbol that is present everywhere.

The most common way how to weight (boolean-counted) features in text retrieval with respect to their frequency is the IDF (inverse document frequency) scheme [9]. This scheme weights a term t in a collection of documents D using the logarithm of the inverse of the term's frequency in the document collection:

$$\mathrm{IDF}(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

For example, a term (symbol) contained only in one document (formula) will have weight $\log |D|$, a term contained in all of them will have weight $log(1) = 0$, and a term contained in half of the documents will have weight $log(2) = 1$. Apart from using the standard IDF, we have also found useful two other IDF-based weighting schemes, the (smoothed) inverse frequency:

$$\mathrm{IDF}_1(t, D) = \frac{1}{1 + |\{d \in D : t \in d\}|}$$

and quadratically scaled (smoothed) inverse frequency:

$$\mathrm{IDF}_2(t, D) = \frac{1}{(1 + |\{d \in D : t \in d\}|)^2}$$

[1] http://mws.cs.ru.nl/~mptp/hol-flyspeck/trunk/Multivariate/topology.html#DISCRETE_IMP_CLOSED

These three feature weighting schemes were combined with different values of k nearest neighbors and (as usual) with ATPs run on different number of top-rated premises (using 30s time limit and the same hardware as in [10]). Table 1 shows the performance of the best 12 methods after these experiments, together with the performance of the best previous method (naive_bayes_0154e). The precision of the logarithmically scaled IDF is clearly the best, and in comparison with the non-logarithmic scaling, only relatively few nearest neighbors are needed. The previously best method (last line in Table 1) is improved by 103 problems, which is a 30% improvement. The proof data (denoted as ATP2 in Table 2) used for training the k-NNs were by a negligible margin older (worse, i.e., containing more irrelevant proof dependencies that we eventually prune) than the proof data (ATP3) used for the previously best method. The feature extraction method was the same (the standard one).

Table 1: The top 12 AI/ATP methods, together with the previously best.

Method	Premises	Prover	Theorem (%)	Σ-SOTAC	Processed
40-NN_IDF	128	E	446 (31.43)	4.24	1419
80-NN_IDF	512	E	445 (31.36)	4.25	1419
40-NN_IDF	512	E	443 (31.22)	4.20	1419
80-NN_IDF	128	E	436 (30.73)	4.22	1419
160-NN_IDF$_1$	128	E	423 (29.81)	3.64	1419
300-NN_IDF$_1$	128	E	422 (29.74)	3.50	1419
40-NN_IDF	512	V	419 (29.53)	3.09	1419
760-NN_IDF$_1$	128	E	417 (29.39)	3.16	1419
160-NN_IDF$_1$	128	E	417 (29.39)	3.49	1419
40-NN_IDF	128	V	416 (29.32)	3.01	1419
1200-NN_IDF$_1$	128	E	416 (29.32)	4.29	1419
1000-NN_IDF$_2$	128	E	415 (29.25)	3.21	1419
naive_bayes	154	E	343 (24.17)	1.97	1419

Method: 40-NN_IDF means that 40 nearest neighbors are used with the standard IDF weighting.

Prover: V stands for Vampire [16].

Σ-SOTAC For each problem P solved by a system, its SOTAC for P is the inverse of the number of systems that solved P. Σ-SOTAC is the sum of a system's SOTAC over all problems.

Table 2 shows the newly computed 14-long *greedy covering sequence*, i.e., the joint performance of the (greedily) best combination of 14 methods, ordered by their inclusion in the greedy algorithm. While the logarithmic IDF scaling is obviously at the top, the linearly-scaled IDF provided many useful complementary predictive methods. The overall 14-method coverage went up from 39.0% (Table 14 in [10]) to 45.45%, i.e., by 16%. The frequency-scaling code in the k-NN implementation responsible for this improvement takes about 5 lines of Perl, and the whole sparse distance-weighted multiclass k-NN implementation takes about 200 lines of Perl code. Some large improvements in the ARLT domain are still very easy.

3 Better Strategies for Different Premise Selections

BliStr (Blind Strategymaker) [21] is a recently developed system that automatically develops strategies for E prover on a large set of related problems. Its main idea is to interleave (i)

Table 2: The greedy sequence including the new k-NNs.

Method	Premises	Prover	Training data	Sum %	Sum
40-NN_IDF	128	E	ATP2	31.43	446
760-NN_IDF$_1$	128	V	ATP2	35.09	498
naive_bayes	128	Z3	ATP4	37.27	529
760-NN_IDF$_1$	32	Z3	ATP2	38.97	553
naive_bayes	184	E	ATP3	40.31	572
naive_bayes	12	E	ATP3	41.22	585
160-NN_IDF$_1$	128	Z3	ATP2	42.07	597
naive_bayes	512	E	ATP0+HOL0	42.91	609
80-NN_IDF	512	V	ATP2	43.48	617
760-NN_IDF$_1$	512	E	ATP2	43.97	624
40-NN_IDF$_1$	32	V	ATP2	44.39	630
1000-NN_IDF$_2$	740	V	ATP2	44.74	635
40-NN_IDF	32	E	ATP2	45.10	640
40-NN	32	Z	ATP2	45.45	645

iterated low-timelimit local search for new strategies on small sets of similar easy problems with (ii) higher-timelimit evaluation of the new strategies on all problems. The accumulated results of the global higher-timelimit runs are used to define and evolve the notion of "similar easy problems", and to control the selection of the next strategy to be improved.

BliStr was used to grow a set of E strategies for the Mizar@Turing competition.[2] The final improvement of the resulting strategy-scheduler (Epar) over the E's auto-mode was 25% on the Mizar@Turing competition problems. Epar has already been used for practically all AI/ATP experiments over Flyspeck, i.e., whenever we refer to E above, it was run using the Epar strategy-scheduler.

Since the Mizar@Turing pre-competition training problems were quite small (ca. 25 premises per problem), while the best methods in Table 2 use 128 premises, it seemed potentially rewarding to automatically evolve E strategies on such Flyspeck problem classes instead. This has been so far tried for the two top problem classes (i.e., classes of problems generated using the particular premise selection method described in the table) from Table 2 that use E: 40-NN_IDF_128_ATP2 and naive_bayes_184_ATP3. For each of them, the set of (randomly chosen) 1419 Flyspeck problems was further randomly divided into a training part (800 problems) and a testing part (619 problems). The 800 training problems were then used for ca. 30 hours of parallelized strategy evolution with Blistr. Since a major factor in ATP efficiency over problems with many premises is also a good SInE pre-selection,[3] the E parameters tunable by BliStr were extended in comparison to [21] to also evolve the SInE parameters. The starting set of strategies is the same for both problem sets. These were 15 strategies previously developed by BliStr that were giving good performance on Mizar/MPTP problems. These strategies were however slightly weaker than the old Epar, because SInE parameters have been heuristically added to Epar, while the 15 strategies do not contain any SInE parameters. We left it to the extended BliStr to develop good SInE parameters.

For 40-NN_IDF_128_ATP2, the 15 initial strategies cover (in 10s) 257 of the 800 training

[2] http://www.cs.miami.edu/~tptp/CASC/J6/Design.html#CompetitionDivisions

[3] SInE-based selection is obviously interacting in various ways with the premise selection done by naive Bayes and k-NN. This typically turns out to be a fruitful interaction of two different ranking methods, see [13].

problems. 12 BliStr instance were run in total (on a 32-core Intel machine), producing 353 strategies eventually covering 320 of the 800 training problems when run with a 10s time limit. For naive_bayes_184_ATP3, the 15 initial strategies cover (in 10s) 215 of the 800 training problems. 16 BliStr instances were run in total (on a 64-core AMD machine), producing 637 strategies, covering 253 of the 800 training problems with 10s time limit. To construct a new set of Epar strategies, we used again greedy algorithm that chooses the 14 strategies with the (greedily) best joint coverage. Note that using for example Minisat++ [5] for solving the set-cover problem optimally is not easy: it takes 400s for the 353 strategies (finding an 18-cover), and Minisat++ did not finish within one hour for the 637 strategies. The 14 new greedy strategies for 40-NN_IDF_128_ATP2 cover 313 training problems, and the 14 naive_bayes_-184_ATP3 strategies cover 245 training problems. The new version of Epar (marked as E2 below) was then for each problem class evaluated with 30s overall time limit on the 64-CPU AMD machine both on the training and testing problems. The results are shown in Table 3 and Table 4. Note that in both cases the performance of the new Epar is obviously lower than the combined 10s performance of its underlying 14 strategies, because within its 30s overall time limit Epar gives only 2s to each of its strategies (this could obviously be made smarter, as is done in Vampire [16]).

Table 3: The strategy evaluation for 40-NN_IDF_128_ATP2.

40-NN_IDF_128_ATP2	OldEpar	NewEpar	Improvement (%)
Training	254	290	36 (14.2)
Testing	192	209	17 (8.9)
Total	446	499	53 (11.9)

Table 4: The strategy evaluation for naive_bayes_184_ATP3.

naive_bayes_184_ATP3	OldEpar	NewEpar	Improvement (%)
Training	173	208	35 (20.2)
Testing	132	162	30 (22.7)
Total	305	370	65 (21.3)

The tables seem to suggest that some overfitting on the training problems took place for 40-NN_IDF_128_ATP2, while the new Epar testing performance on naive_bayes_184_ATP3 is even better than its performance on the training problems. It is interesting to note that SInE is used in 11 of the 14 new 40-NN_IDF_128_ATP2 strategies, and 12 of the 14 new naive_bayes_-184_ATP3 strategies. Quite often the SInE parameters in the later complementary methods are quite severe, limiting the SInE recursion to 1 or 2. This could mean that the BliStr's loop run separately for the current premise-selection slice without interaction with other premise slices causes quite a lot of incompleteness. This might however be destroying some complementarity with the other premise-selection methods specialized in low premise numbers. An obvious remedy would be to evolve the strategies together on merged problem classes, and only later select the best strategies for the particular classes in a way that provides minimal overlap between them.

Finally, the new 14-long greedy covering sequence using the new Epars for 40-NN_IDF_-128_ATP2 and naive_bayes_184_ATP3 was computed on our main evaluation machine. The result is shown in Table 5. While originally the new Epar developed for `naive_bayes_184_ATP3` was indeed second in the greedy sequence, in the final version it became completely redundant, because we have also run the new strategies grown for `40-NN_IDF_128_ATP2` on the `440-NN_IDF` slices. This strengthened the `440-NN_IDF_128_ATP2` slice by 46 problems (12% improvement), and the `440-NN_IDF_512_ATP2` slice by 102 problems (33% improvement). This improvement was again quite unexpected, and it suggests that the overfitting suspected in Table 3 is not really a problem. The 14 methods of the final combination solve without any user interaction 46.86% of the Flyspeck problems.

Table 5: The new greedy sequence including the new Epars.

Method	Premises	Prover	Training data	Sum %	Sum
40-NN_IDF	128	E2	ATP2	34.88	495
440-NN_IDF	128	E2	ATP2	38.33	544
440-NN_IDF	512	E2	ATP2	40.02	568
760-NN_IDF$_1$	32	Z3	ATP2	41.64	591
naive_bayes	128	V	ATP2	42.84	608
40-NN	512	Z	ATP2	43.55	618
1000-NN_IDF$_2$	740	V	ATP2	44.11	626
naive_bayes	64	E	ATP3	44.67	634
160-NN_IDF$_1$	512	Z3	ATP2	45.10	640
naive_bayes	32	Z3	ATP0+HOL0	45.52	646
naive_bayes	512	E	ATP0+HOL0	45.94	652
40-NN	32	E	ATP2	46.30	657
80-NN_IDF	512	V	ATP2	46.58	661
160-NN_IDF$_1$	128	V	ATP2	46.86	665

Prover: E2 is the new Epar, while is the old Epar. Z3 is the Z3 [3] SMT solver.

Training data: ATP$_i$ is worse (older, less pruned) than ATP$_{i+1}$. ATP0+HOL0 is a combination of proof data obtained from ATPs with the proof data extracted directly by tracking proof dependencies inside HOL.

4 Conclusion and Future Work

It is interesting that a similar treatment of features as in processing of natural language texts helps so significantly also for formal mathematical libraries. While mathematics in its extreme is undecidable, and has (theoretically constructed) parts that provably behave as random, it is clear that human-organized mathematics is very far from such randomness. While the automation that we currently have is still very far from the power of human mathematicians, the observance of such simple statistical laws may be providing evidence that the Penrose-style arguments about the "necessarily super-Turing" power of the human mathematical mind might more and more look like just another case of the *AI effect*.[4] The obvious future work concerning feature weighting is to re-run the machine learning methods tested in [13] on the MPTP2078 benchmark (in particular, the kernel-based MOR, and van Laarhoven's BiLi, whose

[4] http://en.wikipedia.org/wiki/AI_effect

random-projection mechanism should be quite sensitive to feature distribution) with such feature weightings.

Since ATPs are in some sense universal problems solvers, it should not be very surprising that their parameterization matters a lot. However, some of the results, like the 33% improvement of the `440-NN_IDF_512_ATP2` slice (which was not subject to any direct tuning), were quite unexpected. While we have added the SInE parameters to BliStr, there are still many more E parameters that could be further tuned. A particularly interesting challenge is to grow and guess suitable term orderings for similar classes of problems.

In general the experimental results show that large improvements can be still achieved in the ARLT domain by quite simple methods and simple technology transfer. It seems that this AI domain is still wide open to a number of techniques that could further considerably improve the strenth of automation for large-theory mathematics.

5 Acknowledgments

While we have been for long time without much action discussing experiments with various state-of-the-art feature preprocessing methods, it was Jasmin Blanchette and his immediate interest in improving the machine learning for Sledgehammer [12] that led the second author to propose and quickly implement feature weighting in k-NN as a reasonable remedy to some of the problems discussed by Jasmin. While the TF-IDF scaling is reasonably known in the text retrieval domain, the idea of using the inverse document frequency as a formula classifier is in some sense also at the heart of the SInE axiom-selection heuristic [8] developed independently by Kryštof Hoder.

References

[1] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise Selection for Mathematics by Corpus Analysis and Kernel Methods. *CoRR*, abs/1108.3446, 2011.

[2] Fabrice Colas and Pavel Brazdil. Comparison of svm and some older classification algorithms in text classification tasks. In Max Bramer, editor, *Artificial Intelligence in Theory and Practice*, volume 217 of *IFIP International Federation for Information Processing*, pages 169–178. Springer US, 2006.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[4] Sahibsingh A. Dudani. The distance-weighted k-nearest-neighbor rule. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-6(4):325–327, 1976.

[5] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[6] Thomas C. Hales. Introduction to the Flyspeck project. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[7] John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *FMCAD*, volume 1166 of *LNCS*, pages 265–269. Springer, 1996.

[8] Krystof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 299–314. Springer, 2011.

[9] Karen Sprck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[10] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *CoRR*, abs/1211.7012, 2012.

[11] Cezary Kaliszyk and Josef Urban. Automated reasoning service for HOL Light. In *CICM*, volume 7961 of *Lecture Notes in Artifical Intelligence*, pages 120–135. Springer, 2013.

[12] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine Learning for Sledgehammer. 2013. Accepted to ITP 2013.

[13] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 378–392. Springer, 2012.

[14] Lawrence C. Paulson and Jasmin Blanchette. Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. In *8th IWIL*, 2010. Invited talk.

[15] Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.

[16] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[17] Stephan Schulz. E - A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002.

[18] Geoff Sutcliffe. The 6th IJCAR automated theorem proving system competition - CASC-J6. *AI Commun.*, 26(2):211–223, 2013.

[19] Josef Urban. MoMM - fast interreduction and retrieval in large libraries of formalized mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.

[20] Josef Urban. An Overview of Methods for Large-Theory Automated Theorem Proving (Invited Paper). In Peter Höfner, Annabelle McIver, and Georg Struth, editors, *ATE Workshop*, volume 760 of *CEUR Workshop Proceedings*, pages 3–8. CEUR-WS.org, 2011.

[21] Josef Urban. BliStr: The Blind Strategymaker. *CoRR*, abs/1301.2683, 2013.

[22] Josef Urban, Piotr Rudnicki, and Geoff Sutcliffe. ATP and presentation service for Mizar formalizations. *J. Autom. Reasoning*, 50:229–241, 2013.

[23] Josef Urban and Geoff Sutcliffe. Automated reasoning and presentation support for formalizing mathematics in Mizar. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors, *AISC/MKM/Calculemus*, volume 6167 of *LNCS*, pages 132–146. Springer, 2010.

[24] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. MaLARea SG1 - Machine Learner for Automated Reasoning with Semantic Guidance. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 441–456. Springer, 2008.

Extended Resolution as Certificates for Propositional Logic

Chantal Keller

LIX and INRIA Saclay–Île-de-France at École Polytechnique, Palaiseau, France
Chantal.Keller@inria.fr

Abstract

When checking answers coming from automatic provers, or when skeptically integrating them into proof assistants, a major problem is the wide variety of formats of certificates, which forces to write lots of different checkers. In this paper, we propose to use the extended resolution as a common format for every propositional prover. To be able to do this, we detail two algorithms transforming proofs computed respectively by tableaux provers and provers based on BDDs into this format. Since this latter is already implemented for SAT solvers, it is now possible for the three most common propositional provers to share the same certificates.

1 Introduction

Different theorem provers can communicate to benefit from each other capabilities. It is the case for instance when *automatic theorem provers*, which can prove efficiently even hard problems, cooperate with *interactive theorem provers*, well known for their trustworthiness. The powerful Isabelle [23] tactic `sledgehammer` [22] implements such an interaction, by calling in parallel various kinds of automatic provers to solve Isabelle goals.

To take advantage of efficiency without compromising soundness, the cooperation must be *skeptical*: in addition to a yes/no answer, the automatic prover must return a *proof witness* that can be checked or reconstructed in the proof assistant. In the `sledgehammer` tactic, this is for instance the case for the SMT solver Z3 whose proof witnesses are reconstructed to produce Isabelle theorems [4].

In addition to the fact that most automatic provers do not give (detailed enough) proof witnesses and thus must be taken at face value, the ones that do provide such witnesses all implement their own formats to prove the validity or the unsatisfiability of a given formula. It thus requires much effort to write a checker for a new prover, even when some already exist for other tools.

Besson *et al.* [3] proposed a format for the particular case of SMT solvers which was argued to be both easy to generate and easy to check. This affirmation was actually backed up: the competitive SMT solver veriT [5] is able to return a variant of these proof witnesses at small cost, which can then be efficiently checked in Coq [1, 2]. The propositional part of this format is based on extended resolution [28], into which theory reasoning can be plugged.

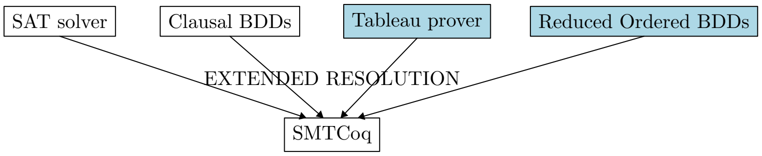
The aim of this work is to promote extended resolution as a common format for certificates about propositional logic, based on several observations:

- on a theoretical point of view, extended resolution is known to p-simulate most existing proof systems [29, 24];
- on a practical point of view, we can efficiently translate reasoning performed by some proofs systems in extended resolution (see eg. [20] for DPLL with backjumping and [26] for clausal BDDs) and efficiently check such certificates [1, 2];

- this format is easily extendable beyond propositional logic, like theory reasoning as already implemented for SMT solvers [3, 2] or quantifiers [10].

The work presented here extends this scheme with two other families of provers: the method of analytic tableaux [27, 9] and the reduced ordered binary decision diagrams [6] (BDDs in short). To do so, we detail two algorithms translating the proofs found by each of these provers into a proof in extended resolution which is polynomial in the length of the original proof.

The objective is to have a common proof format in order to share checkers and thus save a lot of human work. The work presented in this paper can be used to instrument already existing provers in order to return certificates in the concrete format of [3] – which corresponds to extended resolution – and thus directly plugged into checkers understanding it like SMTCoq [2]. As such, we could check with great confidence answers coming from these provers without having to write new code in an interactive theorem prover; this could then be extended into tactics in order to enjoy tableaux and BDDs automation inside Coq without compromising soundness.



Moreover, to add safe propositional automation into another interactive prover than Coq, one single checker would be sufficient.

Note that the goal of this paper is to give two new algorithms to generate certificates, but not to explain how to efficiently check them after: this has already been detailed in previous work [1, 2].

The paper is organized as follows. After explaining the extended resolution proofs (Section 2) and their already existing applications to SAT and SMT, we present in Section 3 the method of analytic tableaux and the algorithm to deduce a resolution certificate from a tableau proof. The same approach is applied to the BDD method in the following section (Section 4). We finally discuss related and future work in Section 5 before concluding.

2 Extended resolution

Extended resolution [28] is an extension of the well known resolution proof system [25] with the possibility to add new variables representing larger terms, giving more compact proofs than standard resolution. We first recall its definition and present our notations, before giving examples of applications.

2.1 Definitions

We are given a countable set of propositional variables $\mathcal{V}$. A *literal* l is a variable v (in which case it is called a *positive* literal) or its negation $\bar{v}$ (in which case it is called a *negative* literal). A *clause* C is a disjunction of literals, written $l_1 \vee \cdots \vee l_n$ when it is nonempty and $\Box$ otherwise. A *conjunctive normal form* (CNF in short) $\mathcal{S}$ is a set of clauses seen as their conjunction.

A *valuation* $\rho : \mathcal{V} \to \{\top, \bot\}$ is a total function mapping variables to one of the values *true* or *false*. Given a valuation ρ, it is possible to define the interpretation of literals, clauses and CNFs (respectively written $|l|_\rho$, $|C|_\rho$ and $|\mathcal{S}|_\rho$) in the standard way. We say that a CNF $\mathcal{S}$ is *satisfiable* if there exist a valuation ρ such that $|\mathcal{S}|_\rho = \top$; otherwise, we say that $\mathcal{S}$ is *unsatisfiable*.

The *resolution rule* is a deduction rule that builds a new clause from two existing clauses:

$$\frac{v \vee C \qquad \bar{v} \vee D}{C \vee D}$$

where v does not appear in C nor D, and no variable appears with one polarity in C and the other in D. The variable v is called the *resolution variable*. A comb tree of resolutions is a *resolution chain*. This rule is *refutationally complete*: a CNF $\mathcal{S}$ is unsatisfiable if and only if the empty clause can be derived by applications of the resolution rule starting with the clauses of $\mathcal{S}$.

Extended resolution extends the resolution rule with additional rules without premisses that introduce new clauses containing fresh variables implicitly representing terms of propositional logic.

A typical use consists in folding and unfolding logical connectives: to express that a fresh variable x represents $f_1 \star \cdots \star f_n$ where $\star$ is a connective, the rules are the tautological clauses stating that $x \Leftrightarrow f_1 \star \cdots \star f_n$. Such rules are used for instance to transform a Boolean problem into an equisatisfiable one in CNF [28, 3].

In the remaining of this paper, we are going to use these rules that fold and unfold connectives for all the connectives. In this section we give the examples of the $\wedge$, $\Rightarrow$ and ite (if ... then ... else ...) connectives; the same method applies for all the others (one may refer to [28, 3] for more details).

Example 2.1. A fresh variable x can represent the conjunction $x_1 \wedge x_2$ by introducing the following three rules:

$$\frac{}{\bar{x} \vee x_1} \qquad\qquad \frac{}{\bar{x} \vee x_2} \qquad\qquad \frac{}{x \vee \bar{x}_1 \vee \bar{x}_2}$$

respectively stating that x implies x_1, x implies x_2, and x_1 and x_2 together imply x.

Similarly, a fresh variable y can represent the implication $y_1 \Rightarrow y_2$ by introducing the following three rules:

$$\frac{}{\bar{y} \vee \bar{y}_1 \vee y_2} \qquad\qquad \frac{}{y \vee y_1} \qquad\qquad \frac{}{y \vee \bar{y}_2}$$

and a fresh variable z can represent the branching $\mathrm{ite}(z_1, z_2, z_3)$ (stating "if z_1 then z_2 else z_3") by introducing the following four rules:

$$\frac{}{\bar{z} \vee z_1 \vee z_3} \qquad\qquad \frac{}{\bar{z} \vee \bar{z}_1 \vee z_2} \qquad\qquad \frac{}{z \vee z_1 \vee \bar{z}_3} \qquad\qquad \frac{}{z \vee \bar{z}_1 \vee \bar{z}_2}$$

2.2 Applications

Introduced to establish lower bounds on the minimal length of proofs, extended resolution was shown to bypass resolution since it has the same power as the Extended Frege Systems [8, 29], the most powerful known proof systems. It is also known to provide short proofs to problems hard for resolution like Haken's pigeon-hole formulae [7].

The clauses learned during conflict analysis performed by modern SAT solvers can be easily derived by a resolution tree [20], and thus state-of-the-art SAT solvers like zChaff [14] or MiniSat [12] are instrumented to return resolution proofs for unsatisfiable problems. The extension allows to define certificates for more complex proof systems like clausal BDDs [26] or the Boolean part of SMT solvers [3] (which do not require their inputs to be in CNF). Even if such a proof witness can be rather huge, it takes a negligible cost to output it compared to finding that a formula is unsatisfiable. On the other side, it can be efficiently checked, for instance inside proof assistants [1, 2].

Extended resolution is thus a good candidate as a common proof format for propositional reasoning. It has already been widely studied and implemented for SAT solvers. In the remaining of this paper, we focus on two other popular propositional proof methods: the method of analytic tableaux and full BDDs and show that we can as well translate their reasoning into certificates based on extended resolution.

3 Certificates for the method of analytic tableaux

The method of analytic tableaux [27, 9] is a decision procedure for various kinds of logic. Applied to propositional logic, it can establish the unsatisfiability of any quantifier-free formula without requiring it to be in some normal form, contrary to SAT solvers. The popularity of this method comes from the fact that it can be extended to a large spectrum of standard features like quantifiers or modal logic. Its efficiency and simplicity make it largely used in applications requiring great confidence: it is for instance at the heart of the widely used `blast` tactic of the Isabelle proof assistant [21].

After presenting the method for propositional logic and theoretical results concerning its power, we explain how to deduce certificates based on extended resolution from tableaux proofs.

3.1 The method

A *refutation tableau* is a tree whose nodes are labeled with propositional formulas such that:

- a decomposition rule is applied at each node; and
- every branch from the root to a leaf contains at least a formula and its negation – in this case, we say that a branch is *closed*.

It is established that **a formula F is unsatisfiable if and only if there exists a refutation tableau of root F**. Tableaux thus give a complete method to establish the unsatisfiability of propositional formulas.

A *decomposition rule* splits a formula labeling a node above in the tree (not necessarily the current node) into one or more sub-formulas, depending on the head symbol. It can be generically described by the node:

$$\begin{tikzcd} & t_1 \star \cdots \star t_n \arrow[dl] \arrow[dr] & \\ f_1 & \cdots\cdots & f_p \end{tikzcd}$$

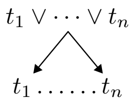
where $\star$ can be any connective, $p \leqslant n$, and any f_j can be either t_i or $\bar{t}_i$ for some i, depending on $\star$.

We give examples of these decomposition rules for the $\wedge$, $\vee$ and $\Rightarrow$ connectives.

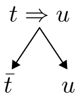
Example 3.1. A conjunction can be projected into any of its direct sub-terms:

$$\begin{tikzcd} t_1 \wedge \cdots \wedge t_n \arrow[d] \\ t_i \end{tikzcd}$$

An n-ary disjunction decomposes into n branches:

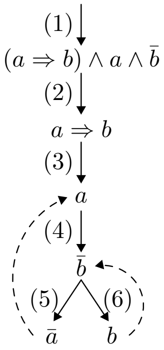


Similarly, an implication decomposes into 2 branches:



The following example proves the unsatisfiability of $(a \Rightarrow b) \wedge a \wedge \bar{b}$.

Example 3.2. A refutation tableau proving the unsatisfiability of $(a \Rightarrow b) \wedge a \wedge \bar{b}$ is:



The first extra-edge (1) simply states the initial formula as the root. Edges (2) to (4) decompose it as a conjunction. Edges (5) and (6) decomposes the implication $a \Rightarrow b$. Finally, the backwards dashed arrows illustrate the closure of each branch.

We theoretically know that resolution p-simulates analytic tableaux on CNF formulas (**Theorem 5.1** of [29]). The converse is not true: on some classes of problems, resolution can build exponentially smaller proofs than tableaux. To our knowledge, there is no link between extended resolution and full analytic tableaux.

3.2 From refutation tableaux to extended resolution

In this section, we describe a generic algorithm to transform any refutation tableau proving the unsatisfiability of f into a proof of the empty clause in extended resolution starting from f, **without requiring f to be in normal form** (contrary to [29]). It produces a proof tree whose number of nodes is **linear** in the number of nodes in the original tableau proof.

3.2.1 The algorithm on an example

To understand the idea of the algorithm, we first conduct it step by step on **Example 3.2**.

First, we assign fresh variables to each (non-strict) sub-formula of the initial formula which is not a literal. In our example, we thus add two fresh variables: $f \triangleq a \Rightarrow b$ and $g \triangleq f \wedge a \wedge \bar{b}$.

Second, we build a piece of a proof tree for each edge in the tableau in the following way:

(1) We initiate the process by stating that the fresh variable assigned to the initial formula holds: g.

(2) This step is the first projection of g, which can be derived from (1) in extended resolution:

$$\dfrac{\dfrac{}{\bar{g} \lor f} \qquad \dfrac{(1)}{g}}{f}$$

(3) - (4) Similarly, these steps are the second and third projections of g:

$$\dfrac{\dfrac{}{\bar{g} \lor a} \qquad \dfrac{(1)}{g}}{a} \qquad\qquad \dfrac{\dfrac{}{\bar{g} \lor \bar{b}} \qquad \dfrac{(1)}{g}}{\bar{b}}$$

(5) This step corresponds to decomposing f which has been obtained at step (2). This is the resolution of what has been obtained at (2) with the rule of extended resolution to decompose an implication:

$$\dfrac{\dfrac{}{\bar{f} \lor \bar{a} \lor b} \qquad \dfrac{(2)}{f}}{\bar{a} \lor b}$$

(6) This step is obtained when closing the left branch of the tree: it is a resolution between the piece of tree labeling the edge above a (3) and the the piece of tree labeling the edge above $\bar{a}$ (5):

$$\dfrac{\dfrac{(3)}{a} \qquad \dfrac{(5)}{\bar{a} \lor b}}{b}$$

Finally, we consider the closure of the last branch, as a resolution between the piece of tree labeling the edge above $\bar{b}$ (4) and the the piece of tree labeling the edge above b (6):

$$\dfrac{\dfrac{(4)}{\bar{b}} \qquad \dfrac{(6)}{b}}{\Box}$$

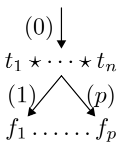
Putting everything together, we obtain the following proof of the empty clause from g:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bar{g} \lor f} \quad g}{f} \qquad \dfrac{}{\bar{f} \lor \bar{a} \lor b}}{\bar{a} \lor b} \qquad \dfrac{g \qquad \dfrac{}{\bar{g} \lor a}}{a}}{b} \qquad \dfrac{g \qquad \dfrac{}{\bar{g} \lor \bar{b}}}{\bar{b}}}{\Box}$$

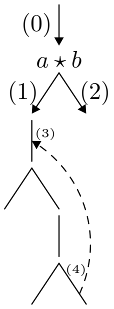
3.2.2 Formal description

Algorithm As we explained, the first step is to assign fresh variables to each (non-strict) sub-formula of the initial formula which is not a literal, and to define the very simple piece of certificate – which is a clause containing only one literal – stating that the initial formula holds.

The second step consists in successively labeling the edges with pieces of certificates, from top to bottom and from left to right. We consider the generic rule decomposing a connective $\star$:



where (0) has already been computed. We explain how to compute (1) to (p). The label of the left edge, (1), is a resolution between (0) and the rule of extended resolution which corresponds to the decomposition of $\star$. Then, for $i \in [\![2; p]\!]$, (i) is the resolution between the two pieces of certificates leading to the two formulas that finally close the branch directly on the left. For instance, on the following tableau shape, (2) is the resolution between (3) and (4).



Finally, we obtain the empty clause by a resolution between the two pieces of certificates leading to the two formulas that close the last branch.

Remarks The correctness of this algorithm mainly relies on the following invariant: a piece of certificate labeling an edge above the formula f proves a clause containing f. This ensures that the resolutions performed are always possible.

For each edge in the tableau proof, the corresponding piece of certificate contains at most two nodes: a resolution and possibly a rule of extended resolution. The final step adds a resolution to the final proof. It entails that the number of nodes in the obtained proof is linear in the number of nodes in the tableau proof.

4 Certificates for reduced ordered binary decision diagrams

A reduced ordered binary decision diagram is a normalized decision tree of a propositional formula. BDDs enjoy the property to be canonical [18]: two equivalent formulas have the same BDD – up to the order of the variables, as we will see below. As a result, it provides a decision procedure for the unsatisfiability of propositional formulas [6], which consists in progressively building the BDD of the formula, and check that the result is the false BDD.

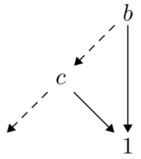
This method is rather popular since it is very efficient for certain classes of SAT problems, and well suited for circuit generation and simplification [11, 13]. Its efficiency mostly relies on the choice of a good order for the variables, which is an active research area [19].

After presenting the method for propositional logic and theoretical results concerning its power, we explain how to deduce certificates based on extended resolution from BDD proofs.

4.1 The method

A BDD is a directed acyclic graph whose nodes have either zero or two children and are labeled with propositional variables with respect to a given order. The idea is that, for every variable whose value has an influence on the formula, we construct the two sub-BDDs obtained by successively putting this variable to $\bot$ and $\top$: this is called the *Shannon expansion* of the variable.

Example 4.1. The BDD corresponding to $(a \Rightarrow (b \vee c)) \wedge (a \vee b \vee c)$ with the ordering $a > b > c$ is:



It is established that two equisatisfiable formulas have the same BDD up to the order of the variables. It entails that **every unsatisfiable formula has the 0 BDD**. BDDs thus give a complete method to establish the unsatisfiability of a formula F: it is sufficient to build the BDD of F and check that it is 0.

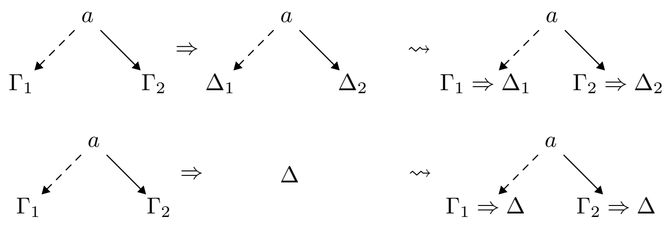
The difficulty is that the naive algorithm to compute the BDD of a formula is equivalent to computing a truth table, and thus impossible to run in practice. The idea to cope with this issue is to **build the BDD little by little and simplify it at the same time**.

To build the BDD corresponding to a formula F, we thus start with the BDDs corresponding to the variables appearing in the formulas, and we alternate between two phases:

1. building the BDD associated to a sub-formula of F of which every sub-formula has already been treated;
2. simplifying the obtained BDD.

The first step consists in recursively running through the BDDs concerned by the connective, until we reach the leaves. The following example presents the rules corresponding to the implication.

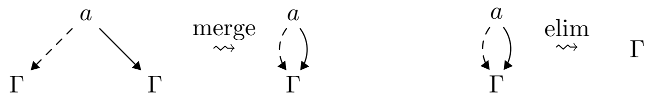
Example 4.2. Given two BDDs, their implication can be constructed using the following rules:



$$0 \Rightarrow \Delta \rightsquigarrow 1 \qquad 1 \Rightarrow \Delta \rightsquigarrow \Delta \qquad \Gamma \Rightarrow 0 \rightsquigarrow \neg\Gamma \qquad \Gamma \Rightarrow 1 \rightsquigarrow 1$$

The second rule applies when the top variable of Δ is smaller than a. The symmetric rule when this variable is greater than a is similar.

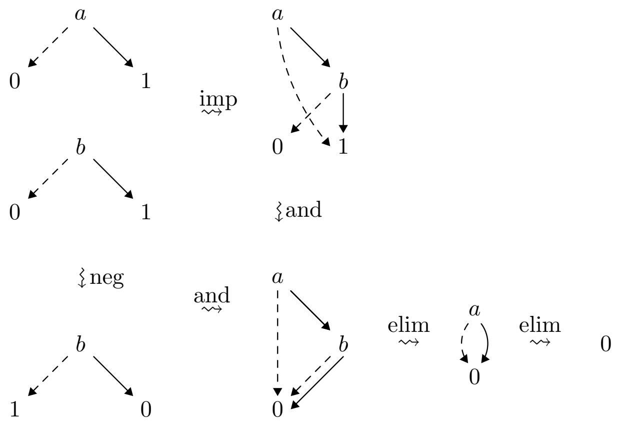
The second step is an application of the following two rules, respectively called *merge* and *elim*, until the BDD is normalized:



This step is the key point to avoid building exponential BDDs.

We give as an example the proof of unsatisfiability of $(a \Rightarrow b) \wedge a \wedge \bar{b}$.

Example 4.3. The following steps build the proof of unsatisfiability of $(a \Rightarrow b) \wedge a \wedge \bar{b}$:



We already theoretically know that extended resolution p-simulates BDDs (**Corollary 1** of [24]).

4.2 From BDDs to extended resolution

In this section, we describe a generic algorithm to transform any BDD proof of unsatisfiability into a proof of the empty clause in extended resolution, **without requiring the initial formula to be in normal form**. It produces a proof tree whose number of nodes is **polynomial** in the number of nodes in the length of the whole BDD proof.

Peltier [24] proved that extended resolution p-simulates BDDs, and this proof of course contains an algorithm to transform a BDD proof into a proof in extended resolution. We propose here a variant which is more implementation-oriented. Some ideas remain the same, but contrary to [24], we build the clauses corresponding to extended rules on demand (and not at the beginning), which changes the way we handle connectives. The construction remains polynomial.

The idea is the following:

- we define how a set of clauses can represent a BDD: it mainly consists in labeling all the nodes with names, and expressing the Shannon expansion in terms of clauses *a la* Tseitin;
- starting from the variables, we progressively build both the BDD and the resolution proof: for each step of the building of a BDD, we explain how to transform a set of clauses representing the initial BDD into a set of clauses representing the final BDD, by applying the rules of extended resolution;
- in the end, since we obtain the 0 BDD, we have built a close proof in extended resolution of the negation of the original formula. It only remains to resolve with the original formula.

This algorithm thus builds two kinds of objects:

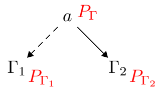
- the sets of clauses representing the intermediate BDDs: it is not mandatory to actually construct them, but the fact that they represent the intermediate BDDs is the invariant making this algorithm correct;
- the resolution proof produced little by little to switch between these sets of clauses: this is the final output of the algorithm.

This time we do not first present the algorithm on an example, since the proof which is obtain (even for a simple formula like $a \wedge \bar{a}$) is rather huge and unreadable; it is in fact easier to understand each step in the general case.

4.2.1 Algorithm

A set of clauses representing a BDD relates connected nodes by clauses.

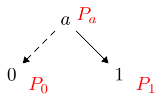
First, all the nodes have been given fresh names. Then, for each internal node, we add to the set four clauses corresponding to the Shannon expansion that this node represent:



adds the four clauses $\bar{P}_\Gamma \vee a \vee P_{\Gamma_1}; P_\Gamma \vee a \vee \bar{P}_{\Gamma_1}; \bar{P}_\Gamma \vee \bar{a} \vee P_{\Gamma_2}; P_\Gamma \vee \bar{a} \vee \bar{P}_{\Gamma_2}$ to the set.

Finally, we add one clause a leaf depending on its value: 0_{P_0} adds the clause $\bar{P}_0$ and 1_{P_1} adds the clause P_1.

Example 4.4.
- The set of clauses associated to the BDD 0_{P_0} is $\{\bar{P}_0\}$.
- The BDD of a variable a is:

  The corresponding set of clauses is $\{\bar{P}_a \vee a \vee P_0; P_a \vee a \vee \bar{P}_0; \bar{P}_a \vee \bar{a} \vee P_1; P_a \vee \bar{a} \vee \bar{P}_1; \bar{P}_0; P_1\}$.
- We come back to **Example 4.1** and give the names P_b to the node labeled with b, P_c to the node labeled with c, P_0 to the node labeled with 0 and P_1 to the node labeled with 1. The set of clauses associated to this BDD is $\{\bar{P}_b \vee b \vee P_c; P_b \vee b \vee \bar{P}_c; \bar{P}_b \vee \bar{b} \vee P_1; P_b \vee \bar{b} \vee \bar{P}_1; \bar{P}_c \vee c \vee P_0; P_c \vee c \vee \bar{P}_0; \bar{P}_c \vee \bar{c} \vee P_1; P_c \vee \bar{c} \vee \bar{P}_1; \bar{P}_0; P_1\}$.

All these clauses correspond to the rules for the ite, $\top$ or $\bot$ connectives in extended resolution, and are thus provable in extended resolution.

As we said, the algorithm consists in starting with the BDDs of the variables and the corresponding sets of clauses, and then successively apply the connectives, merge and elim rules both on the BDDs and on the set of clauses. In the end, we will obtain the BDD 0_{P_Γ}, and thus a proof in extended resolution of $\bar{P}_\Gamma$, where P_Γ is the fresh variable associated to the initial formula. We will finally conclude by resolving with it.

It thus remains to explain how we transform sets of clauses representing BDDs into a set of clauses representing the BDD obtained after an application of a connective or the merge and elim rules. In this paper, we focus on the connectives, since the simplification rules are handled like in [24] (and it is roughly the same ideas as for connectives).

To handle a connective $f \star g$, we prove by induction on the sum of the number of nodes in Γ and Δ that we can at the same time:

(a) transform the sets of clauses associated to Γ and Δ into a set of clauses representing $\Gamma \star \Delta$;

(b) generate the extended rules corresponding to $\Gamma \star \Delta$;

where Γ is the BDD corresponding to f and Δ is the BDD corresponding to g. Only the first item is needed for the final algorithm to work, but the second item is required in the proof by induction. This is how we avoid to first transform the formula into a set of clauses, like in [24].

We show the proof in the case of the implication: this both variant and covariant connective illustrates well the process.

Base cases We consider only the base case $0_{P_\Gamma} \Rightarrow \Delta_{P_\Delta} \rightsquigarrow 1_{P_{\Gamma\Rightarrow\Delta}}$ since the others are similar.

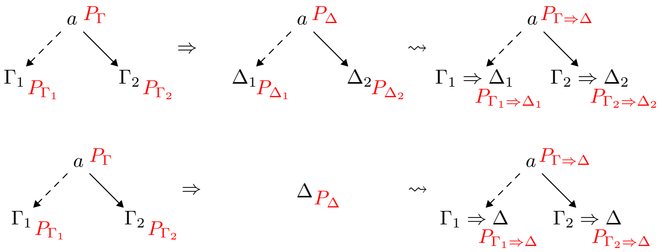
We first define the name $P_{\Gamma\Rightarrow\Delta}$ by extended resolution:

$$\frac{}{\bar{P}_{\Gamma\Rightarrow\Delta} \vee \bar{P}_\Gamma \vee P_\Delta}\,(1) \qquad \frac{}{P_{\Gamma\Rightarrow\Delta} \vee P_\Gamma}\,(2) \qquad \frac{}{P_{\Gamma\Rightarrow\Delta} \vee \bar{P}_\Delta}\,(3)$$

This already fulfills step (b).

The set of clauses representing 0 is $\{\bar{P}_\Gamma\}$, obtained by extended resolution on the connective $\bot$. By resolving it with (2), we obtain a proof of $P_{\Gamma\Rightarrow\Delta}$, which fulfills step (a).

Inductive cases The inductive cases correspond to the following labeling:


a P_Γ
Γ_1 P_{Γ_1}
Γ_2 P_{Γ_2}
$\Rightarrow$
a P_Δ
Δ_1 P_{Δ_1}
Δ_2 P_{Δ_2}
$\rightsquigarrow$
a $P_{\Gamma\Rightarrow\Delta}$
$\Gamma_1 \Rightarrow \Delta_1$ $P_{\Gamma_1\Rightarrow\Delta_1}$
$\Gamma_2 \Rightarrow \Delta_2$ $P_{\Gamma_2\Rightarrow\Delta_2}$
a P_Γ
Γ_1 P_{Γ_1}
Γ_2 P_{Γ_2}
$\Rightarrow$
Δ P_Δ
$\rightsquigarrow$
a $P_{\Gamma\Rightarrow\Delta}$
$\Gamma_1 \Rightarrow \Delta$ $P_{\Gamma_1\Rightarrow\Delta}$
$\Gamma_2 \Rightarrow \Delta$ $P_{\Gamma_2\Rightarrow\Delta}$

We are going to concentrate only on the first one; the second one is similar (with even fewer resolutions).

The induction hypothesis for (b) gives us:

$$\left\{\begin{array}{ll} \bar{P}_{\Gamma_1\Rightarrow\Delta_1} \vee \bar{P}_{\Gamma_1} \vee P_{\Delta_1} & (1) \\ P_{\Gamma_1\Rightarrow\Delta_1} \vee P_{\Gamma_1} & (2) \\ P_{\Gamma_1\Rightarrow\Delta_1} \vee \bar{P}_{\Delta_1} & (3) \end{array}\right. \quad \text{and} \quad \left\{\begin{array}{ll} \bar{P}_{\Gamma_2\Rightarrow\Delta_2} \vee \bar{P}_{\Gamma_2} \vee P_{\Delta_2} & (4) \\ P_{\Gamma_2\Rightarrow\Delta_2} \vee P_{\Gamma_2} & (5) \\ P_{\Gamma_2\Rightarrow\Delta_2} \vee \bar{P}_{\Delta_2} & (6) \end{array}\right.$$

The induction hypothesis for (a) is:

$$\left\{\begin{array}{ll} \bar{P}_\Gamma \vee a \vee P_{\Gamma_1} & (7) \\ P_\Gamma \vee a \vee \bar{P}_{\Gamma_1} & (8) \\ \bar{P}_\Gamma \vee \bar{a} \vee P_{\Gamma_2} & (9) \\ P_\Gamma \vee \bar{a} \vee \bar{P}_{\Gamma_2} & (10) \end{array}\right. \quad \text{and} \quad \left\{\begin{array}{ll} \bar{P}_\Delta \vee a \vee P_{\Delta_1} & (11) \\ P_\Delta \vee a \vee \bar{P}_{\Delta_1} & (12) \\ \bar{P}_\Delta \vee \bar{a} \vee P_{\Delta_2} & (13) \\ P_\Delta \vee \bar{a} \vee \bar{P}_{\Delta_2} & (14) \end{array}\right.$$

We first define (b) by extension:

$$\left\{\begin{array}{ll} \bar{P}_{\Gamma\Rightarrow\Delta} \vee \bar{P}_\Gamma \vee P_\Delta & (15) \\ P_{\Gamma\Rightarrow\Delta} \vee P_\Gamma & (16) \\ P_{\Gamma\Rightarrow\Delta} \vee \bar{P}_\Delta & (17) \end{array}\right.$$

and then (a) by resolution:

$$\begin{cases} \bar{P}_{\Gamma\Rightarrow\Delta} \vee a \vee P_{\Gamma_1\Rightarrow\Delta_1} & \text{(resolution of } 15, 8, 11, 2, 3) \\ P_{\Gamma\Rightarrow\Delta} \vee a \vee \bar{P}_{\Gamma_1\Rightarrow\Delta_1} & \text{(resolution of } 17, 12, 1, 7, 16) \\ \bar{P}_{\Gamma\Rightarrow\Delta} \vee \bar{a} \vee P_{\Gamma_2\Rightarrow\Delta_2} & \text{(resolution of } 15, 10, 13, 5, 6) \\ P_{\Gamma\Rightarrow\Delta} \vee \bar{a} \vee \bar{P}_{\Gamma_2\Rightarrow\Delta_2} & \text{(resolution of } 17, 14, 4, 9, 16) \end{cases}$$

4.2.2 Remarks

As proved in [24], the whole algorithm – together with the transformation of the merge and elim rules – builds a proof whose number of nodes is polynomial in the length of the original BDD proof.

As we said, the correctness of this algorithm relies on the fact that the intermediate sets of clauses always represent the intermediate BDDs, in order to obtain in the end a proof of the negation of the initial formula.

We think that the new presentation for connectives makes this algorithm easier to implement: the functions combining BDDs for each connective just have to return the clauses generated by the (b) step in addition to the (a) step, instead of somehow looking into a large set initially computed.

5 Discussion

5.1 Related works

Lots of related works were already presented throughout the paper.

To our knowledge, this is the first transformation of full propositional tableaux (and not only clausal tableaux) into extended resolution, and this is the first work aiming at providing certificates that can be checked by an external tool (instead of a theoretical comparison of two proof systems). The Isabelle tableau prover [21] gives witnesses, but encoded directly into Isabelle proofs, and thus not applicable to systems not based on Higher-Order Logic.

The BDD algorithm highly relies on [24], but in an implementation perspective (as we argued in Section 4.2.2). [26] presented an implementation of a translator from clausal BDDs into extended resolution, but this is limited to CNF formulas, and requires to treat lots of particular cases whereas our algorithm is more generic.

Even if this work relies on different previous works for the different parts, this is a first attempt to unify certificates for three major paradigms for propositional proving: DPLL with backjumping, the method of tableaux, and BDDs.

Other proof formats for certificates in propositional logic have been proposed. The format based on extended resolution used in [26] called TraceCheck (which is in particular returned by the SAT solver BooleForce) is very close to ours, and thus could be directly used by our algorithms. The recent Reverse Unit Propagation format [15, 16] (RUP in short) gives shorter proofs than resolution, but is currently restricted to inputs in CNF– whereas tableaux and BDD provers deal with the full propositional logic without requiring preprocessing.

5.2 Future works

Obviously, the next step is to instrument existing provers in order to return these certificates, and to evaluate the efficiency (in particular, it must not be costly to output and check the certificate compared to finding the proof). This could then be plugged into a certified checker like SMTCoq [2], in order to check *a posteriori* the answers given by the automatic provers.

The tableaux and BDD algorithms we presented here are rather naive, and we need to understand how the variants that are actually implemented by the provers enter into this schema. Some improvements do not affect our algorithms, like the choice of the order of the variables in BDDs, but others may require changes.

Our algorithms could also be extended with other features, in particular quantifiers: we know how to extend our certificates with quantifiers [10], and they are well handled by tableaux proofs. We would also like to deal with other logics than classical logic, like intuitionistic or modal logic, for which tableaux are quite frequently used.

A broad spectrum study should also extend this work to other proof formats, like an extension of the RUP proofs [15] to full propositional logic, as well as other proof search paradigms, like stochastic search algorithms.

Acknowledgments The author thanks Filip Marić who asked a question that motivated this work, and the anonymous reviewers for their insightful comments.

6 Conclusion

In this paper, we presented two new algorithms to transform into certificates in extended resolution the proofs computed by two major propositional provers: tableaux provers and BDDs. Since this translation is already efficiently implemented for SAT solvers based on DPLL with backjumping, this opens the way towards a common format for certificates for propositional solvers for which we already have efficient certified checkers.

References

[1] M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. In Kaufmann and Paulson [17], pages 83–98.

[2] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *CPP*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[3] F. Besson, P. Fontaine, and L. Théry. A Flexible Proof Format for SMT: a Proposal. In *PxTP 2011: First International Workshop on Proof eXchange for Theorem Proving August 1, 2011 Affiliated with CADE 2011, 31 July-5 August 2011 Wrocław, Poland*, pages 15–26, 2011.

[4] S. Böhme and T. Weber. Fast LCF-Style Proof Reconstruction for Z3. In Kaufmann and Paulson [17], pages 179–194.

[5] T. Bouton, D.C.B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In R. A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[6] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[7] Stephen A Cook. A Short Proof of the Pigeon Hole Principle using Extended Resolution. *ACM SIGACT News*, 8(4):28–32, 1976.

[8] Stephen A. Cook and Robert A. Reckhow. On the Lengths of Proofs in the Propositional Calculus (Preliminary Version). In Robert L. Constable, Robert W. Ritchie, Jack W. Carlyle, and Michael A. Harrison, editors, *STOC*, pages 135–148. ACM, 1974.

[9] Marcello D'Agostino, Dov M Gabbay, Reiner Hähnle, and Joachim Posegga. *Handbook of Tableau Methods*. Springer, 1999.

[10] D. Deharbe, P. Fontaine, and B. W. Paleo. Quantier Inference Rules for SMT proofs. In *PxTP 2011: First International Workshop on Proof eXchange for Theorem Proving August 1, 2011 Affiliated with CADE 2011, 31 July-5 August 2011 Wrocław, Poland*, pages 33–39, 2011.

[11] Rolf Drechsler, Junhao Shi, and Görschwin Fey. Synthesis of Fully Testable Circuits From BDDs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(3):440–443, 2004.

[12] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[13] Görschwin Fey, Junhao Shi, and Rolf Drechsler. BDD Circuit Optimization for Path Delay Fault Testability. In *DSD*, pages 168–172. IEEE Computer Society, 2004.

[14] Z. Fu, Y. Marhajan, and S. Malik. zChaff. *Research Web Page. Princeton University, USA,(March 2007) http: // www. princeton. edu/ ~ chaff/ zchaff. html*, 2007.

[15] Allen Van Gelder. Verifying RUP Proofs of Propositional Unsatisfiability. In *ISAIM*, 2008.

[16] Allen Van Gelder. Producing and verifying extremely large propositional refutations - Have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.

[17] M. Kaufmann and L. C. Paulson, editors. *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*. Springer, 2010.

[18] D. Kroening and O. Strichman. *Decision Procedures: an Algorithmic Point of View.* Springer-Verlag New York Inc, http://www.decision-procedures.org, 2008.

[19] A. Layeb, N. Tabib, and B. Brirem. Genetic Algorithm and Variable Neighbourhood Search for BDD Variable Ordering Problem. *INFOCOMP Journal of Computer Science*, 10(1):29–35, 2011.

[20] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(). *J. ACM*, 53(6):937–977, 2006.

[21] Lawrence C. Paulson. A Generic Tableau Prover and its Integration with Isabelle. *J. UCS*, 5(3):73–87, 1999.

[22] Lawrence C Paulson and Jasmin Christian Blanchette. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In G. Sutcliffe, E. Ternovska, and S. Schulz, editors, *International Workshop on the Implementation of Logics (IWIL-2010)*, 2010.

[23] L.C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[24] Nicolas Peltier. Extended Resolution Simulates Binary Decision Diagrams. *Discrete Applied Mathematics*, 156(6):825–837, 2008.

[25] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[26] Carsten Sinz and Armin Biere. Extended Resolution Proofs for Conjoining BDDs. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006.

[27] R. M. Smullyan. *First-Order Logic.* Springer-Verlag, 1968.

[28] G. Tseitin. On the Complexity of Proofs in Propositional Logics. In *Seminars in Mathematics*, volume 8, pages 466–483, 1970.

[29] Alasdair Urquhart. The Complexity of Propositional Proofs. *Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

Challenges in Using OpenTheory to Transport Harrison's HOL Model from HOL Light to HOL4

Ramana Kumar

Computer Laboratory, University of Cambridge, UK
Ramana.Kumar@cl.cam.ac.uk

Abstract

OpenTheory is being used for the first time (in work to be described at ITP 2013) as a tool in a larger project, as opposed to in an example demonstrating OpenTheory's capability. The tool works, demonstrating its viability. But it does not work completely smoothly, because the use case is somewhat at odds with OpenTheory's primary design goals. In this extended abstract, we explore the tensions between the goals that OpenTheory-like systems might have, and question the relative importance of various kinds of use. My hope is that describing issues arising from work in progress will stimulate fruitful discussion relevant to the development of proof exchange systems.

1 The OpenTheory Vision: Reusable Theory Packages

OpenTheory [3] is a format for representing theories in higher-order logic, inspired by an analogy: theories are software packages, which may depend on one another, and proofs are bytecode for a portable virtual machine. Just as a piece of software (especially a library) can be useful in the development of another (library or application), so can the definitions and theorems in one formal development be a useful component for another. For example, existing theories about floating-point numbers and finite words might both be used in the development a formal model of an instruction set architecture. The problem is that these theories may have been developed in different provers and may not be directly compatible.

There are multiple HOL prover implementations, analogous to different platforms, including HOL Light, HOL4, ProofPower, and Isabelle/HOL, each with a broadly similar logic and architecture, but with significant differences in their libraries of native theories and their integrated proof development (e.g. automation) tools. The aim of OpenTheory is to support theory engineering across provers, akin to the software engineering applicable to large software developments. (OpenTheory does not yet support provers using different logics like Coq or Twelf: bridging the differences between HOL-based systems is enough of a challenge already.) Large formal developments that span multiple provers require methods of exchange, and benefit when those methods promote reuse of native theories as opposed to isolated duplications.

OpenTheory's approach to exchange is based on a standard format for low-level proofs coupled with a standard library of theory packages. The standard format enables capture and replay of proof work in any prover supporting the format. The standard library factors out various core theories that are likely to be used by many developments and provides a standard interface for depending on such library theories by reusing a prover's native theories as opposed to rederiving them on import. (For example, if the standard library has some basic theorems about `foo` and HOL4 has an equivalent constant `FOO`, then we map `foo` to `FOO` on import and enable reuse of all of HOL4's `FOO` theorems, rather than importing `foo` theorems, proving $\vdash \texttt{foo} = \texttt{FOO}$, and mediating any combined reasoning through the equivalence.)

When using OpenTheory, the goal therefore is to create a package that is independent of any particular prover in the sense that its dependencies are all standard theories that are supposed to be supported by every prover. Furthermore, the package should have a clear topic and a clean interface (the theorems and definitions it exports should be meaningful, useful results, as opposed to auxiliary constants or intermediate lemmas), because these criteria promote reusability in different environments: if intermediate steps can be handled entirely within OpenTheory, they never need to be proved at the endpoints.

2 The Use Case: Proof Transport, with Modifications

John Harrison [2] created a formal model of HOL in HOL Light and proved it sound[1]. We (Myreen et al. [5]) are working on tools in HOL4 for verifying implementations of code derived from shallow embeddings. Harrison's soundness proof for HOL supports an excellent case study for our tools wherein we will verify an implementation of HOL Light, but it lives in a different prover. Our options to use his work are:

- Manually port the high-level script files from HOL Light to HOL4.
- Automatically port the low-level proofs from HOL Light to HOL4.
- (Manually port our verification tools from HOL4 to HOL Light: too much work.)
- (Automatically port the high-level script files: we are unaware of any tools for that.)

A manual port of the high-level scripts would have been feasible, although more work than using the OpenTheory link. It would also have been tedious, and would not scale to larger developments, so we went for the automatic option. OpenTheory is currently the only tool for low-level proof transport with an exporter for HOL Light and an importer for HOL4, so it was the obvious choice.

Transport Overview

We used Joe Hurd's proof-logging fork of HOL Light[2], rather than mainline HOL Light, because that is the only version with an OpenTheory exporter. The key steps to transport the contents of a single HOL Light script file are:

1. Modify the HOL Light script file so that it builds in the proof-logging fork.
2. Mark each theorem that is desired in the exported output.
3. Run the exporter to create an OpenTheory article.
4. Run the OpenTheory tool on the article to compress it and clean it of artefacts[3] produced by the HOL Light exporter.
5. Run the HOL4 importer on the article, and find which constants and theorems the article requires as assumptions. (These should only be standard library theorems, but HOL4 does not yet automatically support everything in the standard library.)

[1] Specifically, his theories define the syntactic inference rules of HOL and their intended semantics, and prove soundness (statements derived by the rules are true in the semantics) and consistency (some statement cannot be derived).

[2] `http://src.gilith.com/hol-light.html`

[3] For example, the identity function `NUMERAL` wrapped around numerals to aid parsing and printing.

6. Create a HOL4 script that proves the article's assumptions, imports the article, and saves the theorems. The result is a HOL4 theory suitable for native use.

We only transported a single script, the one defining Harrison's inference rules for HOL. We made little use of OpenTheory's facilities for composing theory packages, since our focus was getting all the results rather than packaging them nicely for others.

The Need for Modifiability

Harrison's model of HOL does not include rules for defining new constants, but the implementation we plan to verify will support definitions. Therefore, we would like to extend his model and soundness proof with support for definitions. The problem is that low-level proofs are effectively unmodifiable.

After following the transport process above, the original HOL Light script files remain the only human-modifiable sources for the theory. We must work with them to make our extensions, and then transport the results. The transport process thus becomes part of our development chain.

We contend that most uses of proof transport will share this feature of wanting access to modifiable sources. Even when no major extensions need to be made to the transported theory, it is often necessary to tweak definitions or expose internal results to make downstream development possible or easier. Harrison's proofs could be packaged, as they are, as a reusable theory, but we would still want the option for an easy re-export of the proofs after—and while—we modify and extend them to support definitions.

Specific Experiences

In this section, I describe some of the particular issues that affected our use of OpenTheory. At present, we have successfully transported Harrison's definition of the syntactic inference rules of HOL (130 theorems/definitions in total). We also updated his definition to support constant definition (and transported it again). We have not yet updated or transported the soundness proof.

The main result of transport is the definition of the provability relation, |-, and all the constants it depends on (such as ACONV). The theorem in HOL4 after transport, shown partially below, looks the same as the original[4] in HOL Light.

$$
\begin{aligned}
&\vdash (\forall t\, defs.\, \mathtt{welltyped\_in}\ t\ defs \implies ((defs,[\,])\ \mathtt{|-}\ t\ \mathtt{===}\ t)) \land\\
&\quad (\forall asl_1\, asl_2\, l\, m_1\, m_2\, r\, defs.\\
&\qquad ((defs,asl_1)\ \mathtt{|-}\ l\ \mathtt{===}\ m_1) \land ((defs,asl_2)\ \mathtt{|-}\ m_2\ \mathtt{===}\ r) \land\\
&\qquad \mathtt{ACONV}\ m_1\ m_2 \implies\\
&\qquad ((defs,\mathtt{TERM\_UNION}\ asl_1\ asl_2)\ \mathtt{|-}\ l\ \mathtt{===}\ r)) \land\\
&\quad (\forall asl_1\, l_1\, r_1\, asl_2\, l_2\, r_2\, defs.\\
&\qquad ((defs,asl_1)\ \mathtt{|-}\ l_1\ \mathtt{===}\ r_1) \land ((defs,asl_2)\ \mathtt{|-}\ l_2\ \mathtt{===}\ r_2) \land\\
&\qquad \mathtt{welltyped}\ (\mathtt{Comb}\ l_1\ l_2) \implies\\
&\qquad ((defs,\mathtt{TERM\_UNION}\ asl_1\ asl_2)\ \mathtt{|-}\ \mathtt{Comb}\ l_1\ l_2\ \mathtt{===}\ \mathtt{Comb}\ r_1\ r_2)) \land\\
&\quad \ldots
\end{aligned}
$$

[4] Not Harrison's original, but the extended version with support for definitions; the extension was made to the HOL Light script file before transport.

Some of the issues described below arise in part because the tools (the HOL Light exporter, the OpenTheory tool, and the HOL4 importer) and the OpenTheory standard library are all still young. I expect that if OpenTheory is used more, and the standard library gains traction, proof transport will enjoy much better support.

Reuse of native constants and theorems On the whole, the standard ontology provided by OpenTheory was a useful intermediary for mapping constants in HOL Light to their moral equivalents in HOL4. For our work a direct mapping of constants would also have been fine, but we would have had to write it. The advantage of using standard library constants is that both the HOL Light exporter and the HOL4 importer contained (most of) the mappings we needed before we started. We needed to manually provide only six additional mappings, mostly for constants in the standard library that don't exist (and hence also had to be defined) in HOL4, for example the standard library's `BIT0` constant for numerals (since HOL4 uses `BIT1` and `BIT2` instead).

It was also good that many of the assumptions of the transported theory were satisfied by corresponding native theorems in HOL4. We needed, however, to state and prove sixty additional theorems in order to match the article's assumptions exactly. All of these had trivial (less than two lines) proofs, and were often rephrasings of native theorems.

For example, the standard library asks for both

$$\vdash \forall p.\, \texttt{EVERY}\; p\; [\,] \text{ and } \vdash \forall p\, h\, t.\, \texttt{EVERY}\; p\; (h{::}t) = p\; h \wedge \texttt{EVERY}\; p\; t,$$

whereas HOL4 has a single theorem

$$\vdash (\forall P.\, \texttt{EVERY}\; P\; [\,] = \texttt{T}) \wedge (\forall P\, h\, t.\, \texttt{EVERY}\; P\; (h{::}t) = P\; h \wedge \texttt{EVERY}\; P\; t).$$

Slightly less trivially, the standard library asks for

$$\vdash \forall p\, g\, h.\, \exists f.\, \forall x.\, f\; x = \texttt{if}\; p\; x\; \texttt{then}\; f\; (g\; x)\; \texttt{else}\; h\; x,$$

whereas HOL4 has a constant, `WHILE`, and a theorem saying it is a suitable witness for f.

The HOL4 importer might be augmented to try strategies for automatically proving such assumptions on import. Alternatively, since all the assumptions are supposed to be in the standard library, HOL4 might want a native copy of every theorem in the standard library. It is unclear which of these approaches is better.

Representation of sets OpenTheory represents sets as an abstract type (A `set`), which leads to a clean interface and easy import, but can make exporting scripts that represent sets as predicates ($A \to$ `bool`) difficult. Harrison's work, especially the soundness proof, makes considerable use of sets as predicates[5]. We hope to port his proof, but doing so would be non-trivial because it has not been made to work with the proof-logging fork of HOL Light and makes extensive use of sets represented by predicates.

Source annotations The proof-logging fork of HOL Light was designed primarily for creating theory packages for the standard library. As such, many of HOL Light's native theories have been modified so they can be exported as the standard library theories, and the exporter is not designed to accept native HOL Light theories without modifications.

[5] His use of predicates is in addition to his definition of a new type for representing sets in the HOL semantics; being of a separate type, the latter "sets" pose no problem.

At best, these modifications include marking each theorem in the script that is desired as an output of the final theory package, which for us meant marking essentially every theorem proved in the file.

At worst, proof scripts need to be changed substantially to accommodate the standard library's representation of sets and the fact that some of HOL Light's automation has not yet been made to work with the rest of the fork. We ran into this problem when one of the definitions required some of HOL Light's advanced recursive function definition tools, which did not work in the proof-logging fork. After failing to devise an alternative proof, we called on Joe Hurd to upgrade the fork. Thankfully he was able to do so.

Article file size, and transport efficiency The transport process centres around an article file that contains the log of primitive inferences comprising the theory. The generated article for Harrison's definition of HOL's inference rules is around twelve megabytes in size. The original script file is only thirty kilobytes in size. Having such a large file in the development chain is unwieldy, especially when using version control since it increases our repository size significantly. We opt to keep the article in our repository because we don't expect everyone using the repository to have the tools necessary to generate it (e.g. an installation of proof-logging HOL Light). Processing (exporting, compressing, importing) the article also takes several minutes.

Kaliszyk and Krauss [4] will present techniques for efficient proof transport, in terms of both speed and size, at ITP 2013. We hope these will be adopted by OpenTheory, where efficiency has been an important concern but not the highest priority.

Theorem names OpenTheory avoids names wherever possible, and refers to theorems by their statements. This policy has the benefit of reducing opportunities for conflict. But nicely named theorems are useful in most provers, including HOL4. A principled approach, which is currently unimplemented, would be to pass names and other metadata (e.g. whether a theorem should be an automatic rewrite rule) on a separate channel alongside the OpenTheory proof data. We used the simpler hack of encoding the theorem name in the theorem itself (as the name of an extra variable).

3 The Impossible Wish: Script Portability

In light of the need for modifiability, the holy grail of theory exchange would be automatic methods for porting high-level script files. This corresponds to porting software between high-level languages as opposed to compiling to a common bytecode that has an interpreter on each platform. But just as software ports are usually done manually, I expect the variety and complexity of script files to make automatic ports unlikely: HOL Light/HOL4 script files may contain arbitrary OCaml/SML code.

Another approach would be to generate script files by "decompiling" low-level proofs. This idea is relatively unexplored in the context of proofs, but the analogous field of decompilation for software binaries may offer some ideas. Recovering an invocation of a high-level tactic from the trace of its primitive inferences sounds like a difficult problem, but perhaps the traces are sufficiently idiosyncratic.

The Common HOL platform, as used by HOL Zero [1], is aimed in the direction of script file portability. It defines an API of high-level functions that a HOL-based prover might use in a script file. Beyond such an API, one could use a standard, purpose-built script file language, such as the Isar formal proof language [6] used by Isabelle. Or, going further, we might standardise on a single prover and eliminate the need for exchange at all.

Of course, diversity has its advantages: there are various factors affecting the experience of using a prover including the interface, the speed, the automation libraries, the theory libraries, and the degree to which hacking the prover is encouraged. There are different trade-offs to make between these; we don't know the single best answer and there probably isn't one.

4 Summary of Tensions

In the table below, we characterise the two ways to consider a proof exchange system that we have explored in our use of OpenTheory.

For Making Reusable Packages	For Proof Transport
export a clean interface	export everything
depend on and produce standard results	depend on standard results
export once and archive	export and import with every modification
export can be expensive so import is cheap	export and import must be cheap
can expect script file preparation	should accept script files untouched

In some ways, the differences are in degree rather than in kind. For example, reusable packages may have a clean interface that happens to include everything, or may need occasional modification for maintenance. But when modifications are being made in a development cycle, where one is experimenting with the effects of changes in imported definitions on further theories in the target system, cheap proof transport becomes more important.

Should more work be done by the importer or the exporter? OpenTheory tries to find a balance in the middle, but perhaps biases against the exporter since the work to make a native theory into an OpenTheory package is slightly different for each theory, and must be done for each one, whereas the work required to import any OpenTheory package (which respects the standard library) as a native theory can be done once and for all.

Two Modes of Use

Direct proof transport is likely to be a commonly desired function of proof exchange systems like OpenTheory. Such usage benefits from the existence of a well-supported (by provers) well-designed standard library of theories. However, the concerns when creating such a library can get in the way of making the transport process smooth. Efficient transport is necessary as long as the original sources remain the only modifiable ones.

It might be possible, however, for OpenTheory to support two modes of use. One mode would be for creating theory packages fit for public consumption, and conforming to the standards required by a high-quality package repository. The other mode would be for porting a theory from one prover to another wholesale, without modifications. Implementing the second (proof transport) mode would require more ingenuity, since the source theory may depend on particular representations and theories that are outside the standard library. However, one could always fall back on exporting a larger, less reusable package that derives its results from (at worst) the axioms.

Acknowledgements I am grateful to Magnus Myreen for discussion about his experiences with OpenTheory, and for advice and comments on this paper. I appreciate feedback on this paper from Mike Gordon, and from the anonymous reviewers. I thank Joe Hurd for introducing me to OpenTheory and keeping me involved in its development. My work is supported financially by the Gates Cambridge Trust.

References

[1] Mark Adams. Introducing HOL Zero - (extended abstract). In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *ICMS*, volume 6327 of *Lecture Notes in Computer Science*, pages 142–143. Springer, 2010.

[2] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.

[3] Joe Hurd. The OpenTheory standard theory library. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Third International Symposium on NASA Formal Methods (NFM 2011)*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, April 2011.

[4] Cezary Kaliszyk and Alexander Krauss. Scalable LCF-style proof translation. To appear in ITP 2013.

[5] Magnus Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of HOL Light. To appear in ITP 2013.

[6] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *TPHOLs*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999.

Robust, Semi-Intelligible Isabelle Proofs from ATP Proofs

Steffen Juilf Smolka and Jasmin Christian Blanchette

Technische Universität München, Germany

Abstract

Sledgehammer integrates external automatic theorem provers (ATPs) in the Isabelle/HOL proof assistant. To guard against bugs, ATP proofs must be reconstructed in Isabelle. Reconstructing complex proofs involves translating them to detailed Isabelle proof texts, using suitable proof methods to justify the inferences. This has been attempted before with little success, but we have addressed the main issues: Sledgehammer now transforms the proofs by contradiction into direct proofs (as described in a companion paper [4]); it reconstructs skolemization inferences; it provides the right amount of type annotations to ensure formulas are parsed correctly without overwhelming them with types; and it iteratively tests and compresses the output, resulting in simpler and faster proofs.

1 Introduction

Sledgehammer [22] is a proof tool that connects the Isabelle/HOL proof assistant [19] with external automatic theorem provers (ATPs), including first-order resolution provers and SMT solvers. Given an interactive proof goal, it heuristically selects hundreds of facts (lemmas, definitions, and axioms) from Isabelle's vast libraries, translates them to first-order logic (FOL), and invokes the external provers. Although Sledgehammer can be trusted as an oracle,[1] most users are satisfied only once the proof has been reduced to Isabelle primitives.

When Sledgehammer was originally conceived, the plan was to have it deliver detailed proofs in Isabelle's Isar language [31], a textual, human-readable format inspired by Mizar [17]. Paulson and Susanto [23] designed a prototype that performs inference-by-inference translation of ATP proofs into Isar proofs and justifies each Isar inference using *metis*, a proof method based on Hurd's Metis resolution prover [14]. This idea was abandoned for several reasons: The resulting proofs by contradiction were unpalatable, so that users were disinclined to insert them in their theory text; they were often syntactically incorrect due to technical issues; and a single *metis* call with the short list of needed lemmas usually sufficed to re-find the proof.

Proof reconstruction with *metis* one-liners means that the proof must be re-found each time the Isabelle theory text is processed. This sometimes fails for difficult proofs that *metis* cannot re-find within a reasonable time and is vulnerable to small changes in the formalization. It also provides no answer to users who would like to understand the proof—whether it be novices who expect to learn from it, experts who must satisfy their curiosity, or merely skeptics. But perhaps more importantly, *metis* supports no theories beyond equality, which is becoming a bottleneck as automatic provers are being extended with dedicated procedures for theory

[1] For many years, Sledgehammer employed type-unsound encodings by default [18], making it unsuitable as an oracle. Newer versions use optimized type-sound encodings [5].

reasoning. The Z3-based *smt* proof method [7] is a powerful alternative to *metis*, but it depends on the availability of Z3 on the user's machine for proof replay, which hinders its acceptance among users. Moreover, due to its incomplete quantifier handling, it can fail to re-find a proof generated by a resolution prover.

The remedy to all these issues is well known: to generate detailed, structured Isar proofs based on the machine-generated proofs, as originally envisioned by Paulson and Susanto. The first issue is that the Isar proof, like the underlying ATP proof, is by contradiction. A companion paper describes an algorithm that turns such proofs around [4]. The present paper describes further enhancements that increase the intelligibility and robustness of the output and that are implemented in Sledgehammer's proof translation pipeline (Section 3).

- *Skolemization*: Sledgehammer communicates with ATPs in full FOL as opposed to quantifier-free clause normal form (CNF). Skolemization is performed by the external ATPs, but it must be reconstructed in Isar (Section 4).

- *Type annotations*: Isabelle can generate strings from formulas, but it does not always understand its own output. Terms are often read back with overly general polymorphic types, resulting in failures. Annotating each subterm with type constraints impedes readability. Instead, Sledgehammer now employs an algorithm that introduces a minimal, complete set of type annotations (Section 5).

- *Proof preplay*: Sledgehammer users waste precious time on proofs that fail or take too long. Proof preplay addresses this by testing the generated proofs for a few seconds before presenting them to users. If several proofs are available, users can choose the fastest one and insert it in their theory text (Section 6).

- *Proof compression*: The generated proofs can be arbitrarily detailed depending on which ATP is used. Users normally want to compress straightforward chains of deduction into single Isar inferences, justified by a single *metis* call. This is performed in combination with preplaying to obtain faster and simpler proofs (Section 7).

Although the focus is on Isabelle, most of these techniques are equally applicable to proof construction for other Sledgehammer-like tools, such as HOL(y)Hammer for HOL Light [15] and Miz$\mathbb{AR}$ for Mizar [1].

2 Isabelle/HOL

The Isabelle/HOL proof assistant is based on polymorphic higher-order logic (HOL) [11] extended with axiomatic type classes [30]. The types and terms of HOL are that of the simply-typed λ-calculus [9] augmented with type constructors, type variables, and term constants.

The types are either type variables (e.g., α, β) or n-ary type constructors, usually written in postfix notation (e.g, α *list*). Nullary type constructors are also called type constants (e.g., *nat*). The binary type constructor $\alpha \to \beta$ is interpreted as the (total) function space from α to β. Type variables can carry type class constraints, which are essentially predicates on the types.

Terms are either constants (e.g., map), variables (e.g., x), function applications (e.g., $f\ x$), or λ-abstractions (e.g., $\lambda x.\ f\ x\ x$). Constants and variables can be functions. HOL formulas are simply terms of type *bool*. The familiar connectives and quantifiers are predefined ($\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\exists$). Constants can be polymorphic; for example, $\mathsf{map} : (\alpha \to \beta) \to \alpha\ \mathit{list} \to \beta\ \mathit{list}$ applies a unary function elementwise to a list of α elements.

Isabelle is a generic theorem prover whose metalogic is an intuitionistic fragment of HOL. In the metalogic, propositions have type *prop*, universal quantification is written $\bigwedge$, implication is written $\Longrightarrow$, and equality is written $\equiv$. The object logic is embedded in the metalogic using a constant $\mathsf{Trueprop} : \mathit{bool} \to \mathit{prop}$, which is normally not printed. Some foundational properties can only be expressed in the metalogic, but they play no role in Sledgehammer. We preserve the distinction between the two levels to avoid distracting the trained Isabelle eye, but readers unfamiliar with Isabelle can safely ignore the distinction.

Types are inferred using Hindley–Milner inference. Type annotations $: \tau$ give rise to additional constraints that further restrict the inferred types. A classic example where type annotations are needed is $2 + 2 = 4$. Without type annotations, the formula is parsed as $(2\!:\!\alpha) + (2\!:\!\alpha) = (4\!:\!\alpha)$, where α belongs to the *numeric* type class, which defines basic numeric operators and syntax but imposes no semantics on the "numbers." An annotation is necessary to make the formula provable—e.g., $(2\!:\!\mathit{int}) + 2 = 4$. A single annotation is sufficient because of the constraints arising from the most general types of the involved operators: $\mathrm{op} + : \alpha \to \alpha \to \alpha$ and $\mathrm{op} = : \alpha \to \alpha \to \mathit{bool}$.

For both types and terms, Isabelle distinguishes two kinds of free variable: *schematic variables*, which can be instantiated, and *nonschematic variables*, which stand for fixed, unknown entities. When stating a conjecture and proving it, the type and term variables are normally fixed, and once it is proved, they become schematic so that users of the lemma can instantiate them when applying the lemma.

3 The Translation Pipeline

The translation from an ATP proof to an Isar proof involves two main intermediate data structures. The ATP proof is first parsed and translated into a *proof by contradiction* with the same structure but with HOL formulas instead of first-order formulas. The proof is then transformed into a *direct proof*, from which Isar proof text is synthesized. Various operations are implemented on these data structures to enhance the proof.

ATP Proof. Paulson and Susanto had the foresight to choose TSTP (Thousands of Solutions for Theorem Provers) [28] as input format for their prototype. Among the automatic provers they wanted to integrate with Isabelle, only E [25] supported the format at the time. Nowadays, most provers feature some support for TSTP.

TSTP specifies the basic syntax for representing proofs as a directed acyclic graph of inferences. A single parser can be used to integrate all provers that can generate the syntax. However, the format does not mandate any proof system; hence, interfacing a new ATP usually requires some work, especially for processing inferences that introduce new symbols (e.g., skolemization). Isar proof construction is currently supported for the resolution provers E and Vampire [24] and the unit-equality prover Waldmeister [13]. For the future, we are also interested in the higher-order provers LEO-II [2] and Satallax [8], which are partly integrated in Sledgehammer already [26].

SPASS generates proofs in its custom DFG format only (even though it can parse TPTP FOF [27]). Fortunately, DFG is based on similar concepts and can be represented using the same data structure as TSTP in memory, so it is also supported to a large extent.

Proof by Contradiction. The ATP proof is translated into an Isabelle proof by contradiction. This step preserves the graph structure of the proof, but the nodes are labeled by HOL formulas.

Some consolidation can already take place at this level. ATPs tend to record many more inferences than are interesting to Isabelle users. For example, trivial operations such as clausification and variable renaming produce linear inference chains that can be collapsed.

This translation corresponds largely to the work by Paulson and Susanto. We refer to their paper [23] for details. In particular, they describe how HOL terms, types, and type classes are reconstructed from their encoded FOL form. Their code had to be adapted to cope with the variety of type encodings supported by newer versions of Sledgehammer [5], but nonetheless their description fairly accurately describes the current state of affairs.

Direct Proof. The proof redirection algorithm, presented in the companion paper [4], takes a proof by contradiction as the input and produces a direct proof. The latter can be regarded as a fragment of Isar proofs. The abstract syntax of proofs (π) and inferences (ι) is given by the production rules

$$\begin{aligned}\pi &::= (\texttt{fix}\ x^*)^*\ (\texttt{assume}\ l\colon \phi)^*\ \iota^* \\ \iota &::= \texttt{prove}\ q^*\ l\colon \phi\ l^*\ \pi^* \\ &\quad\ \mid \texttt{obtain}\ q^*\ x^*\ \texttt{where}\ l\colon \phi\ l^*\ \pi^*\end{aligned}$$

where x ranges over HOL variables (which may be of function types), ϕ over HOL formulas, l over Isar fact labels (names), and q over Isar qualifiers (`then` and `show`). Asterisks (*) denote repetition. Nested proof blocks are possible, as indicated by the syntax π^*.

A `fix` command fixes the specified variables in the local context, and `assume` enriches the context with an assumption. Standard inferences are performed using `prove`. Its variant `obtain` proves the existence of HOL variables for which a property holds; the variables are added to the context.

Once the direct proof is constructed, it is iteratively compressed and preplayed. Finally, qualifiers are introduced: `then` indicates that the previous fact is needed to prove the current fact, whereas `show` is required for the last inference in the top-level block. The `then` keyword is only a convenience; the same effect can be achieved less elegantly using labels. At the end, useless labels are removed, and the remaining labels are changed to *f1*, *f2*, etc.

Isar Proof. The final step of the translation pipeline produces a textual Isar proof. This step is straightforward, but some care is needed to generate strings that can be parsed back by Isabelle. This is especially an issue for formulas, where type annotations might be needed.

Example. The following Isabelle theory fragment declares a two-valued *state* datatype, defines a flip function, and states a conjecture about flip:

`datatype` *state* = On | Off

`fun` flip : *state* → *state* `where`
flip On = Off |
flip Off = On

`lemma` flip $x \neq x$

Invoking Sledgehammer launches a collection of ATPs (typically, E, SPASS, Vampire, and Z3). The conjecture is easy, so they rapidly return. Vampire delivers the following proof, presented in a slightly abbreviated TSTP-like format:

51	axiom	$\mathsf{flip}(\mathsf{on}) = \mathsf{off}$	*flip_simps_1*	
52	axiom	$\mathsf{flip}(\mathsf{off}) = \mathsf{on}$	*flip_simps_2*	
55	axiom	$\neg\ \mathsf{off} = \mathsf{on}$	*state_distinct_1*	
57	axiom	$\forall X_3\ (\neg\ \mathsf{state}(X_3) = \mathsf{on} \longrightarrow \mathsf{state}(X_3) = \mathsf{off})$	*state_exhaust*	
58	axiom	$\mathsf{state}(\mathsf{s}) = \mathsf{s}$	*type_of_s*	
774	conj	$\neg\ \mathsf{flip}(\mathsf{s}) = \mathsf{s}$	*goal*	
775	neg_conj	$\neg\neg\ \mathsf{flip}(\mathsf{s}) = \mathsf{s}$	774	negate
776	neg_conj	$\mathsf{flip}(\mathsf{s}) = \mathsf{s}$	775	flatten
781	plain	$\mathsf{off} \neq \mathsf{on}$	55	flatten
892	plain	$\forall X_0\ (\neg\ \mathsf{state}(X_0) = \mathsf{on} \longrightarrow \mathsf{state}(X_0) = \mathsf{off})$	57	rectify
893	plain	$\forall X_0\ (\mathsf{state}(X_0) \neq \mathsf{on} \longrightarrow \mathsf{state}(X_0) = \mathsf{off})$	892	flatten
1596	plain	$\forall X_0\ (\mathsf{state}(X_0) = \mathsf{on} \vee \mathsf{state}(X_0) = \mathsf{off})$	893	ennf_trans
2238	neg_conj	$\mathsf{flip}(\mathsf{s}) = \mathsf{s}$	776	cnf_trans
2239	plain	$\mathsf{state}(\mathsf{s}) = \mathsf{s}$	58	cnf_trans
2287	plain	$\mathsf{flip}(\mathsf{on}) = \mathsf{off}$	51	cnf_trans
2288	plain	$\mathsf{flip}(\mathsf{off}) = \mathsf{on}$	52	cnf_trans
2375	plain	$\mathsf{off} \neq \mathsf{on}$	781	cnf_trans
2485	plain	$\forall X_0\ (\mathsf{state}(X_0) = \mathsf{off} \vee \mathsf{state}(X_0) = \mathsf{on})$	1596	cnf_trans
3342	plain	$\mathsf{on} = \mathsf{s} \vee \mathsf{state}(\mathsf{s}) = \mathsf{off}$	2239, 2485	superpos
3362	plain	$\mathsf{on} = \mathsf{s} \vee \mathsf{off} = \mathsf{s}$	3342, 2239	fwd_demod
3402	neg_conj	$\mathsf{flip}(\mathsf{on}) = \mathsf{on} \vee \mathsf{off} = \mathsf{s}$	2238, 3362	superpos
3404	neg_conj	$\mathsf{off} = \mathsf{on} \vee \mathsf{off} = \mathsf{s}$	3402, 2287	fwd_demod
3405	neg_conj	$\mathsf{off} = \mathsf{s}$	3404, 2375	subsum_res
3407	neg_conj	$\mathsf{flip}(\mathsf{off}) = \mathsf{off}$	3405, 2238	bwd_demod
3408	neg_conj	$\mathsf{off} = \mathsf{on}$	3407, 2288	fwd_demod
3409	neg_conj	$\bot$	3408, 2375	subsum_res

The formulas used from the original problem are listed first. Each line gives a formula number, a role, and a FOL formula. Any problem formula that can be used to prove the conjecture is an axiom for the automatic prover, irrespective of its status in Isabelle (lemma, definition, or actual axiom). The rightmost columns indicate how the formulas was arrived at: Either it appeared in the original problem, in which case its identifier is given (e.g., *flip_simps_1*), or it was derived from one or more already proved formulas using a Vampire-specific proof rule.

If Sledgehammer's *isar_proofs* option is enabled, textual Isar proof reconstruction is attempted. The Isabelle proof by contradiction for the ATP proof above is as follows:

775	$\mathsf{flip}\ s = s$	$\neg$ *goal*
3402	$\mathsf{flip\ On} = \mathsf{On} \lor \mathsf{Off} = s$	775, *state.exhaust*
3404	$\mathsf{Off} = \mathsf{On} \lor \mathsf{Off} = s$	3402, *flip.simps*(1)
3405	$\mathsf{Off} = s$	3404, *state.distinct*(1)
3407	$\mathsf{flip\ Off} = \mathsf{Off}$	775, 3405
3409	False	3407, *flip.simps*(2), *state.distinct*(1)

Linear inference chains are drastically compressed, and the lemmas

state.distinct(1):	$\mathsf{Off} \neq \mathsf{On}$
state.exhaust:	$(y = \mathsf{On} \Longrightarrow P) \Longrightarrow (y = \mathsf{Off} \Longrightarrow P) \Longrightarrow P$
flip.simps(1):	$\mathsf{flip\ On} = \mathsf{Off}$
flip.simps(2):	$\mathsf{flip\ Off} = \mathsf{On}$

are referenced by name rather than repeated. The passage from FOL to HOL also eliminates encoded type information, such as the state function and the auxiliary axiom *type_of_s*. After redirection, the proof becomes

```
prove [] 3407: “flip Off ≠ Off” [flip.simps(2), state.distinct(1)] []
prove [] 3405: “flip s ≠ s ∨ Off ≠ s [3407] []
prove [] 3404: “flip s ≠ s ∨ Off ≠ s ∧ Off ≠ On [3405, state.distinct(1)] []
prove [] 3402: “flip s ≠ s ∨ flip On ≠ On ∧ Off ≠ s [3404, flip.simps(1)] []
prove [show] 775: “flip s ≠ s” [3402, state.exhaust] []
```

Compression and cleanup simplify the proof further:

```
prove [] ϵ: “flip Off ≠ Off” [flip.simps(2), state.distinct(1)] []
prove [then, show] ϵ: “flip s ≠ s” [flip.simps(1), state.distinct(1), state.exhaust] []
```

From this simplified direct proof, the Isar proof is easy to produce:

```
proof –
  have Off ≠ flip Off by (metis flip.simps(2) state.distinct(1))
  thus flip s ≠ s by (metis flip.simps(1) state.distinct(1) state.exhaust)
qed
```

4 Skolemization

The typical architecture of modern first-order provers combines a clausifier and a CNF-based reasoning core. It is the clausifier's duty to skolemize the problem and move the nonskolemizable quantifiers to the front of the formulas, where they can be omitted. Sledgehammer historically performed clausification itself, using a naive exponential application of distributive laws. This was changed a few years ago to use the ATPs' native clausifiers, which generate a polynomial number of clauses [3, §6.6.1]. Skolemization transforms a formula into an equisatisfiable, but not equivalent, formula. As a result, it must be treated specially when reconstructing the proof. Simply invoking *metis*, as done in Paulson and Susanto's prototype, will not work to replay skolemization inferences.

Conjecture and axioms are treated differently because of their different polarities. By convention, the axioms are positive and the conjecture is negative.[2] In the positive case, skolemization eliminates the essentially existential quantifiers (i.e., the positive occurrences of $\exists$ and the negative occurrences of $\forall$). In the negative case, it eliminates the essentially universal quantifiers. Negative skolemization is usually called dual skolemization or herbrandization [12].

E and Vampire explicitly record skolemization inferences in their proof, and fortunately they do it in the same way. On the other hand, SPASS's proofs are expressed in terms of the clausified problem; we have some ideas on how to recover the missing information but have yet to try them out.

The Positive Case. We start with the easier, positive case. Consider the following concrete but archetypal extract from an E or Vampire proof:

11	axiom	$\forall X\, \exists Y\, \mathsf{p}(X, Y)$	*exists_P*	
53	plain	$\forall X\, \mathsf{p}(X, \mathsf{y}(X))$	11	skolem

In Isar, a similar effect is achieved using the `obtain` command:

`obtain` y `where` $\forall x.\ \mathsf{P}\, x\, (y\, x)$ `by` (*metis exists_P*)

In the abstract Isar-like data structure that stores direct proofs, the inference is represented as

`obtain` [] [y] `where` 53: "$\forall x.\ \mathsf{P}\, x\, (y\, x)$" [*exists_P*] []

The approach works for arbitrary quantifier prefixes. All essentially existential variables are eliminated simultaneously. For example, the ATP proof fragment

18	axiom	$\forall V\, \exists W\, \forall X\, \exists Y\, \forall Z\, \mathsf{q}(V, W, X, Y, Z)$	*exists_Q*	
90	plain	$\forall V\, \forall X\, \forall Z\, \mathsf{q}(V, \mathsf{w}(V), X, \mathsf{y}(V, X), Z)$	18	skolem

is translated to

[2] This choice is justifiable from the point of view of an automatic prover that attempts to derive $\bot$ from a set of axioms and a negated conjecture, because all the premises it starts from and the formulas it derives are then considered positive.

```
obtain [] [w,y] where 90: "∀v x z. Q v (w v) x (y v x) z" [exists_Q] []
```

Reconstruction crucially depends not only on *metis*'s clausifier but also on its support for mildly higher-order problems, because of the implicit existential quantification over the Skolem function symbols in `obtain`. Indeed, *metis* is powerful enough to prove a weak form of the HOL axiom of choice:

```
lemma (∀x. ∃y. P x y) ⟹ ∃f. ∀x. P x (f x)
by metis
```

Of course, nothing is derived ex nihilo: *metis* can only prove the formula because its clausifier depends on the axiom of choice in the first place. Furthermore, *metis* will succeed only if its clausifier puts the arguments to the Skolem functions in the same order as in the proof text. This is not difficult to ensure in practice: Both E and *metis* respect the order in which the universal variables are bound, whereas Vampire uses the opposite order, which is easy to reverse.

Positive skolemization suffers from a technical limitation connected to polymorphism. Lemmas containing polymorphic skolemizable variables cannot be reconstructed, because the variables introduced by `obtain` must have a ground type. An easy workaround would be to relaunch Sledgehammer with a monomorphizing type encoding [5, §3] to obtain a more suitable ATP proof. A more challenging alternative would involve detecting which monomorphic instances of the problematic lemmas are needed and re-engineer the proof accordingly.

The Negative Case. In the ATPs, negative skolemization of the conjecture is simply reduced to positive skolemization of the negated conjecture. For example:

```
25  conj      ∀V ∃W ∀X ∃Y ∀Z q(V, W, X, Y, Z)      goal
41  neg_conj  ¬ ∀V ∃W ∀X ∃Y ∀Z q(V, W, X, Y, Z)    25  negate
43  neg_conj  ¬ ∃W ∃Y q(v, W, x(W), Y, z(W, Y))    41  skolem
```

However, once the proof has been turned around in Sledgehammer, the last two lines are unnegated and exchanged: First, a proof of the (unnegated) conjecture is found for specific fixed variables (cf. formula 43 above); then these are generalized into quantified variables (cf. formula 41). A natural name for this process is *un-herbrandization*. In Isar, the `fix` command achieves a similar effect, as in the example below:

```
lemma ⋀x. R x
proof –
  fix x
  ⟨core of the argument⟩
  show R x ...
qed
```

However, this works only for the outermost universal quantifiers. Since we cannot expect users to always state their conjectures in this format, we must generally use a nested proof block, enclosed in curly braces. Thus, the ATP proof fragment presented above is translated to

```
lemma $\forall v.\ \exists w.\ \forall x.\ \exists y.\ \forall z.\ \mathsf{Q}\ v\ w\ x\ y\ z$
proof –
  { fix $v\ x\ z$
    $\langle$core of the argument$\rangle$
    have $\exists w\, y.\ \mathsf{Q}\ v\ w\ (x\ w)\ y\ (z\ w\ y)$ by (metis ...) }
  thus $\forall v.\ \exists w.\ \forall x.\ \exists y.\ \forall z.\ \mathsf{Q}\ v\ w\ x\ y\ z$ by metis
qed
```

Seen from outside, the nested block proves the formula $\bigwedge v\,x\,z.\ \exists w\,y.\ \mathsf{Q}\ v\ w\ (x\ w)\ y\ (z\ w\ y)$. From there, *metis* derives the desired formula $\forall v.\ \exists w.\ \forall x.\ \exists y.\ \forall z.\ \mathsf{Q}\ v\ w\ x\ y\ z$, in which the quantifiers alternate arbitrarily. In the data structure that stores direct Isar-like proofs, the proof would be represented as

```
prove [] 41: "$\forall v.\ \exists w.\ \forall x.\ \exists y.\ \forall z.\ \mathsf{Q}\ v\ w\ x\ y\ z$" []
  [fix [$v$,$x$,$z$]
   $\langle$core of the argument$\rangle$
   prove [] 43: "$\exists w\, y.\ \mathsf{Q}\ v\ w\ (x\ w)\ y\ (z\ w\ y)$" [...] []]
```

An easy optimization, which is not yet implemented, would be to omit the nested proof block for conjectures of the form $\bigwedge x_1 \ldots x_n.\ \phi$, where ϕ contains no essentially universal quantifiers. It should also be possible to move the inferences that do not depend on the herbrandized symbols outside the nested block.

Alternative Approaches. Given a HOL problem, the *metis* method clausifies it and translates it to FOL, invokes the first-order prover Metis, and replays the Metis inferences using suitable HOL tactics. Skolemization is simulated using Hilbert's choice operator ε [23]; for example, $\forall x.\ \exists y.\ \mathsf{P}\ x\ y$ is skolemized into $\forall x.\ \mathsf{P}\ x\ (\varepsilon y.\ \mathsf{P}\ x\ y)$. A newer experimental skolemizer exploits Isabelle's schematic variables to eliminate the dependency on Hilbert's choice [3, §6.6.7], only requiring the weak axiom of choice to move the existentials to the front. Whichever approach is used, Sledgehammer's textual proof construction exploits *metis*'s machinery (and the reduction of HOL to FOL) instead of replicating it textually.

Other ATP-based proof methods or tactics must also cope with skolemization. Isabelle's *smt* method [7] relies on Hilbert's choice, whereas HOL(y)Hammer's proof reconstructor [15] depends only on the weak axiom of choice. Another option is to trust the ATP's clausifier, leaving it to the user to inspect the generated clausification axioms; this is the approach implemented for reconstructing proofs found by Miz$\mathbb{AR}$ [1]. Finally, a radical approach, designed for textual proof reconstruction in Coq, is to replace Skolem function symbols by predicate symbols and adjust the proof accordingly, a process known as deskolemization [10].

5 Type Annotations

To ensure that types are inferred correctly when the generated HOL formulas are parsed again by Isabelle, it is necessary to introduce type annotations. However, redundant annotations should be avoided: If we insisted on annotating each subterm, the simple equation $xs = ys$, where xs and ys range over lists of integers, would be rendered as

$$((\mathrm{op} = : \mathit{int\ list} \to \mathit{int\ list} \to \mathit{bool})\ (xs : \mathit{int\ list}) : \mathit{int\ list} \to \mathit{bool})\ (ys : \mathit{int\ list}) : \mathit{bool}$$

The goal is not to make the Hindley–Milner inference redundant but rather to guide it.

Paulson and Susanto's prototype generates no type annotations at all. Isabelle provides alternative print modes (e.g., one mode annotates all bound variables at the binding site) but none of them is complete. This may seem surprising to users familiar with other proof assistants, but Isabelle's extremely flexible syntax, combined with type classes, means that some terms cannot be parsed back.

We implemented a custom "print mode" for Sledgehammer, which might become an official Isabelle mode in a future release. The underlying algorithm computes a locally minimal set of type annotations for a formula and inserts the annotations. In Isabelle, type annotations are represented by a polymorphic constant $\mathsf{ann}_\tau : \tau \to \tau$ that can be thought of as the identity function. The term $\mathsf{ann}_\tau\ t$ is printed as $t : \tau$. In the presentation below, the notation t^τ indicates that term t has type τ.

The Algorithm. Given a well-typed formula ϕ to annotate, the algorithm starts by replacing all the types in ϕ by the special placeholder _ (Isabelle's "dummy" type). It then infers the most general types for ϕ using Hindley–Milner, resulting in a formula $\phi^\star$ in which the placeholders are instantiated. Next, it computes the substitution $\rho = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_m \mapsto \tau_m\}$ such that $\phi^\star\rho = \phi$, which must exists if ϕ is well-typed and the inferred types in $\phi^\star$ are the most general. Finally, the algorithm inserts type annotations of the form $: \tau$ that *cover* all the type variables α_i in ρ's domain—i.e., such that each type variable α_i occurs in at least one type annotation.

The last step is where the complexity arises. The algorithm assigns a cost to each candidate site t^τ in ϕ where a type annotation can be inserted. The cost is given as a triple of numbers:

$$\text{cost of } t^\tau = (\text{size of } \tau,\ \text{size of } t,\ \text{preorder index of } t \text{ in } \phi)$$

Triples are compared lexicographically. The first two components encode a preference for smaller annotations and smaller annotated terms. The third component resolves ties by preferring annotations occurring closer to the beginning of the printed formula. All subterms of ϕ are potential candidates to carry type annotations. (It would be desirable to consider the binding sites of variables in quantifiers and λ-abstractions as candidates as well, but unfortunately these are simply name–type pairs and not terms in Isabelle.) Each site t^τ is also associated with the set of type variables α_i it covers.

The goal is to compute a minimal set of sites that completely covers all type variables. The resulting cost need not be a global minimum, though; computing the minimum amounts to solving the weighted set cover problem, which is NP-hard [16]. One could probably use a SAT solver to solve the problem efficiently, but we prefer a more direct greedy approach, which is polynomial and produces satisfactory results in practice.

Starting with the set of all possible sites, the algorithm iteratively removes the most expensive redundant site until the set is minimal in the sense that removing any site from it would make it incomplete. This reverse greedy approach ensures that a minimal set will be reached eventually. In contrast, the classical greedy approach could yield a too large set: For the term $\mathsf{h}^{nat\to real}\ \mathsf{c}^{nat}$ generalized to $\mathsf{h}^{\alpha\to\beta}\ \mathsf{c}^{\alpha}$, it would first pick c to cover α, only to find out that h must be annotated as well to cover β, making the first site redundant.

The names of the variables α_i introduced in $\phi^\star$ are irrelevant as long as they are fresh. In a postprocessing step, variables that occurs only once as a subtype in $\tau_1,\ldots,\tau_m$ are replaced by _, and annotations $:\tau$ that cover only variables converted to _ are omitted. Thus, the formula $\mathsf{length}\ ([] : \alpha\ list) = 0$ is printed as $\mathsf{length}\ [] = 0$ without undesirable gain of generality.

Example. Let $\mathsf{fst} : \alpha\times\beta\to\alpha$ and $\mathsf{snd} : \alpha\times\beta\to\beta$ be polymorphic constants that extract the components of a pair. Suppose the formula ϕ to annotate is

$$\forall x\,y.\ \exists p.\ \mathsf{fst}\ p = x \wedge \mathsf{snd}\ p = y$$

with $x : nat$ and $y : real$. Inside Isabelle, the formula's subterms carry type information (except the bound variables):

$$\begin{array}{l}\mathsf{All}^{(nat\to bool)\to bool}\ (\lambda x^{nat}.\ \mathsf{All}^{(real\to bool)\to bool}\ (\lambda y^{real}.\ \mathsf{Ex}^{(nat\times real\to bool)\to bool}\ (\lambda p^{nat\times real}.\\ \quad (\mathrm{op}\ \vee)^{bool\to bool\to bool}\ ((\mathrm{op}\ =)^{nat\to nat\to bool}\ (\mathsf{fst}^{nat\times real\to nat}\ p)\ x)\\ \qquad ((\mathrm{op}\ =)^{real\to real\to bool}\ (\mathsf{snd}^{nat\times real\to real}\ p)\ y))))\end{array}$$

Replacing the types with _ yields the formula

$$\mathsf{All}^{\_}\ (\lambda x^{\_}.\ \mathsf{All}^{\_}\ (\lambda y^{\_}.\ \mathsf{Ex}^{\_}\ (\lambda p^{\_}.\ (\mathrm{op}\ \vee)^{\_}\ ((\mathrm{op}\ =)^{\_}\ (\mathsf{fst}^{\_}\ p)\ x)\ ((\mathrm{op}\ =)^{\_}\ (\mathsf{snd}^{\_}\ p)\ y))))$$

from which type inference produces the formula $\phi^\star$:

$$\begin{array}{l}\mathsf{All}^{(\alpha\to bool)\to bool}\ (\lambda x^{\alpha}.\ \mathsf{All}^{(\beta\to bool)\to bool}\ (\lambda y^{\beta}.\ \mathsf{Ex}^{(\alpha\times\beta\to bool)\to bool}\ (\lambda p^{\alpha\times\beta}.\\ \quad (\mathrm{op}\ \vee)^{bool\to bool\to bool}\ ((\mathrm{op}\ =)^{\alpha\to\alpha\to bool}\ (\mathsf{fst}^{\alpha\times\beta\to\alpha}\ p)\ x)\\ \qquad ((\mathrm{op}\ =)^{\beta\to\beta\to bool}\ (\mathsf{snd}^{\alpha\times\beta\to\beta}\ p)\ y))))\end{array}$$

The substitution entailed by ϕ and $\phi^\star$ is $\rho = \{\alpha\mapsto nat,\ \beta\mapsto real\}$. There are several possible ways to annotate the formula so as to cover both α and β, including

$$\begin{array}{c}\forall x\,y.\ \exists p.\ \mathsf{fst}\ (p : nat\times real) = x \wedge \mathsf{snd}\ p = y\\ \forall x\,y.\ \exists p.\ (\mathsf{fst}\ p : nat) = x \wedge (\mathsf{snd}\ p : real) = y\\ \forall x\,y.\ \exists p.\ \mathsf{fst}\ p = (x : nat) \wedge \mathsf{snd}\ p = (y : real)\end{array}$$

The third formula is the one produced by the reverse greedy algorithm. It is arguably the most aesthetically pleasing of the three, because both the annotated terms and the types are atomic.

Incidentally, the annotations could have been omitted in this example because the property holds generally for arbitrary types α and β, but this cannot always be relied upon. Moreover, omitting the type annotations is not completely harmless because of the poor interaction between skolemization and polymorphism.

6 Proof Preplay

Isar proofs generated from ATP proofs sometimes fail. We already mentioned that skolemization is not supported for polymorphic variables (cf. Section 4). The TSTP or DFG parser occasionally goes wrong if it encounters unexpected (undocumented) syntax. The ATP proof can also contain inferences that are ill-typed from an Isabelle point of view—despite the use of globally sound encodings, individual inferences can violate the type discipline. Moreover, the proof reconstruction code is not bug-free. And even in the absence of errors, the Isar proofs can fail because *metis* fails to discharge a proof obligation within a reasonable amount of time.

As the experience with Sledgehammer-generated *metis* one-liners has shown, it is advantageous to try out, or *preplay*, the proofs before presenting them to users [3, § 6.6.6]. The proofs are then printed together with timing information, including warnings about timeouts or failures. Since Sledgehammer invokes multiple ATPs in parallel, users must often choose between several one-liners and structured Isar proofs. Based on the preplay information, they can make an informed decision while being spared the tedium of having to test the proofs manually.

Until recently, users had to enable Isar proof reconstruction to benefit from it. The preplay mechanism makes a new hybrid mode possible, in which an Isar proof is generated whenever the one-liner times out or fails. Even when the Isar proof is not entirely correct, it is easier to repair than the corresponding monolithic one-liner—for example, by adding some parentheses to guide the parsing or by replacing a failing or too slow *metis* call by a hand-written proof.

The following example, based on an Isabelle formalization of the Robbins conjecture [29], demonstrates preplaying. When invoked on the goal $x \sqcup -x = -x \sqcup --x$ and configured to use Waldmeister, Sledgehammer produces the following output:

Try this: `by` (*metis huntington sup_assoc sup_comm*) (> 3 s).

Structured proof (54 steps, 1.33 s):

```
proof –
  have f1: $\bigwedge x_1\, x_2.\ -(-x_1 \sqcup x_2) \sqcup -(-x_1 \sqcup -x_2) = x_1$
    by (metis huntington sup_comm)
  have f2: $\bigwedge x_1\, x_2\, x_3.\ x_1 \sqcup (x_2 \sqcup x_3) = x_3 \sqcup (x_1 \sqcup x_2)$
    by (metis sup_assoc sup_comm)
  have $\bigwedge x_1\, x_2\, x_3.\ x_1 \sqcup (x_2 \sqcup x_3) = x_2 \sqcup x_1 \sqcup x_3$
    by (metis sup_assoc sup_comm)
  hence f3: $\bigwedge x_1\, x_2\, x_3.\ x_1 \sqcup (x_2 \sqcup x_3) = x_2 \sqcup (x_1 \sqcup x_3)$
    by (metis sup_assoc)
  have f4: $\bigwedge x_1\, x_2\, x_3.\ x_1 \sqcup (x_2 \sqcup x_3) = x_3 \sqcup (x_2 \sqcup x_1)$
    using f2 by (metis sup_comm)
  have f5: $\bigwedge x_1\, x_2.\ x_1 = -(x_2 \sqcup -x_1) \sqcup -(-x_1 \sqcup -x_2)$
    using f1 by (metis sup_comm)
  hence f6: $\bigwedge x_1\, x_2.\ x_1 = -(x_2 \sqcup -x_1) \sqcup -(-x_2 \sqcup -x_1)$
    by (metis sup_comm)
      ⋮
  hence x ⊔ −−−x = x ⊔ (−(−x ⊔ x) ⊔ −(−−x ⊔ −−x))
    using f10 by metis
  hence x ⊔ −−−x = x ⊔ −x
    using f12 by metis
  hence −−x = −(x ⊔ −x) ⊔ −(−x ⊔ −−−x)
    using f6 by metis
  hence −−x = −(x ⊔ −x) ⊔ −(−x ⊔ −x)
    using f22 by metis
  hence −−x = x
    using f5 by metis
  thus x ⊔ −x = −x ⊔ −−x
    by (metis sup_comm)
qed
```

Waldmeister found a proof involving the same three lemmas over and over (*huntington*, *sup_assoc*, and *sup_comm*). However, *metis* fails to re-find the proof within 3 seconds, as indicated by the mention "> 3 s" on the first line. (Indeed, *metis* stands no chance even if given several minutes.) In contrast, the above (abridged) 54-step Isar proof was replayed in 1.33 seconds. Users can click it to insert it in their proof text and move on to the next conjecture.

Behind the scenes, the Isar proof preplay procedure starts by enriching the context with all the local facts introduced in the proof (*f1*, *f2*, etc.). For each inference $\Phi \vdash \phi$, it measures the time *metis* takes to deduce ϕ from Φ and stores it in a data structure. The total is printed at the end, with a '>' prefix if any of the *metis* calls timed out. In the rare event that a *metis* call failed prematurely, Sledgehammer displays the mention "may fail" in the banner.

An alternative approach would have been to have Isabelle parse the Isar proof using its usual interfaces, thereby covering more potential sources of error. For example, with our approach the Isabelle terms are not printed and reparsed; because of Isabelle's flexible syntax, parsing is problematic even if enough type annotations are inserted. On the other hand, the better coverage would come at the price of additional overhead, and it is not clear how to achieve it technically. More importantly, the alternative approach offers no way to collect timing information on a per-step basis. This information is essential for proof compression, as we will see in the next section, and recomputing it would waste the user's time.

Currently, *metis* is invoked to reconstruct each ATP inference in Isabelle. With proof preplay in place, it should be easy to try out other proof methods. To reconstruct proofs with theory-specific or higher-order reasoning, we would need both to appeal to existing decision procedures in Isabelle (e.g., for linear arithmetic) and develop dedicated methods.

7 Proof Compression

The generated Isar proofs can involve dozens or hundreds of steps. It is usually beneficial to compress them. Compressed proofs can be faster to recheck; for example, when the Robbins proof from Section 6 is compressed from 54 to 29 steps, Isabelle also takes nearly half a second less to process it. Moreover, many users prefer concise Isar proofs, either because they want to avoid cluttering their theory files or because they find the shorter proofs simpler to understand. Of course, compression can also be harmful: A *metis* one-liner is nothing but an Isar proof compressed to the extreme, and it can be both very slow and very cryptic.

Whereas intelligibility is in the eye of the beholder, speed can be measured precisely via preplay. Our compression procedure considers candidate pairs of inferences and performs the merger if the resulting inference is fast enough—no more than 20% slower than the original inferences taken together. This 20% tolerance factor embodies a trade-off between processing speed and conciseness. Given the inferences $\Phi_1 \vdash \phi_1$ and $\{\phi_1\} \uplus \Phi_2 \vdash \phi_2$, where ϕ_1 is not referenced elsewhere in the proof (in an antecedent), the merged inference is $\Phi_1 \cup \Phi_2 \vdash \phi_2$.

The algorithm consists of the following steps:

1. Initialize the worklist with all inferences $\Phi \vdash \phi$ such that ϕ is referenced only once in the rest of the proof.

2. If the worklist is empty, stop; otherwise, take an inference $\Phi_1 \vdash \phi_1$ from the worklist.

3. Let $\{\phi_1\} \uplus \Phi_2 \vdash \phi_2$ be the unique inference that references ϕ_1. Try to merge the two inferences as described above. If this succeeds, add any emerging singly-referenced facts belonging to $\Phi_1 \cap \Phi_2$ to the worklist.

4. Go to step 2.

Step 2 nondeterministically picks an inference. Our implementation prefers inferences with long formulas, because these clutter the proof more. In step 3, merging the two inferences may give rise to new singly-referenced facts ϕ that were referenced by both ϕ_1 and ϕ_2 (i.e., $\phi \in \Phi_1 \cap \Phi_2$) but not by any other inferences.

The process is guided by *metis*'s performance. Users who want to understand the proof may find that too many details have been optimized away. For them, there is a Sledgehammer option that controls the compression factor, which bounds the number of mergers before the algorithm stops in relation to the length of the uncompressed proof.

8 Conclusion

The latest version of Sledgehammer employs a variety of techniques to improve the readability and efficiency of the generated Isar proofs. Whenever one-line proof reconstruction fails or times out, users are offered detailed, direct Isar proofs that discharge the goal, sometimes after a small amount of manual tuning. Users who are interested in inspecting the proofs can force their generation by passing an option. Related options control preplay and compression.

This work is still in progress. Many aspects could be improved further; we mentioned a few in the previous sections. Our next priority is to identify and rectify any remaining failure cases: Preplaying insulates users from failures, but ideally valid ATP proofs should always lead to valid Isar proofs. We plan to integrate the proof reconstruction code with the "Judgment Day" harness [6] to test it more thoroughly and evaluate its impact on Sledgehammer's success rate.

The next step will be to implement proof manipulation algorithms to simplify the proofs further before presenting them to users. For example, users normally prefer sequential chains of deduction to the spaghetti-like structure of some machine-generated proofs; using appropriate algorithms, it should be possible to minimize the number of jumps or introduce block structure to separate independent subproofs. Similar work has been carried out for human-written proofs [20, 21], but we expect machine proofs to offer more opportunities for refactoring.

Acknowledgment. Tobias Nipkow, Lawrence Paulson, Geoff Sutcliffe, and Josef Urban encouraged us to pursue this research. Stefan Berghofer and Cezary Kaliszyk shared ideas with us on how to address the problem of inserting type annotations. Cezary Kaliszyk, Andrei Popescu, Mark Summerfield, and three anonymous reviewers provided helpful feedback on earlier versions of this paper. The second author's research was financed by the Deutsche Forschungsgemeinschaft (DFG) project Hardening the Hammer (grant Ni 491/14-1).

References

[1] J. Alama. Escape to Mizar from ATPs. In P. Fontaine, R. Schmidt, and S. Schulz, editors, *PAAR-2012*, pages 3–11, 2012.

[2] C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. Progress report on LEO-II, an automatic theorem prover for higher-order logic. In K. Schneider and J. Brandt, editors, *TPHOLs: Emerging Trends*. C.S. Dept., Technische Universität Kaiserslautern, 2007.

[3] J. C. Blanchette. *Automatic Proofs and Refutations for Higher-Order Logic*. Ph.D. thesis, Dept. of Informatics, Technische Universität München, 2012.

[4] J. C. Blanchette. Redirecting proofs by contradiction. In J. C. Blanchette and J. Urban, editors, *PxTP 2013*, 2013.

[5] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. In N. Piterman and S. Smolka, editors, *TACAS 2013*, volume 7795 of *LNCS*, pages 493–507. Springer, 2013.

[6] S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In J. Giesl and R. Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNAI*, pages 107–121. Springer, 2010.

[7] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In M. Kaufmann and L. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

[8] C. E. Brown. Satallax: An automatic higher-order prover. In B. Gramlich, D. Miller, and U. Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012.

[9] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[10] H. de Nivelle. Translation of resolution proofs into short first-order proofs without choice axioms. *Inf. Comput.*, 199(1-2):24–54, 2005.

[11] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[12] J. Herbrand. *Thèses présentées à la Faculté des sciences de Paris pour obtenir le grade de docteur ès sciences mathématiques*. Ph.D. thesis, Science Faculty, Université de Paris, 1930.

[13] T. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. WALDMEISTER—High-performance equational deduction. *J. Autom. Reasoning*, 18(2):265–270, 1997.

[14] J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. Di Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.

[15] C. Kaliszyk and J. Urban. PRocH: Proof reconstruction for HOL Light. In M. P. Bonacina, editor, *CADE-24*, volume 7898 of *LNAI*, pages 267–273. Springer, 2013.

[16] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, IBM Research Symposia Series, pages 85–103. Plenum Press, 1972.

[17] R. Matuszewski and P. Rudnicki. Mizar: The first 30 years. *Mechanized Mathematics and Its Applications*, 4(1):3–24, 2005.

[18] J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[19] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[20] K. Pąk. The methods of improving and reorganizing natural deduction proofs. In *MathUI10*, 2010.

[21] K. Pąk. Methods of lemma extraction in natural deduction proofs. *J. Autom. Reasoning*, 50(2):217–228, 2013.

[22] L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In G. Sutcliffe, E. Ternovska, and S. Schulz, editors, *IWIL-2010*, 2010.

[23] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.

[24] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Comm.*, 15(2-3):91–110, 2002.

[25] S. Schulz. System description: E 0.81. In D. Basin and M. Rusinowitch, editors, *IJCAR 2004*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[26] N. Sultana, J. C. Blanchette, and L. C. Paulson. LEO-II and Satallax on the Sledgehammer test bench. *J. Applied Logic*, 11(1):91–102, 2013.

[27] G. Sutcliffe. The TPTP problem library and associated infrastructure—The FOF and CNF parts, v3.5.0. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[28] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP data-exchange formats for automated theorem proving tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, volume 112 of *Frontiers in Artificial Intelligence and Applications*, pages 201–215. IOS Press, 2004.

[29] M. Wampler-Doty. A complete proof of the Robbins conjecture. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Robbins-Conjecture.shtml`, 2010.

[30] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *TPHOLs 1997*, volume 1275 of *LNCS*, pages 307–322. Springer, 1997.

[31] M. Wenzel. Isabelle/Isar—A generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof—Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*. University of Białystok, 2007.

Author Index

Keyword Index