EPTCS 186

Proceedings of the

Fourth Workshop on

Proof eXchange for Theorem Proving

Berlin, Germany, August 2-3, 2015

Edited by: Cezary Kaliszyk and Andrei Paskevich

Published: 30th July 2015
DOI: 10.4204/EPTCS.186
ISSN: 2075-2180
Open Publishing Association

Preface

This volume of EPTCS contains the proceedings of the Fourth Workshop on Proof Exchange for Theorem Proving (PxTP 2015), held on August 2-3, 2015 as part of the International Conference on Automated Deduction (CADE 2015) in Berlin. The PxTP workshop series brings together researchers working on various aspects of communication, integration, and cooperation between reasoning systems and formalisms.

The progress in computer-aided reasoning, both automated and interactive, during the past decades, made it possible to build deduction tools that are increasingly more applicable to a wider range of problems and are able to tackle larger problems progressively faster. In recent years, cooperation of such tools in larger verification environments has demonstrated the potential to reduce the amount of manual intervention. Examples include the Sledgehammer tool providing an interface between Isabelle and (untrusted) automated provers, and also collaboration of the HOL Light and Isabelle systems in the formal proof of the Kepler conjecture.

Cooperation between reasoning systems relies on availability of theoretical formalisms and practical tools to exchange problems, proofs, and models. The PxTP workshop strives to encourage such cooperation by inviting contributions on suitable integration, translation and communication methods, standards, protocols, and programming interfaces. The workshop welcomes the interested developers of automated and interactive theorem proving tools, developers of combined systems, developers and users of translation tools and interfaces, and producers of standards and protocols. We are interested both in success stories and in descriptions of the current bottlenecks and proposals for improvement.

Previous editions of the workshop took place in Wrocław (2011), Manchester (2012), and Lake Placid (2013).

The workshop features seven regular papers and two invited talks by Georges Gonthier (Microsoft Research) and Bart Jacobs (University of Leuven).

We would like to thank the authors for submitting papers of high quality to these proceedings, the programme committee and external reviewers for diligently reviewing the submissions, and the organisers of CADE 2015 for their help in organizing PxTP 2015.

July 6, 2015

Cezary Kaliszyk
Andrei Paskevich

Organization

Program Committee

- Jesse Alama, Vienna University of Technology, Austria
- Peter Baumgartner, NICTA, Australia
- Jasmin Blanchette, TU München, Germany
- Guillaume Burel, CÉDRIC, ENSIIE, France
- Évelyne Contejean, LRI, CNRS, Université Paris-Sud, France
- Cezary Kaliszyk (PC co-chair), University of Innsbruck, Austria
- Ramana Kumar, University of Cambridge, UK
- Dale Miller, Inria / LIX, École polytechnique, France
- Bruno Woltzenlogel Paleo, Vienna University of Technology, Austria
- Andrei Paskevich (PC co-chair), LRI, Université Paris-Sud, France
- Damien Pous, LIP, CNRS, ENS Lyon, France
- Geoff Sutcliffe, University of Miami, USA
- Laurent Théry, Inria, France
- Cesare Tinelli, University of Iowa, USA
- Josef Urban, Radboud University Nijmegen, The Netherlands

External Reviewers

- Ali Assaf, École Polytechnique, France
- Thibault Gauthier, University of Innsbruck, Austria
- Robbert Krebbers, Radboud University Nijmegen, The Netherlands
- Tomer Libal, Microsoft Research – Inria, France
- Florian Rabe, Jacobs University Bremen, Germany
- Martin Riener, Vienna University of Technology, Austria

Table of Contents

Reflection, of all shapes and sizes

Georges Gonthier
Microsoft Research

Although it is only a technical term for the common operation of quotation, reflection has had an increasingly important role in the design, organisation, and interconnection of machine formalizations of large theories. Reflexion supports a straightforward method for porting statements, deductions or proof techniques between theories, systems, or even logics: quote in one context to pure text, then reinterpret in a new context. While traditional logicians and computer programmers both have mostly focused on generic quoters and interpreters, computer proofs rely on a wider variety of reflexion forms. In a formal proof the interpretation must carry over meaning across contexts (the interpreter must be provably correct), and this is often easier with a shallower embedding in which not everything is quoted.

Reflection can be used at many scales: from the large scale in which complete problems are shipped to be solved in a different system, to the very small scale that mostly mediates between the flexibility of traditional notation and the rigorous regularity of formal representations, through a medium scale where the interpreter performs significant computation and produces readable notation.

This range of uses is supported by a range of implementation techniques. Quotation can be external (tactics, static overloading), or internal (first- or higher-order unification); interpretation can be deep, shallow, or even parallel. Together these techniques provide a toolkit that makes reflection far easier to use effectively. They have been instrumental in my work on the Four Colour and Odd Order theorems, from which this talk will draw several worked examples.

C.Kaliszyk and A. Paskevich (Eds.): 4th Workshop
on Proof eXchange for Theorem Proving (PxTP'15).
EPTCS 186, 2015, pp. 1–1, doi:10.4204/EPTCS.186.1

The VeriFast program verifier and its SMT solver interaction

Bart Jacobs
University of Leuven

VeriFast is a tool that takes as input a set of C or Java source code files, annotated with specifications and logical definitions written in a variant of separation logic, and then, without further user interaction and usually in a matter of seconds, reports either "0 errors found" or a failed symbolic execution trace. In the former case, barring tool bugs, the program does not access unallocated memory, perform data races, or violate API or library preconditions or the user-provided specifications. The tool operates by symbolically executing each function/method, starting from a symbolic state representing an arbitrary state that satisfies the precondition, and checking that the final state satisfies the postcondition. The symbolic state consists of a symbolic store (mapping local variables to terms of first-order logic), a symbolic heap (a list of so-called chunks, interpreted as a separating conjunction of separation logic predicate applications, whose arguments are terms of first-order logic), and the path condition (a formula of first-order logic). Heap effects are dealt with in the tool itself through simple pattern matching, but proof obligations about data values are delegated to an SMT solver. We have taken care to keep hypotheses pushed into the SMT solver light on quantifiers and disjunctions. The axiomatization of inductive datatypes and primitive recursive functions prevents case splitting and (almost) prevents matching loops. Since neither the tool nor the SMT solver perform significant search, performance is predictable and good. VeriFast supports both Z3 and our own Redux (a partial re-implementation of Simplify).

I will first give an introduction to the specification and proof constructs in VeriFast's input language, then talk about how in VeriFast work is split between the front-end symbolic execution and the back-end SMT solver, going into detail on the encoding strategy, and then give a demo of the kind of user experience this produces.

C.Kaliszyk and A. Paskevich (Eds.): 4th Workshop
on Proof eXchange for Theorem Proving (PxTP'15).
EPTCS 186, 2015, pp. 2–2, doi:10.4204/EPTCS.186.2

Importing SMT and Connection proofs as expansion trees

Giselle Reis
INRIA-Saclay, France
giselle.reis@inria.fr

Different automated theorem provers reason in various deductive systems and, thus, produce proof objects which are in general not compatible. To understand and analyze these objects, one needs to study the corresponding proof theory, and then study the language used to represent proofs, on a prover by prover basis. In this work we present an implementation that takes SMT and Connection proof objects from two different provers and imports them both as expansion trees. By representing the proofs in the same framework, all the algorithms and tools available for expansion trees (compression, visualization, sequent calculus proof construction, proof checking, etc.) can be employed uniformly. The expansion proofs can also be used as a validation tool for the proof objects produced.

1 Introduction

The field of proof theory has evolved in such a way to create the most various proof abstractions. Natural deduction, sequent calculus, resolution, tableaux, SAT, are only a few of them, and even within the same formalism there might be many variations. As a result, automated theorem provers will generate different proof objects, usually corresponding to their internal proof representation. The use of distinct formats has some disadvantages: provers cannot recognize each others proofs; proofs cannot be easily compared; all analysis and algorithms need to be developed on a prover by prover basis.

GAPT is a framework for proof theory that is able to represent, process and visualize proofs. Currently it implements the sequent calculus LK (with or without equality rules) for first and higher order classical logic, Robinson's resolution calculus [11], the schematic calculus LKS [4] and expansion trees [8]. GAPT also provides algorithms for translating proofs between some of these formats, for cut-elimination (reductive methods à la Gentzen [5] and CERES [2]), and for cut-introduction (proof compression) [6], as well as an interactive proof visualization tool [3]. But all these tools depend on having proofs to operate on.

In this work we show how to parse and translate SMT and Connection proofs from veriT and leanCoP, respectively, into expansion proofs in GAPT. SMT are unsatisfiability proofs with respect to some theory and, in veriT, these are represented by resolution refutations of a set including (instances of) the axioms of the theory considered and the negation of the input formula. Connection proofs decide first-order logic formulas by connecting literals of opposite polarity in the clausal normal form of the input. These different conceptions of proofs will be unified under the form of expansion proofs, which can be considered a compact representation of sequent calculus proofs.

The advantages of this work is three-fold. First of all, the use of expansion proofs provides a compact representation for otherwise big and hard to grasp proof objects. Using this representation and GAPT's visualization tool, it is easy to see the theorem that was proved and the instances of quantified formulas used. Second of all, the use of a common representation facilitates the comparison of proofs and makes it possible to run and analyse algorithms developed for this representation without the need to adapt it to different formats. In particular, we have been using the imported proofs for experimenting proof compression via introduction of cuts [6]. Finally, it provides a simple sanity-check procedure and the possibility of building LK proofs.

C.Kaliszyk and A. Paskevich (Eds.): 4th Workshop
on Proof eXchange for Theorem Proving (PxTP'15).
EPTCS 186, 2015, pp. 3–10, doi:10.4204/EPTCS.186.3

This paper is organized as follows. Section 2 defines basic concepts and extends the usual definition of expansion trees to accommodate polarities. Section 3 explains how to extract the necessary information from both formats and how it is then used to build expansion trees. Section 4 presents the results of the transformation applied to a database of proofs in the considered formats. It also discusses the advantages of having the proofs as expansion trees. Section 5 discusses some related work and, finally, Section 6 concludes the paper pointing to future work.

2 Expansion proofs

We will work in the setting of first-order classical logic. We introduce now a few basic concepts.

Definition 1 (Polarity in a sequent)**.** *Let $S = A_1, ..., A_n \vdash B_1, ..., B_m$ be a sequent. We will say that formulas on the left side of $\vdash$, i.e, $A_1, ..., A_n$ have* negative *polarity while formulas on the right, i.e., $B_1, ..., B_m$ have* positive *polarity.*

Definition 2 (Polarity)**.** *Let F be a formula and F' a sub-formula of F. Then we can define the* polarity *of F' in F, i.e., F' can be* positive *or* negative *in F, according to the following criteria:*

- *If $F \equiv F'$, then F' has the same polarity as F.*
- *If $F \equiv A \wedge B$ or $F \equiv A \vee B$ or $F \equiv \forall x.A$ or $F \equiv \exists x.A$ and F is* positive *(*negative*), than A and B are* positive *(*negative*).*
- *If $F \equiv A \to B$ and F is* positive *(*negative*), then A is* negative *(*positive*) and B is* positive *(*negative*).*
- *If $F \equiv \neg A$ and $'$ is* positive *(*negative*) then A is* negative *(*positive*).*

Throughout this document we will use 0 for negative polarity, 1 for positive polarity and $\overline{p}$ to denote the opposite polarity of p, for $p \in \{0,1\}$.

Definition 3 (Strong and weak quantifiers)**.** *Let F be a formula. If $\forall x$ occurs positively (negatively) in F, then $\forall x$ is called a* strong (weak) quantifier. *If $\exists x$ occurs positively (negatively) in F, then $\exists x$ is called a* weak (strong) quantifier.

Strong quantifiers in a sequent will be those introduced by the inferences $\forall_r$ and $\exists_l$ in a sequent calculus proof.

Expansion proofs are a compact representation for first and higher order sequent calculus proofs. They can be seen as a generalization of Gentzen's mid-sequent theorem to formulas which are not necessarily prenex [8]. Expansion proofs are composed by expansion trees. An expansion tree of a formula F has this formula as its root. Leaves are atoms occurring in F and inner nodes are connectives or a quantified sub-formula of F. The edges from quantified nodes to its children are labelled with terms that were used to instantiate the outer-most quantifier. We extend the original definition with the notion of formula polarity and use Π and Λ for strong and weak quantifiers respectively in expansion trees.

Definition 4 (Expansion tree)**.** Expansion trees *and a function $\mathsf{Sh}(E,p)$ (for* shallow*), that maps an expansion tree E to a formula with polarity $p \in \{0,1\}$, are defined inductively as follows:*

- *If A is an atom, then A is an expansion tree with top node A and $\mathsf{Sh}(A,p) = A$ for any choice of p.*
- *If E_0 is an expansion tree, then $E = \neg E_0$ is an expansion tree with $\mathsf{Sh}(E,\overline{p}) = \neg\mathsf{Sh}(E_0,p)$.*
- *If E_1 and E_2 are expansion trees and $\circ \in \{\wedge,\vee\}$, then $E = E_1 \circ E_2$ is an expansion tree with $\mathsf{Sh}(E,p) = \mathsf{Sh}(E_1,p) \circ \mathsf{Sh}(E_2,p)$.*
- *If E_1 and E_2 are expansion trees, then $E = E_1 \to E_2$ is an expansion tree with $\mathsf{Sh}(E,p) = \mathsf{Sh}(E_1,\overline{p}) \to \mathsf{Sh}(E_2,p)$.*

- *If* $\{t_1,...,t_n\}$ *is a set of terms and* $E_1,...,E_n$ *are expansion trees with* $\mathsf{Sh}(E_i,p) = A[x/t_i]$*, then* $E = \Lambda x.A +^{t_1} E_1 ... +^{t_n} E_n$ *(denoting a node with n children) is an expansion tree with* $\mathsf{Sh}(E,0) = \forall x.A$ *and* $\mathsf{Sh}(E,1) = \exists x.A$*.*
- *If* E_0 *is an expansion tree with* $\mathsf{Sh}(E_0,p) = A[x/\alpha]$ *for an Eigenvariable* α*, then* $E = \Pi x.A +^{\alpha} E_0$ *is an expansion tree with* $\mathsf{Sh}(E,0) = \exists x.A$ *and* $\mathsf{Sh}(E,1) = \forall x.A$*.*

Expansion trees can be mapped to a quantifier free formula via the *deep* function, which we also redefine taking the polarities into account.

Definition 5. *We define the function* $\mathsf{Dp}(\cdot,p)$ *(for* deep*),* $p \in \{0,1\}$*, that maps an expansion tree to a quantifier free formula of polarity p as:*

- $\mathsf{Dp}(A,p) = A$ *for an atom A.*
- $\mathsf{Dp}(\neg A,p) = \neg\mathsf{Dp}(A,\overline{p})$
- $\mathsf{Dp}(A \circ B,p) = \mathsf{Dp}(A,p) \circ \mathsf{Dp}(B,p)$ *for* $\circ \in \{\wedge,\vee\}$
- $\mathsf{Dp}(A \to B,p) = \mathsf{Dp}(A,\overline{p}) \to \mathsf{Dp}(B,p)$
- $\mathsf{Dp}(\Lambda x.A +^{t_1} E_1 ... +^{t_n} E_n,0) = \bigwedge_{i=1}^{n} \mathsf{Dp}(E_i,0)$
- $\mathsf{Dp}(\Lambda x.A +^{t_1} E_1 ... +^{t_n} E_n,1) = \bigvee_{i=1}^{n} \mathsf{Dp}(E_i,1)$
- $\mathsf{Dp}(\Pi x.A +^{\alpha} E,p) = \mathsf{Dp}(E,p)$

Definition 6 (Expansion sequent)**.** *An* expansion sequent *ε is denoted by* $E_1,...,E_n \vdash F_1,...,F_m$ *where* E_i *and* F_i *are expansion trees. Its* deep sequent *is the sequent* $\mathsf{Dp}(E_1,0),...,\mathsf{Dp}(E_n,0) \vdash \mathsf{Dp}(F_1,1),...,\mathsf{Dp}(F_m,1)$ *and its* shallow sequent *is* $\mathsf{Sh}(E_1,0),...,\mathsf{Sh}(E_n,0) \vdash \mathsf{Sh}(F_1,1),...,\mathsf{Sh}(F_m,1)$*.*

An expansion sequent may or may not represent a proof. To decide whether this is the case, we need to reason on the *dependency relation* in the sequent.

Definition 7 (Domination)**.** *A term t is said to* dominate *a node N in an expansion tree if it labels a parent node of N.*

Definition 8 (Dependency relation)**.** *Let ε be an expansion sequent and let* $<^0_\varepsilon$ *be the binary relation on the occurrences of terms in ε defined as: $t <^0_\varepsilon s$ if there is an x free in s that is an eigenvariable of a node dominated by t. Then* $<_\varepsilon$*, the transitive closure of* $<^0_\varepsilon$*, is called the* dependency relation *of E.*

Definition 9 (Expansion proof)**.** *An expansion sequent is considered an* expansion proof *if its deep sequent is a tautology and the dependency relation is acyclic.*

Intuitively, the dependency relation gives an ordering of quantifier inferences in a sequent calculus proof of the shallow sequent of ε. That is, $t <_\varepsilon s$ means that the existential quantifiers instantiated with t must occur lower in the proof than those instantiated with s. Using this relation it is possible to build an LK proof from an expansion proof [8].

3 Importing

GAPT[1] is a framework for proof transformations implemented in the programming language Scala. It supports different proof formats, such as LK (with or without equality) for first and higher order logic, Robinson's resolution calculus [11], the schematic calculus LKS [4] and, more recently, expansion trees. It provides various algorithms for proofs, such as reductive cut-elimination [5], cut-elimination by resolution [2], cut-introduction [6], Skolemization, and translations between the proof formats. GAPT also comes with `prooftool` [3], an interactive proof visualization tool supporting all these formats.

VeriT and leanCoP are automated theorem provers that produce unsatisfiability (in the shape of a resolution refutation) and connection proofs respectively. Both output the proof objects to a structured

[1] `https://github.com/gapt/gapt`

text file, having in common the fact that all inferences are listed with the operands and the conclusion. We have implemented parsers (using Scala's parser combinators) for both formats in GAPT. By taking the necessary information of each proof file and processing it accordingly, we can build expansion proofs. We explain the kind of processing needed for each format in Sections 3.1 and 3.2.

The expansion tree of a formula with associated substitutions to its bound variables can be defined as follows:

Definition 10. *Let F be a formula in which all bound variables have pairwise distinct names, Σ a set of substitutions for these variables and $p \in \{0,1\}$ a polarity. Assume that each strong quantifier in F is bound to exactly one term in Σ. We define the function $\mathsf{ET}(F,\Sigma,p)$ that translates a formula to an expansion tree as follows:*

- $\mathsf{ET}(A,\Sigma,p) = A$, *where A is an atom.*
- $\mathsf{ET}(\neg A,\Sigma,p) = \neg\mathsf{ET}(A,\Sigma,\overline{p})$.
- $\mathsf{ET}(A \circ B,\Sigma,p) = \mathsf{ET}(A,\Sigma,p) \circ \mathsf{ET}(B,\Sigma,p)$, *for* $\circ \in \{\wedge,\vee\}$.
- $\mathsf{ET}(A \to B,\Sigma,p) = \mathsf{ET}(A,\Sigma,\overline{p}) \to \mathsf{ET}(B,\Sigma,p)$.
- $\mathsf{ET}(\forall x.A,\Sigma,0) = \Lambda x.A +^{t_1} \mathsf{ET}(A\sigma_1,\{\sigma_1\},0) ... +^{t_n} \mathsf{ET}(A\sigma_n,\{\sigma_n\},0)$, *where σ_i is the substitution in Σ mapping x to t_i (n is the number of times the weak quantifier was instantiated).*
- $\mathsf{ET}(\forall x.A,\Sigma,1) = \Pi x.A +^{\alpha} \mathsf{ET}(A\sigma',\{\sigma'\},1)$ *where σ' is the substitution in Σ mapping x to α.*
- $\mathsf{ET}(\exists x.A,\Sigma,0) = \Pi x.A +^{\alpha} \mathsf{ET}(A\sigma',\{\sigma'\},0)$ *where σ' is the substitution in Σ mapping x to α.*
- $\mathsf{ET}(\exists x.A,\Sigma,1) = \Lambda x.A +^{t_1} \mathsf{ET}(A\sigma_1,\{\sigma_1\},1) ... +^{t_n} \mathsf{ET}(A\sigma_n,\{\sigma_n\},1)$, *where σ_i is the substitution in Σ mapping x to t_i (n is the number of times the weak quentifier was instatiated).*

Note that the term α used for the strong quantifiers is determined by the substitution set Σ. If the eigenvariable condition is not satisfied in these substitutions, then the resulting expansion tree will not be a proof of the formula.

Using the $\mathsf{ET}(F,\sigma,p)$ transformation, it is also possible to define the expansion sequent ε from a sequent S.

Definition 11. *Let $S : A_1,...,A_n \vdash B_1,...,B_m$ be a sequent with pairwise distinct bound variables and σ a set of substitutions for those variables such that each strongly quantified variable is bound to exactly one term. Then we define $\mathsf{ET}(S,\sigma)$ as the expansion sequent $\mathsf{ET}(A_1,\sigma,0),...,\mathsf{ET}(A_n,\sigma,0) \vdash \mathsf{ET}(B_1,\sigma,1),...,\mathsf{ET}(B_m,\sigma,1)$.*

Definitions 10 and 11 show how to build an expansion sequent from a sequent and a set of substitutions. The requirement of pairwise distinct variables can be easily satisfied by a variable renaming. The second requirement, that each variable of a strong quantifier is bound only once, might not be true for arbitrary proofs. Fortunately, it holds for the proofs we are dealing with, either because the input problem contains no strong quantifiers, or because the end-sequent is skolemized. On the second case, it is possible to deduce unique Eigenvariables for each strong quantifier and obtain the expansion tree of the un-skolemized formula.

Lemma 1. $\mathsf{Sh}(\mathsf{ET}(F,\sigma,p),p) = F$

Proof. Follows from the definition of $\mathsf{ET}(F,\sigma,p)$ and $\mathsf{Sh}(E,p)$. □

Theorem 1. *A sequent S with substitutions σ, such that each strongly quantified variable in S is bound exactly once, is valid iff the expansion sequent $\mathsf{ET}(S,\sigma)$ is an expansion proof.*

Proof. By the soundness and completeness of expansion sequents [8], we know that an expansion sequent ε is an expansion proof iff its shallow sequent is valid. From Lemma 1 we have that the shallow sequent of $\mathsf{ET}(S,\sigma)$ is S. Therefore, S is valid iff $\mathsf{ET}(S,\sigma)$ is an expansion proof. □

This theorem provides a "sanity-check" for the expansion sequents extracted from proof objects. If it is an expansion proof, we know that, at least, the end-sequent with the given substitutions is a tautology. Note that this does not provide a check for the proof, as it is not validating each inference applied, but only if the claimed instantiations *can* actually lead to a proof.

3.1 SMT proofs

SMT (*Satisfiability Modulo Theory*) is a decision procedure for first-order formulas with respect to a background theory. It can be seen as a generalization of SAT problems. VeriT[2] is an open-source SMT-solver which is complete for quantifier-free formulas with uninterpreted functions and difference logic on reals and integers. For this work we have used the proof objects produced by VeriT on the `QF_UF` (quantifier-free formulas with uninterpreted function symbols) problems of the SMT-LIB[3]. The background theory in this case was the equality theory composed by the axioms (symmetry and reflexivity are implicit):

$$\forall x_0 ... \forall x_n.(x_0 = x_1 \wedge ... \wedge x_{n-1} = x_n \rightarrow x_0 = x_n)$$
$$\forall x_0 ... \forall x_n \forall y_0 ... \forall y_n.((x_0 = y_0 \wedge ... \wedge x_n = y_n \rightarrow f(x_0,...,x_n) = f(y_0,...,y_n))$$
$$\forall x_0 ... \forall x_n \forall y_0 ... \forall y_n.(x_0 = y_0 \wedge ... \wedge x_n = y_n \wedge p(x_0,...,x_n) \rightarrow p(y_0,...,y_n))$$

The proofs generated are composed of CNF transformations and a resolution refutation, whose leaves are either one of the quantifier-free formulas from the input problem or an instance of an equality axiom. The proof object consists of a comprehensive list of labelled clauses used in the resolution proof and their origin. They are either an input clause, without ancestors, or the result of an inference rule on other clauses, which is specified via the labels. VeriT's proof is purely propositional and no substitutions are involved, since the axioms are quantifier-free and contain no free-variables.

The input problem is propositional, therefore the only substitutions needed were the ones instantiating the (weak) quantifiers of the equality axioms[4]. These are found by collecting the ground instances of these axioms occurring on the leaves of the resolution proof and using a first-order matching algorithm. By matching the instances with the appropriate axiom (without the quantifiers), we can obtain the substitutions for the quantified variables. Given those substitutions and the quantified axioms, we can build the expansion trees. It is worth noting that the quantified equality axioms (i.e., transitivity, symmetry, reflexivity, etc.) are build internally in GAPT, since these are not part of the proof object. Also, the reflexivity instances needed are computed separately, since these are implicit in veriT. The expansion tree of the (propositional) input formula can be built with an empty set of substitutions. Since these are unsatisfiability proofs, all expansion trees will be on the left side of the expansion sequent.

3.2 Connection proofs

Connection calculi is a set of formalisms for deciding first-order classical formulas which consists on connecting unifiable literals of opposite polarities from the input. Proof search in these calculi is characterized as goal-oriented and, in general, non-confluent. LeanCoP[5] is a connection based theorem prover that implements a series of techniques for reducing the search space and making proof search feasible

[2] `http://www.verit-solver.org/`
[3] `http://smt-lib.org/`
[4] Observe that we do not need any information from the inference steps.
[5] `http://leancop.de/`

[10]. Although its strategy is incomplete, it achieves very good performance in practice. For this work, leanCoP 2.2 was used. It can be obtained from the CASC24 competition website[6] or, alternatively, executed online at SystemOnTPTP[7].

Given an input problem (a set of axioms and conjectures in the language of first-order logic), leanCoP will negate the axioms, skolemize the formulas and translate them into a disjunctive normal form (DNF). It works with a positive representation of the problem and uses a special DNF transformation that is more suitable for connection proof search [10]. The prover also adds equality axioms when necessary. LeanCoP is able to produce proof objects in four different formats For this work, we have used `leantptp`, which is closer to the TPTP (thousands of problems for theorem provers) specification [12]. The output file is divided in three parts: (1) input formulas; (2) clauses generated from the DNF transformation of the input and equality axioms; and (3) proof description. Each part is described using a set of predicates with the relevant information.

In part (1), the formulas from the input file are listed and named. Their variables are renamed such that they are pairwise distinct. Moreover, formulas are annotated with respect to their role, e.g, axiom or conjecture. Part (2) contains the clauses, in the form of a list of literals, that resulted from the disjunctive normal form transformation. This can either be the regular naive DNF translation or a *definitional clausal form transformation*, which assigns new predicates to some formulas. Each clause is numbered and associated with the name of the formula that generated it. Equality axioms are labelled with a special keyword, since they do not come from any transformation on the input formulas. The proof *per se* is in part (3), where each line is an inference rule. It contains the number of the clause to which the inference was applied, the bindings used (if any) and the resulting clause.

For building the expansion trees of the input formulas we need the substitutions used in the proof and the Skolem terms introduced during Skolemization. The substitutions will be the terms of the expansion tree's weak quantifiers and the Skolem terms, translated to variables, will be the expansion tree's strong quantifier terms. In the leanCoP proofs, Skolem terms have a specific syntax, so they can be identified and parsed as "Eigenvariables". We use this approach to get an expansion proof of the original problem, instead of the skolemized problem. Since each strong quantifier is replaced by exactly one Skolem term, the condition for the set of substitutions in Definition 10 is satisfied.

The collection of terms used for the weak quantifiers is a bit more involved due to variable renaming. The quantified variables in the input formula are renamed during the clausal normal form transformation. This means that the sets of variables occurring in the original problem and in the clauses are disjoint. The substitutions used in the proof are given with respect to the clauses' variables, but we are interested in building expansion trees of the input formulas. We need therefore to find a way to map the variables in the clauses to the variables in the input formulas.

The solution found was to implement in GAPT the definitional clausal form transformation, trying to remain as faithful as possible to the one leanCoP uses, but without the variable renaming. After applying our transformation to the input formulas, we try to match the clauses obtained to the clauses from the proof object. The first-order matching algorithm returns a substitution if a match is found. Such substitution maps strongly quantified variables to "Eigenvariables" (the result of parsing Skolem terms), and weakly quantified variables to their renamed versions used in the clauses. By composing this substitution with the ones obtained from the bindings in the proof, we are able to correctly identify the terms used for each quantified variable in the input formulas.

[6] `http://pages.cs.miami.edu/~tptp/CASC/24/Systems.tgz`
[7] `http://pages.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP`

4 Results

We were able to import as expansion trees all the 142 proof objects provided to us by the veriT team, and all but one under one minute. The expansion sequents generated have been used as input for the cut-introduction algorithm [6] and some of their features (e.g. high number of instances) have motivated improvements to the algorithm. As for leanCoP, our database consists of 3043 proofs of problems from the TPTP library [12]. Of those, we can successfully import 1224 as expansion sequents. Some errors still occur while parsing and matching (e.g. our generated clauses do not have the same literal ordering as the clauses in the proof file), but we are working to increase the success rate.

Getting proofs from various theorem provers in the shape of expansion sequents allows us to do a number of interesting things. First of all, one can visualize the end-sequent and the instances used of each quantified formula. This is much more comfortable and easier to grasp than a raw text file. It is also possible to check whether the instances used lead indeed to a proof of the end-sequent. This is reduced to checking if the deep sequent of the expansion sequent is a tautology (which can be done, as this sequent is propositional) and if the dependency relation is acyclic. In case the expansion sequent is a proof, we can build an LK proof from it, using the dependency relation to decide the order in which quantifiers are introduced [8]. Finally, one can attempt proof compression and discovery of lemmas using the cut-introduction algorithm [6].

All of these functionalities are implemented in GAPT. The system comes with an interactive command line where commands for loading proofs, opening `prooftool`, introducing cuts, eliminating cuts, building an LK proof from an expansion sequent, among others, can be issued. Some examples of proofs imported and their visualizations can be found at `https://www.logic.at/staff/giselle/examples.pdf`.

5 Related Work

Other projects and tools also address the issues of proof visualization and checking. For proofs in the TPTP language in particular, there is IDV [13], which provides an interactive interface for manipulating the DAG representing a derivation. This tool focuses solely on visualization of proofs in the TPTP format. Our work aims on a more general framework, of which visualization is only a small part. We are also capable to import different proof objects, not only those in the TPTP language.

As for proof checking, [7] proposes a check of leanCoP proofs in HOL Light while [1] shows how to check SAT and SMT proofs using Coq. The former paper involved re-implementing leanCoP's kernel in HOL Light, which differs a lot from our approach of simply parsing the outputs of theorem provers. In the latter, proofs produces by SAT/SMT theorem provers are certified by Coq. We must clarify that, given the information needed to produce expansion proofs, it is not fair to claim we are checking proof objects, but we merely have a sanity check that the instances used by the theorem prover actually lead to a proof of the proposed theorem. Such compromise makes sense if we want a framework general enough to deal with different proof objects, without asking any change on the side of theorem provers.

Finally, it is worth mentioning ProofCert [9], a research project with the aim of developing a theoretical framework for proof representation. In order not to make such compromise, and actually check each step of each proof for various different proof objects, a solid foundation of proof specification needs to be developed. While this does not happen, this work shows how it is still possible to combine existing proof objects into one representation.

6 Conclusion

We have shown how SMT and Connection proofs can be both imported as expansion sequents. The information needed from the proof objects is just the end-sequent being proven and a set of instances

used for the quantified formulas. For both cases presented we relied on a first-order matching algorithm, but this requirement can be lifted if all substitutions are provided directly in the proof object.

The representation using expansion sequents serves various purposes. It provides an easy proof visualization, a simple checking procedure, LK proof construction and introduction of cuts.

This is an ongoing work, and we hope to have many developments in the near future. In particular, the difficulties in importing leanCoP proofs remain to be resolved. This procedure also offers a lot of room for optimization. Once we have a big enough set of parsed leanCoP proofs, we will add those to the benchmark used in the cut-introduction algorithm. As for veriT proofs, we plan to test bigger examples, as the ones provided are only a small subset from the SMT-LIB.

Another future goal is importing other formats from other provers and comparing the different proofs for the same input problem. We also aim on integrating a check for whether the obtained expansion sequent is an expansion proof in the import function.

References

[1] Michael Armand, Germain Faure, Benjamin Grgoire, Chantal Keller, Laurent Thry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In: *CPP*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[2] Matthias Baaz & Alexander Leitsch (2000): *Cut-elimination and Redundancy-elimination by Resolution*. *Journal of Symbolic Computation* 29(2), pp. 149–176, doi:10.1006/jsco.1999.0359.

[3] Cvetan Dunchev, Alexander Leitsch, Tomer Libal, Martin Riener, Mikheil Rukhaia, Daniel Weller & Bruno Woltzenlogel Paleo (2013): *PROOFTOOL: a GUI for the GAPT Framework*. In: *10th UITP, EPTCS* 118, pp. 1–14, doi:10.4204/EPTCS.118.1.

[4] Cvetan Dunchev, Alexander Leitsch, Mikheil Rukhaia & Daniel Weller (2013): *CERES for First-Order Schemata*. *CoRR* abs/1303.4257, doi:10.1007/978-3-662-46906-4_8.

[5] Gerhard Gentzen (1935): *Untersuchungen über das logische Schließen I*. *Mathematische Zeitschrift* 39(1), pp. 176–210, doi:10.1007/BF01201353.

[6] Stefan Hetzl, Alexander Leitsch, Giselle Reis, Janos Tapolczai & Daniel Weller (2014): *Introducing Quantified Cuts in Logic with Equality*. In: *7th IJCAR, Lecture Notes in Computer Science* 8562, Springer, pp. 240–254, doi:10.1007/978-3-319-08587-6_17.

[7] Cezary Kaliszyk, Josef Urban & Jiři Vyskočil (2015): *Certified Connection Tableaux Proofs for HOL Light and TPTP*. CPP '15, ACM, New York, NY, USA, pp. 59–66, doi:10.1145/2676724.2693176.

[8] Dale Miller (1987): *A compact representation of proofs*. *Studia Logica* 46(4), pp. 347–370, doi:10.1007/BF00370646.

[9] Dale Miller (2011): *ProofCert: Broad Spectrum Proof Certificates*. ERC Advanced Grant 2012-2016.

[10] Jens Otten (2010): *Restricting backtracking in connection calculi*. *AI Commun.* 23(2-3), pp. 159–182, doi:10.3233/AIC-2010-0464.

[11] J. A. Robinson (1965): *A Machine-Oriented Logic Based on the Resolution Principle*. *J. ACM* 12(1), pp. 23–41, doi:10.1145/321250.321253.

[12] G. Sutcliffe (2009): *The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0*. *Journal of Automated Reasoning* 43(4), pp. 337–362, doi:10.1007/s10817-009-9143-8.

[13] Steven Trac, Yury Puzis & Geoff Sutcliffe (2007): *An Interactive Derivation Viewer*. *Electronic Notes in Theoretical Computer Science* 174(2), pp. 109 – 123, doi:10.1016/j.entcs.2006.09.025. Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006).

A framework for proof certificates in finite state exploration

Quentin Heath and Dale Miller
Inria Saclay–Île-de-France LIX, École polytechnique

Model checkers use automated state exploration in order to prove various properties such as reachability, non-reachability, and bisimulation over state transition systems. While model checkers have proved valuable for locating errors in computer models and specifications, they can also be used to prove properties that might be consumed by other computational logic systems, such as theorem provers. In such a situation, a prover must be able to trust that the model checker is correct. Instead of attempting to prove the correctness of a model checker, we ask that it outputs its "proof evidence" as a formally defined document—a proof certificate—and that this document is checked by a trusted proof checker. We describe a framework for defining and checking proof certificates for a range of model checking problems. The core of this framework is a (focused) proof system that is augmented with premises that involve "clerk and expert" predicates. This framework is designed so that soundness can be guaranteed independently of any concerns for the correctness of the clerk and expert specifications. To illustrate the flexibility of this framework, we define and formally check proof certificates for reachability and non-reachability in graphs, as well as bisimulation and non-bisimulation for labeled transition systems. Finally, we describe briefly a reference checker that we have implemented for this framework.

1 Introduction

Model checkers are one way in which logic is implemented. While one of the strengths of model checkers is to aid in the discovery of counterexamples and errors in specifications [6], they can also be used to prove theorems. Furthermore, such theorems might be of interest to other computational logic systems such as more general theorem provers. One then encounters the problem of whether or not such a theorem prover is willing to trust that model checker or at least a particular theorem it proves. Formally verifying a model checker might be both extremely hard to do and undesirable especially if that checker is being revised and improved. A more plausible option might be to have a model checker output its "proof evidence" as a document (a *certificate*). If that proof certificate can be formally checked by a trusted checker, one might then be willing to use the theorem in a theorem prover.

Of course, model checkers are asked to solve many kinds of problems so their proof evidence might take many different forms, ranging from decision procedures to paths in graphs, bisimulations, traces, and winning strategies. If we need to have trusted checkers for all these different kinds of proof evidence, then maybe we have not really improved the situation of trust. Here, we contribute to the *foundational proof certificate* (FPC) effort [13] by providing a framework for defining the semantics of a range of proof evidence that naturally arises in model checking. Such a formal semantic model for proof evidence allows anyone to build a proof checker of *any* formally defined evidence. Furthermore, it is possible to have an implementation of the entire framework of FPC so that this one system could check a wide range of proof evidence.

While this paper has a number of parallels with FPCs for first-order logic in [5], that work was limited to first-order logic *without* fixed points and, as a result, that work was not directly applicable to topics of model checking and inductive and co-inductive theorem proving.

C.Kaliszyk and A. Paskevich (Eds.): 4th Workshop
on Proof eXchange for Theorem Proving (PxTP'15).
EPTCS 186, 2015, pp. 11–26, doi:10.4204/EPTCS.186.4

2 Proof theory for fixed points and certificates

Having proof certificates that are foundational here means that we need to find proof theoretic descriptions of model checking. We shall now describe a few recent developments in proof theory that we bring together in this paper. Of course, the topic of model checking is mature and varied. In order to lay down a convincing and direct proof theory for model checking, we eschew many of its more advanced topics—e.g., predicate abstraction and partial order reduction—for later consideration.

2.1 Fixed points as defined predicates

One of the earliest applications of sequent calculus to computational logic was to provide an execution model for logic programming [15]. That analysis, however, supported only the "open world assumption" of logic programming: negation-as-finite-failure was not touched by that work. Schroeder-Heister [19] and Girard [10] showed how sequent calculus could be extended with inference rules for fixed points (or *defined* predicates), thereby embracing important aspects of the *closed world assumption* and negation-as-finite-failure. The key additions to sequent calculus were rules for unfolding fixed point expressions as well as dealing with equality over the Herbrand universe. A series of papers [8, 12, 16] added induction and co-induction to the sequent calculi for intuitionistic and classical logics. Those papers have been used to design the Bedwyr model checker [4, 20] and the Abella interactive theorem prover [3].

Fixed point expressions will be written as $\mu B\bar{t}$ or $\nu B\bar{t}$, where B is an expression representing a higher-order abstraction, and $\bar{t}$ is a list of terms. The unfolding of the fixed point expression $\mu B\bar{t}$ is written as $B(\mu B)\bar{t}$. It is important to understand that we shall treat both μ (least fixed point operator) and ν (greatest fixed point operator) as logical connectives since they will have introduction rules: they are also de Morgan duals of each other.

2.2 Fixed points in linear logic

Surprisingly, it is linear logic and not intuitionistic or classical logics per se that is most relevant to our exposition on model checking. The logic MALL (*multiplicative additive linear logic*) is an elegant, small logic that is, in and of itself, not appropriate for formalizing mathematics and computer science since it is not capable of modeling unbounded behaviors (for example, it is decidable). While Girard extended MALL with the "exponentials" (! and ?) [9], Baelde [2] extended it by adding the least (μ) and greatest (ν) fixed points operators as logical connectives. The resulting logic, called μMALL, forms the proof theoretic foundation of this paper.

To make the use of linear logic easier to swallow for those more familiar with model checking, we adopt the following shallow changes to its presentation. First, we use a two-sided sequent calculus instead of the one-sided calculus used for μMALL. While this change will double the size of our proof system, it will make inference rules look more familiar. Second, we replace the linear logic connectives with familiar connectives (although with annotations). In particular, we replace $\otimes$, &, $\oplus$ and their units $\mathbf{1}$, $\top$ and $\mathbf{0}$ with $\wedge^+$, $\wedge^-$, $\vee$, t^+, t^- and f^+, respectively. (Truth functionally, the two versions of these operators are equivalent: their differences only influence the structure of focused proofs.) We also replace the negatively biased false $\bot$ with f^-, and instead of the multiplicative disjunction $A \parr B$, we use the implication $A^\bot \supset B$: the de Morgan dual of $A \supset B$ is $A \wedge^+ B^\bot$. Negation is written as $\cdot \supset f^-$.

In addition, we consider μ as positive and ν as negative; this arbitrary choice has been shown to give a convenient natural interpretation to the structure of focused proofs [2]. We therefore have the negative connectives f^-, $\supset$, t^-, $\wedge^-$, $\forall$, $\neq$ and ν, and the positive connectives t^+, $\wedge^+$, f^+, $\vee$, $\exists$, $=$ and μ.

2.3 Focused proof systems

In order to have the kind of control we need to support a definable notion of proof certificate, we make use of a *focused proof system*. Such sequent calculus proof systems are built from alternating phases which allow us to define flexible proof building protocols that can be used to drive proof search. During the *asynchronous* phase of proof building, simple (invertible) computations build a proof and during the *synchronous* phase, information needed for the construction of a proof (such as which branch of a disjunction to prove) must be found.

Focusing requires polarizing all formulas as being either negative or positive. A formula is negative or positive according to its top-level connective, and it is *purely negative* (resp. *purely positive*) when its connectives are positive if, and only if, they occur under an odd (resp. even) number of implications. Notice that the de Morgan dual of a positive (resp. purely positive) formula is a negative (resp. purely negative) formula. We call a formula *bipolar* when it is made of purely negative (resp. positive) subformulas occurring under an even (resp. odd) number of implications in a purely negative context.

Focusing also relies on the sequents having additional storage zones on each side of the turnstile, where formulas can be stored and left untouched by logical inference rules. For instance, the usual one-sided focused presentation of μMALL [2] has one of these zones, similarly to the focused proof system for linear logic given by Andreoli [1]. A two-sided subsystem of μMALLF, called μF, makes use of two storage zones, noted $\mathscr{N}$ and $\mathscr{P}$, which are lists of, respectively, negative and positive formulas. (Appendix B contains an example of a μF proof.) Between the arrows and the turnstile, are the contexts Γ and Δ: these are lists of formulas in (unfocused) $\Uparrow$-sequents, and sets of up to one formula in (focused) $\Downarrow$-sequents. The sequents of the μF system are therefore:

$\mathscr{N} \Uparrow \Gamma \vdash \Delta \Uparrow \mathscr{P}$	unfocused, similar to the μMALLF sequent $\vdash \mathscr{N}^{\perp}, \mathscr{P} \Uparrow \Gamma^{\perp}, \Delta$
$\Downarrow A \vdash$	left-focused, similar to $\vdash \Downarrow A^{\perp}$
$\vdash A \Downarrow$	right-focused, similar to $\vdash \Downarrow A$

2.4 Foundational proof certificates

If we think of the implementers of computational logic systems (*e.g.*, model checking systems) as our clients, our job in this project is to formally check our client's proof evidence for formal correctness. Our approach is to have this evidence translated into a sequent calculus proof. Of course, we would not dream of asking our clients to supply a sequent calculus proof in the first place: such proofs are often huge, too messy, and too esoteric. Instead, we want to take from our clients objects with which they are familiar (*e.g.*, paths, simulations, *etc.*) and find flexible and high-level means to have our framework extract information from those objects in order to trace out a complete formal sequent calculus proof.

To this intent, Figures 1 and 2 present μF^{a}, which is a version of μF augmented with a term Ξ (encoding an actual certificate) as well as with *clerk* and *expert* predicates (examples of which we provide soon). This augmentation has two components. First, every sequent (either $\Uparrow$ or $\Downarrow$) is given an extra argument we write as Ξ. Thus, sequents now display as

$$\Xi : \mathscr{N} \Uparrow \Gamma \vdash \Delta \Uparrow \mathscr{P}, \qquad \Xi : \Downarrow A \vdash, \qquad \text{and} \qquad \Xi : \vdash A \Downarrow .$$

Second, every inference rule is given an additional premise. In all cases, this premise is an atomic formula with either a clerk or expert predicate as its head symbol: if the conclusion of the inference rule is a $\Downarrow$-sequent, then the premise atom uses an expert predicate (noted $\star_e(\dots)$ for the rule $\star$); otherwise, the conclusion is an $\Uparrow$-sequent and the atom uses a clerk predicate (noted $\star_c(\dots)$).

In the case of the clerk rules, the premise atom relates the Ξ value of the concluding sequent with the corresponding value of Ξ for all premises: *e.g.*,

$$\frac{\Xi_1 : \mathscr{N} \Uparrow A_1, \Gamma \vdash \Delta \Uparrow \quad \Xi_2 : \mathscr{N} \Uparrow A_2, \Gamma \vdash \Delta \Uparrow \quad \vee_c(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0 : \mathscr{N} \Uparrow A_1 \vee A_2, \Gamma \vdash \Delta \Uparrow} \vee_L$$

In this way, the certificate Ξ, intended to aid in the proof of the concluding sequent, can be transformed into two certificates that are used to prove the two premise sequents. We refer to the predicates used in the asynchronous phase as clerks since these predicates do not need, in general, to examine the actual information in the proof certificate (except for the induction and co-induction rules, there is no consumption of information during the asynchronous phase). Instead, the clerks are responsible for keeping track of how a proof is unfolding: for example, Ξ_1 might be a copy of Ξ_0 but with the fact that checking has moved to the left premise instead of the right premise.

Experts are responsible for extracting information from a certificate. For example, μF^a contains the inference rule

$$\frac{\Xi_1 : \vdash C t \Downarrow \quad \exists_e(\Xi_0, \Xi_1, t)}{\Xi_0 : \vdash \exists x. C x \Downarrow} \exists_R$$

Notice here that the exists-expert $\exists_e(\cdot,\cdot,\cdot)$ not only computes the continuation certificate Ξ_1 but also a term t to be used to witness the existential variable.

The exact nature of both certificate terms Ξ and of the clerk and expert predicates is not important to guarantee soundness of this system. That is, no matter how certificates, clerks, and experts are specified, if there is a proof in μF^a then there is a proof in μF of the same sequent, which can be obtained by deleting from the proof in μF^a all references to Ξ, including the additional premises. Notice also that experts are not required to act particularly expertly: it is entirely possible for the $\exists_e(\Xi_0, \Xi_1, t)$ premise to functionally determine one t from Ξ_0, or to relate all terms t to Ξ. In the latter case, the actual value of t used in a successful μF^a proof is determined from other aspects of the proof checking process (typically implemented using unification).

3 A proof system underlying model checking

FPCs were first proposed in [5, 13] in the context of first-order logic and were used successfully to define and check proof evidence in the form of resolution refutations, Herbrand instances (expansion trees), natural deduction (λ-terms), Frege proofs, *etc.* We shall now adapt this approach to formally define the semantics of a range of proof evidence that can arise in simple but real model checking problems.

We shall later illustrate just how such a formal semantics can be provided for the following four kinds of proof evidence. These particular examples have been selected for their universality: numerous problems in model checking are related to them.

1. The fact that two nodes are related by the transitive closure of a graph's adjacency relation can be witnessed by an *explicit path* through the graph.
2. The fact that two nodes are *not* related by transitivity can be witnessed by pointing out that the *reachable set* of one does not contain the other.
3. Given an LTS (labeled transition system), the fact that two nodes are similar/bisimilar can be witnessed by a set of pairs called *simulation/bisimulation*.
4. If two nodes in an LTS are *not* bisimilar, then there is a *Hennessy-Milner logic (HML) formula* that is satisfied by one but not by the other.

ASYNCHRONOUS CONNECTIVE INTRODUCTIONS

$$\frac{\Xi_1\theta : \mathscr{N}\theta \Uparrow \Gamma\theta \vdash \Delta\theta \Uparrow \quad =^s_c(\Xi_0,\Xi_1)}{\Xi_0 : \mathscr{N} \Uparrow s = t, \Gamma \vdash \Delta \Uparrow} =^s_L\dagger \qquad \frac{\Xi_1\theta : \mathscr{N}\theta \Uparrow \vdash \Uparrow \quad \neq^f_c(\Xi_0,\Xi_1)}{\Xi_0 : \mathscr{N} \Uparrow \vdash s \neq t \Uparrow} \neq^f_R\dagger$$

$$\frac{\Xi_1 : \mathscr{N} \Uparrow \Gamma \vdash \Delta \Uparrow \quad t^+_c(\Xi_0,\Xi_1)}{\Xi_0 : \mathscr{N} \Uparrow t^+, \Gamma \vdash \Delta \Uparrow} t^+_L \qquad \frac{\Xi_1 : \mathscr{N} \Uparrow \vdash \Uparrow \quad f^-_c(\Xi_0,\Xi_1)}{\Xi_0 : \mathscr{N} \Uparrow \vdash f^- \Uparrow} f^-_R$$

$$\frac{\Xi_1 : \mathscr{N} \Uparrow A_1, A_2, \Gamma \vdash \Delta \Uparrow \quad \wedge^+{}_c(\Xi_0,\Xi_1)}{\Xi_0 : \mathscr{N} \Uparrow A_1 \wedge^+ A_2, \Gamma \vdash \Delta \Uparrow} \wedge^+{}_L \qquad \frac{\Xi_1 : \mathscr{N} \Uparrow A_1 \vdash A_2 \Uparrow \quad \supset_c(\Xi_0,\Xi_1)}{\Xi_0 : \mathscr{N} \Uparrow \vdash A_1 \supset A_2 \Uparrow} \supset_R$$

$$\frac{=^f_c(\Xi_0)}{\Xi_0 : \mathscr{N} \Uparrow s = t, \Gamma \vdash \Delta \Uparrow} =^f_L\ddagger \qquad \frac{\neq^s_c(\Xi_0)}{\Xi_0 : \mathscr{N} \Uparrow \vdash s \neq t \Uparrow} \neq^s_R\ddagger$$

$$\frac{f^+_c(\Xi_0)}{\Xi_0 : \mathscr{N} \Uparrow f^+, \Gamma \vdash \Delta \Uparrow} f^+_L \qquad \frac{t^-_c(\Xi_0)}{\Xi_0 : \mathscr{N} \Uparrow \vdash t^- \Uparrow} t^-_R$$

$$\frac{\Xi_1 : \mathscr{N} \Uparrow A_1, \Gamma \vdash \Delta \Uparrow \quad \Xi_2 : \mathscr{N} \Uparrow A_2, \Gamma \vdash \Delta \Uparrow \quad \vee_c(\Xi_0,\Xi_1,\Xi_2)}{\Xi_0 : \mathscr{N} \Uparrow A_1 \vee A_2, \Gamma \vdash \Delta \Uparrow} \vee_L$$

$$\frac{\Xi_1 : \mathscr{N} \Uparrow \vdash A_1 \Uparrow \quad \Xi_2 : \mathscr{N} \Uparrow \vdash A_2 \Uparrow \quad \wedge^-{}_c(\Xi_0,\Xi_1,\Xi_2)}{\Xi_0 : \mathscr{N} \Uparrow \vdash A_1 \wedge^- A_2 \Uparrow} \wedge^-{}_R$$

$$\frac{\Xi_1\, y : \mathscr{N} \Uparrow Cy, \Gamma \vdash \Delta \Uparrow \quad \exists_c(\Xi_0,\Xi_1)}{\Xi_0 : \mathscr{N} \Uparrow \exists x.Cx, \Gamma \vdash \Delta \Uparrow} \exists_L \qquad \frac{\Xi_1\, y : \mathscr{N} \Uparrow \vdash Cy \Uparrow \quad \forall_c(\Xi_0,\Xi_1)}{\Xi_0 : \mathscr{N} \Uparrow \vdash \forall x.Cx \Uparrow} \forall_R$$

SYNCHRONOUS CONNECTIVE INTRODUCTIONS

$$\frac{\neq^f_e(\Xi_0)}{\Xi_0 : \Downarrow t \neq t \vdash} \neq^f_L \qquad \frac{=^s_e(\Xi_0)}{\Xi_0 : \vdash t = t \Downarrow} =^s_R \qquad \frac{f^-_e(\Xi_0)}{\Xi_0 : \Downarrow f^- \vdash} f^-_L \qquad \frac{t^+_e(\Xi_0)}{\Xi_0 : \vdash t^+ \Downarrow} t^+_R$$

$$\frac{\Xi_1 : \vdash A_1 \Downarrow \quad \Xi_2 : \Downarrow A_2 \vdash \quad \supset_e(\Xi_0,\Xi_1,\Xi_2)}{\Xi_0 : \Downarrow A_1 \supset A_2 \vdash} \supset_L \qquad \frac{\Xi_1 : \vdash A_1 \Downarrow \quad \Xi_2 : \vdash A_2 \Downarrow \quad \wedge^+{}_e(\Xi_0,\Xi_1,\Xi_2)}{\Xi_0 : \vdash A_1 \wedge^+ A_2 \Downarrow} \wedge^+{}_R$$

$$\frac{\Xi_1 : \Downarrow A_i \vdash \quad \wedge^-{}_e(\Xi_0,\Xi_1,i)}{\Xi_0 : \Downarrow A_1 \wedge^- A_2 \vdash} \wedge^-{}_L \qquad \frac{\Xi_1 : \vdash A_i \Downarrow \quad \vee_e(\Xi_0,\Xi_1,i)}{\Xi_0 : \vdash A_1 \vee A_2 \Downarrow} \vee_R$$

$$\frac{\Xi_1 : \Downarrow Ct \vdash \quad \forall_e(\Xi_0,\Xi_1,t)}{\Xi_0 : \Downarrow \forall x.Cx \vdash} \forall_L \qquad \frac{\Xi_1 : \vdash Ct \Downarrow \quad \exists_e(\Xi_0,\Xi_1,t)}{\Xi_0 : \vdash \exists x.Cx \Downarrow} \exists_R$$

STRUCTURAL RULES

$$\frac{\Xi_1 : N \Uparrow \Gamma \vdash \Delta \Uparrow \quad \mathrm{store}_L(\Xi_0,\Xi_1)}{\Xi_0 : \Uparrow N, \Gamma \vdash \Delta \Uparrow} \mathrm{S}_L \qquad \frac{\Xi_1 : \Uparrow \vdash \Uparrow P \quad \mathrm{store}_R(\Xi_0,\Xi_1)}{\Xi_0 : \Uparrow \vdash P \Uparrow} \mathrm{S}_R$$

$$\frac{\Xi_1 : \Downarrow N \vdash \quad \mathrm{decide}_L(\Xi_0,\Xi_1)}{\Xi_0 : N \Uparrow \vdash \Uparrow} \mathrm{D}_L \qquad \frac{\Xi_1 : \vdash P \Downarrow \quad \mathrm{decide}_R(\Xi_0,\Xi_1)}{\Xi_0 : \Uparrow \vdash \Uparrow P} \mathrm{D}_R$$

$$\frac{\Xi_1 : \Uparrow P \vdash \Uparrow \quad \mathrm{release}_L(\Xi_0,\Xi_1)}{\Xi_0 : \Downarrow P \vdash} \mathrm{R}_L \qquad \frac{\Xi_1 : \Uparrow \vdash N \Uparrow \quad \mathrm{release}_R(\Xi_0,\Xi_1)}{\Xi_0 : \vdash N \Downarrow} \mathrm{R}_R$$

Figure 1: The μF^a_0 proof system. (This proof system is best viewed using color).
y stands for a fresh eigenvariable, s and t for terms, N for a negative formula, P for a positive formula, and C for the abstraction of a formula over a variable.
The † proviso requires that θ is the *mgu* of s and t, and the ‡ proviso requires that s and t are not unifiable.

FIXED-POINT RULES

$$\frac{\Xi_1\,\bar{y}\colon \Uparrow B S \bar{y} \vdash S\bar{y} \Uparrow \quad \Xi_2 \colon \mathscr{N} \Uparrow S\bar{t}, \Gamma \vdash \Delta \Uparrow \quad \text{ind}(\Xi_0, \Xi_1, \Xi_2, S)}{\Xi_0 \colon \mathscr{N} \Uparrow \mu B \bar{t}, \Gamma \vdash \Delta \Uparrow}\ \mu$$

$$\frac{\Xi_1 \colon \mathscr{N} \Uparrow \vdash S\bar{t} \Uparrow \quad \Xi_2\,\bar{y}\colon \Uparrow S\bar{y} \vdash B S \bar{y} \Uparrow \quad \text{co-ind}(\Xi_0, \Xi_1, \Xi_2, S)}{\Xi_0 \colon \mathscr{N} \Uparrow \vdash \nu B \bar{t} \Uparrow}\ \nu$$

$$\frac{\Xi_1 \colon \mathscr{N} \Uparrow B(\mu B)\bar{t}, \Gamma \vdash \Delta \Uparrow \quad \mu\text{-unfold}_L(\Xi_0, \Xi_1)}{\Xi_0 \colon \mathscr{N} \Uparrow \mu B \bar{t}, \Gamma \vdash \Delta \Uparrow}\ \mu_L \qquad \frac{\Xi_1 \colon \mathscr{N} \Uparrow \vdash B(\nu B)\bar{t} \Uparrow \quad \nu\text{-unfold}_R(\Xi_0, \Xi_1)}{\Xi_0 \colon \mathscr{N} \Uparrow \vdash \nu B \bar{t} \Uparrow}\ \nu_R$$

$$\frac{\Xi_1 \colon \Downarrow B(\nu B)\bar{t} \vdash \quad \nu\text{-unfold}_L(\Xi_0, \Xi_1)}{\Xi_0 \colon \Downarrow \nu B \bar{t} \vdash}\ \nu_L \qquad \frac{\Xi_1 \colon \vdash B(\mu B)\bar{t} \Downarrow \quad \mu\text{-unfold}_R(\Xi_0, \Xi_1)}{\Xi_0 \colon \vdash \mu B \bar{t} \Downarrow}\ \mu_R$$

Figure 2: The μF^a proof system results from adding these rules to μF^a_0.
$\bar{y}$ stands for an list of fresh eigenvariables, $\bar{t}$ for an list of terms, and B for the abstraction of a formula over a predicate and a variable list.

3.1 Core proof system

Figures 1 and 2 contain the rules for the augmented focused proof systems μF^a_0 and μF^a. One could obtain the non-augmented systems μF_0 and μF by ignoring the certificates (annotated Ξ variables) and the clerk and expert premises; the resulting rules would be no more than (slightly restricted) two-sided versions of the μMALLF rules. The various clerk and expert predicates are named and displayed in their corresponding inference rules. Notice that those inference rules that involve the use of eigenvariables ($\exists_L$, $\forall_R$, μ and ν) require the associated clerk predicates to return abstractions over certificates: in this way, premise certificates can be applied to the eigenvariables.

A key element of our proof theoretic treatment of model checking via μF^a is the fact that focused sequents contain only one formula. This fact entails that μF^a can only be complete with respect to μMALLF on a fragment where derivations satisfy this constraint. In particular, the $\mathscr{N}$ and $\mathscr{P}$ zones must never contain more than one formula, and never both at the same time. This can be ensured at least for the μF^a_0 subsystem by the following restriction on formulas.

Definition 1 (switchable formula, switchable occurrence)**.** *A μF^a formula is* switchable *if*

- *whenever a subformula $C \wedge^+ D$ occurs negatively (under an odd number of implications), either C or D is purely positive;*
- *whenever a subformula $C \supset D$ occurs positively (under an even number of implications), either C is purely positive or D is purely negative.*

An occurrence of a formula B is switchable *if it appears on the right-hand side (resp. left-hand side) and B (resp. $B \supset f^-$) is switchable.*

Notice that both a purely positive formula and its de Morgan dual are switchable. The follow theorem is proved by a simple induction on the structure of μF^a_0 proofs.

Theorem 1 (switchability)**.** *Let Π be a μF^a_0 derivation of either $\Uparrow A \vdash \Uparrow$ or $\Uparrow \vdash A \Uparrow$, where the occurrence of A is switchable. Every sequent in Π that is the conclusion of a rule that switches phases (either a decide or a release rule) contains exactly one occurrence of a formula and that occurrence is switchable.*

Theorem 1 states that an invariant of the μF_0^a proof system (for switchable theorems) is that the number of non-purely asynchronous formulas (*i.e.* non-purely positive from $\mathscr{N}$ and Γ, and non-purely negative from $\mathscr{P}$ and Δ) is one or less. Keeping sequents mostly asynchronous allows the asynchronous phase to deal with most of the context: that way the synchronous phase is left with a single, meaningful formula. (The structure of focused proofs based on switchable formulas is similar to the structure of *simple games* in the game-theoretic analysis of focused proofs in [7, Section 4].) While the restriction to switchable formulas provides a match to the model checking problems we develop here, that restriction is not needed for using clerks and experts (the examples in [5] involve non-switchable formulas).

3.2 Encoding of recursively defined predicates

In order to exploit the properties of μF_0^a in model checking problems, we need them to extend to μF^a by adding fixed-point rules. As those rules make use of the higher-order variables S (an invariant which is either a pre-fixed point or a post-fixed point) and B (the body of a predicate definition), they cannot be used freely without violating Theorem 1. We propose the following constraints on μF^a proofs of switchable formulas so as to have exactly one formula per sequent when phases are switched:

- "arithmetic" restriction: S and B are purely positive (resp. negative);
- "model checking" restriction: S is purely negative (resp. positive), and the context does not trigger synchronous rules ($\mathscr{N}$ is empty, Γ is purely positive and Δ is purely negative).

The former restriction would allow to extend the scope of the framework by handling simple theorems involving inductive definitions (*e.g.* about natural numbers), but is not treated here. The latter restriction better suits our needs (since an asynchronous context fits the spirit of model checking) and is respected by all our examples.

Example 2. *Horn clauses (in the sense of Prolog) can be encoded as purely positive fixed point expressions. For example, here is the Horn clause logic program (using the λProlog syntax, the* `sigma Y\` *construction encodes the quantifier ∃Y) for specifying the graph in Figure 3 and its transitive closure:*

```
step a b.   step b c.    step c b.
path X Y :- step X Y.    path X Z :- sigma Y\ step X Y, path Y Z.
```

We can translate the `step` *relation into the binary predicate $\cdot \longrightarrow \cdot$ defined by*

$$\mu\left(\lambda A\lambda x\lambda y.\,((x=a)\wedge^{+}(y=b))\vee((x=b)\wedge^{+}(y=c))\vee((x=c)\wedge^{+}(y=b))\right)$$

which only uses positive connectives. Likewise, `path` *can be encoded as path:*

$$\mu\left(\lambda A\lambda x\lambda z.\,x\longrightarrow z\vee(\exists y.\,x\longrightarrow y\wedge^{+}A\,y\,z)\right)$$

In general, it is sensible to view any purely positive least fixed point expression as a predicate specified by Horn clauses. (For example, SOS rules for CCS are easily seen as Horn clauses.)

Example 3. *Let the ternary predicate $\cdot \dot{\longrightarrow} \cdot$ describe a labeled transition system. It can be defined as a purely positive fixed point expression of the form*

$$\mu\left(\lambda A\lambda p\lambda a\lambda q.\,\textstyle\bigvee_i((p=u_i)\wedge^{+}(a=v_i)\wedge^{+}(q=w_i))\right)$$

and the simulation and bisimulation relations can be defined as the following greatest fixed point expressions (note: the second contains both $\wedge^{-}$ and $\wedge^{+}$). Both of these formulas are switchable.

$$\nu\left(\lambda S\lambda p\lambda q.\,\forall a\forall p'.\,p\xrightarrow{a}p'\supset\exists q'.\,q\xrightarrow{a}q'\wedge^{+}S\,p'\,q'\right) \tag{sim}$$

$$\begin{aligned}\nu\big(\lambda B\lambda p\lambda q.\,&(\forall a\forall p'.\,p\xrightarrow{a}p'\supset\exists q'.\,q\xrightarrow{a}q'\wedge^{+}B\,p'\,q')\\ &\wedge^{-}(\forall a\forall q'.\,q\xrightarrow{a}q'\supset\exists p'.\,p\xrightarrow{a}p'\wedge^{+}B\,q'\,p')\big)\end{aligned} \tag{bisim}$$

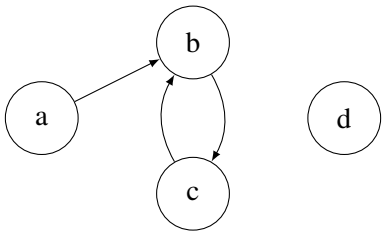


Figure 3: (Un)reachability problem

3.3 Common proof certificates

The presentation of an FPC now involves the following three steps.

1. Describe how unpolarized formulas should be polarized.
2. Describe the structure of certificates Ξ. This can be done, for example, by describing the signature of constructors for certificates.
3. Define the clerk and expert predicates.

To ease steps 2 and 3, we define the following certificate constructors (shown together with their types), which describe generic focused proof behaviors. The associated clauses can be included into any subsequent clerks and experts definitions.

The **stop**$:\texttt{cert}$ certificate authorizes no search; it is to be used as a continuation certificate for other certificate constructors.

The **sync**$:\texttt{cert->cert}$ certificate constructor authorizes μF^a to conduct an unbounded synchronous search for a proof before handing the search over to a continuation certificate. It has no clerks and its experts run an exhaustive non-deterministic search for $\vee$ and $\exists$. The experts for the right rules are:

$$
\begin{array}{ll}
=^s_e(\mathsf{sync}(\Xi)). & \vee_e(\mathsf{sync}(\Xi),\mathsf{sync}(\Xi),1). \\
\wedge^+{}_e(\mathsf{sync}(\Xi),\mathsf{sync}(\Xi),\mathsf{sync}(\Xi)). & \vee_e(\mathsf{sync}(\Xi),\mathsf{sync}(\Xi),2). \\
\forall T.\, \exists_e(\mathsf{sync}(\Xi),\mathsf{sync}(\Xi),T). & \mu\text{-unfold}_R(\mathsf{sync}(\Xi),\mathsf{sync}(\Xi)). \\
\text{release}_R(\mathsf{sync}(\Xi),\Xi). &
\end{array}
$$

The **async**$:\texttt{cert->cert}$ certificate constructor is the dual of sync; it handles an asynchronous phase and has no experts apart from the decide rules. The clerks for the left rules are:

$$
\begin{array}{ll}
=^s_c(\mathsf{async}(\Xi),\mathsf{async}(\Xi)). & \vee_c(\mathsf{async}(\Xi),\mathsf{async}(\Xi),\mathsf{async}(\Xi)). \\
\wedge^+{}_c(\mathsf{async}(\Xi),\mathsf{async}(\Xi)). & \\
\exists_c(\mathsf{async}(\Xi),\lambda x.\,\mathsf{async}(\Xi)). & \mu\text{-unfold}_L(\mathsf{async}(\Xi),\mathsf{async}(\Xi)). \\
\text{store}_L(\mathsf{async}(\Xi),\mathsf{async}(\Xi)). & \text{decide}_L(\mathsf{async}(\Xi),\Xi).
\end{array}
$$

bipole$_n:\texttt{cert}$ is actually short-hand for a chain of n $\mathsf{async}(\mathsf{sync}(\cdot))$ before a final stop. It is used for bounded-depth search when simple search strategies would otherwise not terminate. We also write **bipole**$:\texttt{cert}$ for $\mathsf{bipole}_1 = \mathsf{async}(\mathsf{sync}(\mathsf{stop}))$.

The **decproc**$:\texttt{cert}$ constructor is short-hand for bipole_∞, the unbounded version of bipole_n. It is a general purpose decision procedure used for automated and unguided proving. Its rules are similar to those from sync and async, and can be obtained via the equivalence $\mathsf{decproc} = \mathsf{async}(\mathsf{sync}(\mathsf{decproc}))$.

The two constructors **inv** and **co-inv** : `(i->i->bool)->cert->cert` each take an explicit predicate S as parameter. It is expected to be proved to be an invariant with the help of bipole.

$$\forall S.\,\mathsf{ind}(\mathsf{inv}(S,\Xi),\lambda\bar{x}.\,\mathsf{bipole},\Xi,S) \qquad \forall S.\,\mathsf{co\text{-}ind}(\mathsf{co\text{-}inv}(S,\Xi),\Xi,\lambda\bar{x}.\,\mathsf{bipole},S)$$

We now turn our attention to describing how to formally define the four kinds of proof evidence mentioned earlier in Section 3. Some of the constructors defined above will be used in those definitions.

4 Examples: certificates for graphs

We use the notations from Theorem 2 to define $\cdot \longrightarrow \cdot$ and *path*.

4.1 Lists as reachability certificates

The natural choice for a certificate of the proof of $\vdash \mathit{path}(x,y)$ is an explicit path, *i.e.* a list of nodes starting right after x and ending right before y. In fact, this list L can be used directly as the proof certificate. Aside from the initial store_R, no clerks are invoked in the process of checking this particular FPC. The following clauses defining the experts only use the provided information to instantiate the logical variables of the proof.

$$\forall X\forall L.\,\exists_e(X :: L, L, X). \qquad \forall L.\,\wedge^+{}_e(L,\mathsf{sync}(\mathsf{stop}),L). \qquad \forall L.\,\mathsf{decide}_R(L,L).$$
$$\forall X\forall L.\,\vee_e(X :: L, X :: L, 2). \qquad \forall L.\,\vee_e(\mathit{nil},\mathsf{sync}(\mathsf{stop}),1). \qquad \forall L.\,\mu\text{-}\mathsf{unfold}_R(L,L).$$

In this setting, the $\mathsf{sync}(\mathsf{stop})$ certificate will terminate quickly since it is only searching through the term that defines $\cdot \longrightarrow \cdot$.

Example 4. *In Figure 3, (c) is reachable from (a), as witnessed by certificates like $[b]$, $[b;c;b]$, etc.*

4.2 Invariants as non-reachability certificates

The non-reachability problem comes in two forms: if there are no loops in the graph, then a simple check of the set of nodes reachable from the first node provides a simple decision procedure; if there are loops, then induction is needed.

In the first case, the decision procedure can directly be translated as an FPC for proving $\vdash \neg\mathit{path}(x,y)$.

Example 5. *In Figure 3, (a) is not reachable from (d), as witnessed by $\mathsf{async}(\mathsf{stop})$.*

On the other hand, if the underlying graph has loops, then the rules of Figure 1 only do not allow proof search to terminate. As the body B of the *path* expression (*i.e.*, the displayed formula without μ) is purely positive, a bipole can prove that a chosen purely negative predicate S containing no fixed point expressions is an induction invariant (bipole: $\Uparrow B\,S\,x\,y \vdash S\,x\,y \Uparrow$), which means that we can use the certificate constructor $\mathsf{inv}(S,\cdot)$. Then we use another bipole as continuation certificate for this constructor to check that the invariant is adequate for the refutation of $\mathit{path}(t,u)$ (bipole: $\Uparrow S\,t\,u \vdash \cdot \Uparrow$).

Here, the invariant can be chosen so as to represent the fact of *not* belonging to the set $\mathscr{T} \times \{u\}$, where $\mathscr{T}$ is the reachable set of $\{t\}$.

Example 6. *In Figure 3, (d) is not reachable from (b), as witnessed by $\mathsf{inv}(S,\mathsf{bipole})$, where the invariant S (built from the set $\{b,c\} \times \{d\}$) is*

$$\lambda x\lambda y.\,((x = b \wedge^+ y = d) \vee (x = c \wedge^+ y = d)) \supset f^-$$

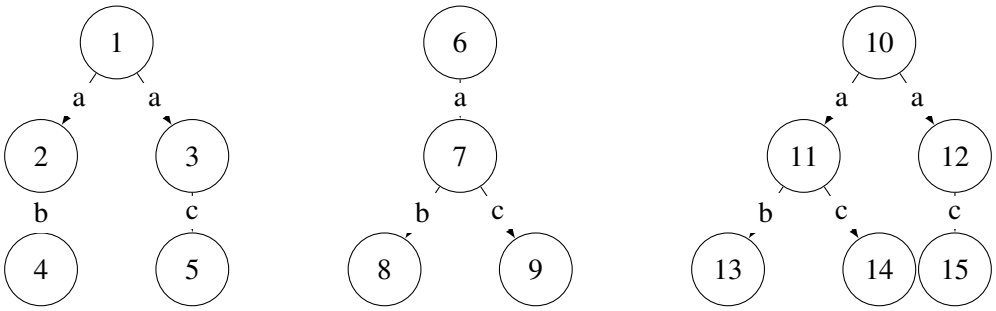


Figure 4: Non-(bi)similar noetherian labeled transition systems

5 Examples: certificates for labeled transition systems

Bisimilarity and similarity are important relationships in the domain of process calculus and model checking. To illustrate how these can be captured as FPCs, we first restrict our attention to the existence of a simulation between finite labeled transition systems; bisimilarity is then addressed by expanding on this presentation. We define $\cdot \dot{\longrightarrow} \cdot$ (for the LTS), *sim* (for simulated-by) and *bisim* (for bisimulated-by) as seen in Theorem 3.

5.1 Invariants as simulation certificates

We shall consider two cases: one where the underlying transition system is noetherian and one where it is not. An LTS is said to be *noetherian* if there is no infinite sequence of transitions $p_1 \xrightarrow{a_1} p_2 \xrightarrow{a_2} \cdots$ (in the setting of finite LTSs, this is equivalent to the absence of loops).

In the noetherian case, there is a decision procedure to determine whether or not one process is simulated by another: one simply attempts to incrementally check simulation at every point. This systematic search can be described using the clerks and experts of the decproc certificate, which allows a proof to be built from any number of bipoles (one for each unfolding of the simulation predicate, which formula is itself bipolar).

Example 7. *In Figure 4, the process* (1) *is simulated by the process* (6)*, as witnessed by the certificate decproc.*

In the more general (possibly non-noetherian) setting, we need to recall the formal definition of the simulation relation as a set. A binary relation S is a simulation if whenever $\langle p,q\rangle \in S$ and whenever $p \xrightarrow{a} p'$ holds, then there exists a q' such that $q \xrightarrow{a} q'$ holds and $\langle p',q'\rangle \in S$. We say that process p is simulated by process q if there is a simulation S such that $\langle p,q\rangle \in S$.

Let S be a finite set of pairs and let $\hat{S}$ be the purely positive expression $\lambda x\lambda y.\ \bigvee_{\langle p,q\rangle\in S}(x = p \wedge^+ y = q)$. As the body B of the *sim* expression is a bipolar formula, a bipole can prove the closure condition for (finite) simulations (bipole: $\Uparrow \hat{S}\,x\,y \vdash B\,\hat{S}\,x\,y \Uparrow$), so we can use the certificate constructor co-inv$(\hat{S},\cdot)$. Once again, we use another bipole as continuation certificate to complete the proof that p is simulated by q (bipole: $\Uparrow \cdot \vdash \hat{S}\,p\,q \Uparrow$).

Example 8. *According to Figure 5, the set* $\{(21,23),(22,24)\}$ *is a simulation and, therefore, the process* (21) *is simulated by the process* (23)*. This corresponds to the following certificate*

$$\mathit{co\text{-}inv}\left(\lambda x\lambda y.\,(x = 21 \wedge^+ y = 23) \vee (x = 22 \wedge^+ y = 24), \mathit{bipole}\right).$$

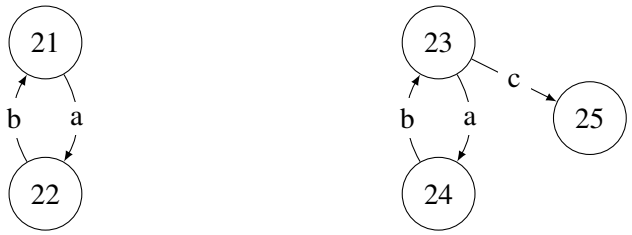


Figure 5: Non-noetherian labeled transition systems

Providing an entire invariant as part of a proof certificate or restricting to the case when an invariant is finite certainly limits what kinds of simulation relationships can be proved. In general, invariants will not be finite and, even when they are, they are large. It is for reasons such as this that there has been a great deal of work on bisimulation-up-to [17, 18]: generally, it is possible to discover and check a closure property of a much smaller relationship and then via various meta-theoretic properties, ensure that such closure properties entail the existence of a proper (bi)simulation.

5.2 Assertions as non-simulation certificates

Hennessy and Milner [11] provided a characterization of bisimulation in terms of an assertion language over modal operators $[a]$ and $\langle a\rangle$. The characterization states that two processes are bisimilar if and only if they satisfy the same assertion formulas. Thus, if p and q are not bisimilar, there is some assertion formula A which is true for p and not for q. Formally, we write $p \models A$ and $q \not\models A$.

It is possible to use such assertion formulas directly as proof certificates in the simpler and related problem of the absence of simulation, *i.e.* for theorems of the form $\vdash \neg sim(p,q)$. In that case, the assertion language needs only the diamond modality $\langle\cdot\rangle$ as well as the conjunction. More formally, let *Act* be a set of actions. The restricted set of assertions over *Act* is given by the recurrence $A := \bigwedge_{i\in I}\langle a_i\rangle A_i$, where I is a finite set and $a_i \in Act$; that is, we have a strict alternation of (indexed) conjunctions and the diamond modality. The statement $p \models \bigwedge_{i\in I}\langle a_i\rangle A_i$ means that, for every $i \in I$, there exists a q_i such that $p \xrightarrow{a_i} q_i$ and $q_i \models A_i$. We shall choose to write *true* for empty conjunctions and we can drop $\bigwedge_{i\in I}$ when I is a singleton. Thus, $\langle a\rangle \mathit{true}$ stands for $\bigwedge_{i\in\{\bullet\}}\langle a\rangle \bigwedge_{j\in\{\}}\langle b_{i,j}\rangle A_{i,j}$.

Some of the clerks and experts needed for this interpretation of an assertion as a certificate are listed below; the rest of the definition can be taken from the async constructor.

$$\forall(a_i)_i\forall(A_i)_i\forall j.\,\mathrm{decide}_L(\textstyle\bigwedge_i\langle a_i\rangle A_i, \langle a_j\rangle A_j). \qquad \forall a\forall A.\,\forall_e(\langle a\rangle A, A, a).$$

$$\forall A.\,\supset_e(A, \mathsf{sync}(\mathsf{stop}), A). \qquad \forall T\forall A.\,\forall_e(A, A, T).$$

$$\forall a\forall A.\,\nu\text{-unfold}_L(\langle a\rangle A, \langle a\rangle A). \qquad \forall A.\,\mathrm{release}_L(A, A).$$

Example 9. *In Figure 4, the process* (6) *is not simulated by the process* (1)*: if* Ξ *is the assertion formula* $\langle a\rangle(\langle b\rangle \mathit{true} \wedge \langle c\rangle \mathit{true})$*, then* $6 \models \Xi$ *but* $1 \not\models \Xi$*.*

5.3 Assertions as non-bisimilarity certificates

It is possible to extend the FPC described in Section 5.2 to account for the absence of bisimulation in addition to the absence of simulation. As bisimilarity is finer than similarity, this will require a richer class of assertion formulas. The fact that it is a symmetric relation suggests that assertions should contain negations.

We could use full Hennessy-Milner logic (*i.e.* any arbitrary mix of $\langle\cdot\rangle$, $[\cdot]$, $\vee$ and $\wedge$ or, equivalently, $\langle\cdot\rangle$, $\wedge$ and $\neg$), but instead we choose the smaller but equivalent set of assertions defined by the following recurrence.

$$A := \bigwedge_{i\in I} B_i$$
$$B := \langle a_i\rangle A_i \mid \neg(\langle a_i\rangle A_i)$$

It can be shown that this set characterizes the same relation as full Hennessy-Milner logic. The statement $p \models \bigwedge_{i\in I} B_i$ means that, for every $i \in I$, $p \models B_i$; the statement $p \models \langle a\rangle A$ means that there exists a q such that $p \xrightarrow{a} q$ and $q \models A$; and the statement $p \models \neg(\langle a\rangle A)$ means that $p \not\models \langle a\rangle A$.

Very little more is needed to extend the FPC to handle this certificate. We need to make sure that, in addition to certificates with a top-level $\langle\cdot\rangle$, decide$_L$ and ν-unfold$_L$ allow (and propagate) certificates with a top-level $\neg\langle\cdot\rangle$. We also need an expert to consume $\neg$, and an expert to handle the additional $\wedge^-$ connective (see the definition of bisimilarity from Theorem 3). If we give these two roles to the same new expert, namely $\wedge^-{}_e$, the link between reflexivity and negations in the assertions appears clearly.

The resulting set of clerks and experts for theorems of the form $\vdash \neg bisim(p,q)$ is the following.

$$\forall A.\,\text{store}_L(A,A). \qquad \forall (B_i)_i \forall j.\,\text{decide}_L(\textstyle\bigwedge_i B_i, B_j).$$
$$\forall B.\,\nu\text{-unfold}_L(B,B). \qquad \forall a \forall A.\,\wedge^-{}_e(\ \langle a\rangle A, \langle a\rangle A, 1).$$
$$\forall a \forall A.\,\forall_e(\langle a\rangle A, A, a). \qquad \forall a \forall A.\,\wedge^-{}_e(\neg\langle a\rangle A, \langle a\rangle A, 2).$$
$$\forall T \forall A.\,\forall_e(A,A,T). \qquad \forall A.\,\supset_e(A, \mathsf{sync}(\mathsf{stop}), A).$$
$$\forall A.\,\text{release}_L(A,A).$$
$$\forall A.\,\exists_c(A, \lambda x.A). \qquad \forall A.\,\wedge^+{}_c(A,A).$$
$$\forall A.\,\mu\text{-unfold}_L(A,A). \qquad \forall A.\,\vee_c(A,A,A).$$
$$\forall A.\,=^s_c(A,A).$$

This FPC extension is conservative, in that it can still check a certificate for non-simulation.

Example 10. *In Figure 4, the processes* (6) *and* (10) *are similar but not bisimilar: if* Ξ *is the generalized assertion formula* $\langle a\rangle\neg\langle b\rangle$true*, then* $10 \models \Xi$ *but* $6 \not\models \Xi$.

6 A reference proof checker

The framework for foundational proof certificates described in [13, 5] was based on proof theory without fixed point definitions. In that setting, a standard logic programming language (in that case, λProlog [14]) was an ideal prototyping language for implementing and testing FPCs. The FPCs described in this paper are not so easily implemented in standard logic programming languages since the unification of eigenvariables must be done alongside the usual unification of "logic variables" that makes proof reconstruction possible. The implementation of λProlog, for example, considers eigenvariables as constants during unification.

We have built a prototype proof checker for testing the FPCs described in this paper using the Bedwyr extension to logic programming [21, 4]. That system, originally designed to tackle various kinds of

model checking problems, provides the necessary unification for logic and eigenvariables along with backtracking search and support for λ-terms, λ-conversion and higher-order pattern unification.

One could have imagined implementing the non-augmented proof system μF_0 directly and in a sense, this is already done by Bedwyr itself. For example, if $(\mu B\bar{t})$ is a purely positive fixed point encoding a Prolog predicate, when the system is given the sequent $\vdash (\mu B\bar{t}) \Downarrow$, it would emulate the Prolog search. Similarly, if it is given the sequent $\Uparrow (\mu B\bar{t}) \vdash \Uparrow$, it would emulate a finite failing proof search. But, as anyone familiar with Prolog-style depth-first search knows, such proof search is limited in its effectiveness. For example, if one is attempting to prove that there is or is not a path between two points, a cycle in the underlying graph can make the search non-terminating. Bedwyr handles this with a loop-detection mechanism that can be embedded in the rules from Figure 2, making it a partial implementation of μF.

However, our goal with the μF proof system is not to use it by itself, but together with clerks and experts, as the engine (as a "kernel") for checking already existing proof evidence. Since the logic of the existing Bedwyr system has no native support for proof objects, we implemented μF^a as an "object logic", without using some native features such as loop-detection. The Bedwyr specification files that we use (available directly at `http://slimmer.gforge.inria.fr/bedwyr/pcmc/`, or from the authors' homepages) are rather direct translations of the inference rules in Figures 1 and 2 as well as of the various FPCs listed in the previous few sections. It has thus been easy for us to experiment and test FPCs.

While we have found the Bedwyr system to be useful for prototyping a proof checker, our proposal for FPC is not tied to any one particular implementation. Instead, the framework is defined using inference rules (such as found in Figures 1 and 2). Any system that can implement the logical principles required by such inference rules can be used as a proof checking FPC kernel.

7 Conclusion

We have taken the basic structure of *foundational proof certificates* that had been developed elsewhere for first-order logic and described how it could be imposed on a logic based on fixed points. The resulting logic is much richer (think of the difference between first-order logic and first-order arithmetic) and additional logic principles need to be accounted for in the description of proof certificates.

In the areas of model checking that we have discussed, proof evidence is often taken to be, say, a path through a graph, a set of pairs of nodes (satisfying certain closure conditions), or a Hennessy-Milner logic assertion formula. We have illustrated how each of these familiar objects can be easily transformed into hints to guide a proof checker though the construction of a detailed and complete sequent calculus proof. The architecture of focused proof systems and the clerk and expert predicates allow this conceptual gap (between familiar proof evidence and sequent calculus proofs) to be bridged in a flexible and natural fashion.

We have also provided a novel look at the proof theory foundations of model checking systems by basing our entire project on the μMALL variant of linear logic and on the notion of *switchable formulas*. This latter notion seems to provide an interesting demarcation between the logically simpler notion of model checking and the more general notion of (inductive and co-inductive) deduction.

Acknowledgment. We thank the reviewers for their detailed and useful comments on an earlier draft of this paper. This work has been funded by the ERC Advanced Grant ProofCert.

References

[1] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic*. *J. of Logic and Computation* 2(3), pp. 297–347, doi:10.1093/logcom/2.3.297.

[2] David Baelde (2012): *Least and greatest fixed points in linear logic*. *ACM Trans. on Computational Logic* 13(1), doi:10.1145/2071368.2071370.

[3] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A System for Reasoning about Relational Specifications*. *Journal of Formalized Reasoning* 7(2), doi:10.6092/issn.1972-5787/4650.

[4] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur & Alwen Tiu (2007): *The Bedwyr system for model checking over syntactic expressions*. In F. Pfenning, editor: *21th Conf. on Automated Deduction (CADE)*, *LNAI* 4603, Springer, New York, pp. 391–397, doi:10.1007/978-3-540-73595-3_28.

[5] Zakaria Chihani, Dale Miller & Fabien Renaud (2013): *Foundational proof certificates in first-order logic*. In Maria Paola Bonacina, editor: *CADE 24: Conference on Automated Deduction 2013*, *LNAI* 7898, pp. 162–177, doi:10.1007/978-3-642-38574-2_11.

[6] Edmund M. Clarke, Orna Grumberg & Doron A. Peled (1999): *Model Checking*. The MIT Press, Cambridge, Massachusetts.

[7] Olivier Delande, Dale Miller & Alexis Saurin (2010): *Proof and refutation in MALL as a game*. *Annals of Pure and Applied Logic* 161(5), pp. 654–672, doi:10.1016/j.apal.2009.07.017.

[8] Andrew Gacek, Dale Miller & Gopalan Nadathur (2011): *Nominal abstraction*. *Information and Computation* 209(1), pp. 48–73, doi:10.1016/j.ic.2010.09.004.

[9] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[10] Jean-Yves Girard (1992): *A Fixpoint Theorem in Linear Logic*. An email posting to the mailing list linear@cs.stanford.edu.

[11] Matthew Hennessy & Robin Milner (1985): *Algebraic Laws for Nondeterminism and Concurrency*. *JACM* 32(1), pp. 137–161, doi:10.1145/2455.2460.

[12] Raymond McDowell & Dale Miller (2000): *Cut-elimination for a logic with definitions and induction*. *Theoretical Computer Science* 232, pp. 91–119, doi:10.1016/S0304-3975(99)00171-1.

[13] Dale Miller (2011): *A proposal for broad spectrum proof certificates*. In J.-P. Jouannaud & Z. Shao, editors: *CPP: First International Conference on Certified Programs and Proofs*, *LNCS* 7086, pp. 54–69, doi:10.1007/978-3-642-25379-9_6.

[14] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*. Cambridge University Press, doi:10.1017/CBO9781139021326.

[15] Dale Miller, Gopalan Nadathur, Frank Pfenning & Andre Scedrov (1991): *Uniform Proofs as a Foundation for Logic Programming*. *Annals of Pure and Applied Logic* 51, pp. 125–157, doi:10.1016/0168-0072(91)90068-W.

[16] Dale Miller & Alwen Tiu (2005): *A proof theory for generic judgments*. *ACM Trans. on Computational Logic* 6(4), pp. 749–783, doi:10.1145/1094622.1094628.

[17] Robin Milner (1989): *Communication and Concurrency*. Prentice-Hall International.

[18] Damien Pous & Davide Sangiorgi (2011): *Enhancements of the bisimulation proof method*. In Davide Sangiorgi & Jan Rutten, editors: *Advanced Topics in Bisimulation and Coinduction*, Cambridge University Press, pp. 233–289, doi:10.1017/CBO9780511792588.007.

[19] Peter Schroeder-Heister (1993): *Rules of Definitional Reflection*. In M. Vardi, editor: *8th Symp. on Logic in Computer Science*, IEEE Computer Society Press, IEEE, pp. 222–232, doi:10.1109/LICS.1993.287585.

[20] Alwen Tiu, Gopalan Nadathur & Dale Miller (2005): *Mixing Finite Success and Finite Failure in an Automated Prover*. In: *Empirically Successful Automated Reasoning in Higher-Order Logics (ESHOL'05)*, pp. 79–98.

[21] (2015): *The Bedwyr model checker*. Available at `http://slimmer.gforge.inria.fr/bedwyr/`.

A On the augmented focusing proof system μF^a

It should be noted that, although the system presents two left rules for the connective $=$, one with the clerk $=^s_c$ for success and one with the clerk $=^f_c$ for failure, the implementation is usually expected to have one single unification facility that, given an equation, will or will not succeed, and which is tied to one single clerk. If the unification fails, the rule succeeds immediately without generating a premise certificate Ξ_1, and the constraint on the conclusion certificate Ξ_0 is actually the same as for success. Hence the clerk $=^f_c$ can be defined as an existential closure of $=^s_c$. Likewise, $\neq^s_c$ can be defined in terms of $\neq^f_c$.

$$=^f_c(\cdot) \equiv (\exists \Xi_1 . =^s_c(\cdot, \Xi_1)) \qquad \neq^s_c(\cdot) \equiv (\exists \Xi_1 . \neq^f_c(\cdot, \Xi_1))$$

It is also possible to remove the truth and falsity connectives, as we expect to have the equivalences

$$t^+ \equiv (a = a) \qquad f^+ \equiv (a = b)$$
$$f^- \equiv (a \neq a) \qquad t^- \equiv (a \neq b)$$

if a and b are distinct constants, hence the following:

$$t^+_e \equiv =^s_e \qquad t^+_c \equiv =^s_c \qquad f^+_c \equiv =^f_c$$
$$f^-_e \equiv \neq^f_e \qquad f^-_c \equiv \neq^f_c \qquad t^-_c \equiv \neq^s_c$$

Last, it is customary to leave clerks and experts out of rules with no premises (*i.e.* t^+_e, $=^s_e$, f^-_e, $\neq^f_e$, f^+_c, $=^f_c$, t^-_c and $\neq^f_c$). This has the same effect as setting them to be always true.

The system presented in Figures 1 and 2 does not have these simplifications, but the Bedwyr-based implementation does.

B A simple example of a μF proof

The following proof can be seen as the justification that $\{1,3\} \subseteq \{1,2,3\}$. In particular, encode these two sets as the predicates (i.e., abstractions over formula):

$$\lambda x[x = 1 \vee x = 3] \quad \text{and} \quad \lambda x[x = 1 \vee x = 2 \vee x = 3].$$

The sequent calculus proof of inclusion can then be written as the following focused proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash 1 = 1 \Downarrow}}{\vdash 1 = 1 \vee 1 = 2 \vee 1 = 3 \Downarrow}}{\Uparrow \vdash \Uparrow 1 = 1 \vee 1 = 2 \vee 1 = 3}}{\Uparrow \vdash 1 = 1 \vee 1 = 2 \vee 1 = 3 \Uparrow}}{\Uparrow x = 1 \vdash x = 1 \vee x = 2 \vee x = 3 \Uparrow} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash 3 = 3 \Downarrow}}{\vdash 3 = 1 \vee 3 = 2 \vee 3 = 3 \Downarrow}}{\Uparrow \vdash \Uparrow 3 = 1 \vee 3 = 2 \vee 3 = 3}}{\Uparrow \vdash 3 = 1 \vee 3 = 2 \vee 3 = 3 \Uparrow}}{\Uparrow x = 3 \vdash x = 1 \vee x = 2 \vee x = 3 \Uparrow}}{\Uparrow x = 1 \vee x = 3 \vdash x = 1 \vee x = 2 \vee x = 3 \Uparrow}$$
$$\dfrac{\dfrac{\Uparrow \vdash [x = 1 \vee x = 3] \supset [x = 1 \vee x = 2 \vee x = 3] \Uparrow}{\Uparrow \vdash \forall x.[x = 1 \vee x = 3] \supset [x = 1 \vee x = 2 \vee x = 3] \Uparrow}$$

Systematic Verification of the Modal Logic Cube in Isabelle/HOL*

Christoph Benzmüller Maximilian Claus
Dep. of Mathematics and Computer Science, Freie Universität Berlin, Germany
c.benzmueller|m.claus@fu-berlin.de

Nik Sultana
Computer Lab, Cambridge University, UK
nik.sultana@cl.cam.ac.uk

We present an automated verification of the well-known modal logic cube in Isabelle/HOL, in which we prove the inclusion relations between the cube's logics using automated reasoning tools. Prior work addresses this problem but without restriction to the modal logic cube, and using encodings in first-order logic in combination with first-order automated theorem provers. In contrast, our solution is more elegant, transparent and effective. It employs an embedding of quantified modal logic in classical higher-order logic. Automated reasoning tools, such as Sledgehammer with LEO-II, Satallax and CVC4, Metis and Nitpick, are employed to achieve full automation. Though successful, the experiments also motivate some technical improvements in the Isabelle/HOL tool.

1 Introduction

We present an approach to meta-reasoning about modal logics, and apply it to verify the relative strengths of logics in the well-known *modal logic cube*, which is illustrated in Figure 1. In particular, proofs are given for the equivalences of different axiomatizations and the inclusion relations shown in the cube.

Our solution makes extensive use of the fact that all modal logics found in the cube are sound and complete because they arise from base modal logic K by adding Sahlqvist axioms. This is in contrast to prior work by Rabe et al. [16], who address the more general problem of determining the relation between two arbitrary modal logics characterized by their sets of inference rules. In their article the authors apply first-order logic encodings in combination with first-order automated theorem provers to prove an inclusion relation employing a number of different decision strategies. For the subproblem of only comparing logics within the cube (and therefore taking advantage of normality as additional knowledge) our solution improves on the elegance and simplicity of the problem encodings, as well as with automation performance. One motivation of this paper is to demonstrate the advantage of a pragmatically more expressive logic environment (here classical higher-order logic) in comparison to a less expressive language such as first-order logic or decidable fragments thereof.

We exploit an embedding of quantified multimodal logic (QML) in classical higher-order logic (HOL) [7], in which we carry out the automated verification of the aforementioned inclusion relations. These include the logics **K**, **D**, **M** (also known as **T**), **S4**, and **S5**. We analyze inclusion and equivalence relations for modal logics that can be defined from normal modal logic **K** by adding (combinations of) the axioms M, B, D, 4, and 5. In our problem encodings we exploit the well-known correspondences between these

*This work has been supported by the German Research Foundation DFG under grants BE2501/9-2 & BE2501/11-1.

C.Kaliszyk and A. Paskevich (Eds.): 4th Workshop
on Proof eXchange for Theorem Proving (PxTP'15).
EPTCS 186, 2015, pp. 27–41, doi:10.4204/EPTCS.186.5

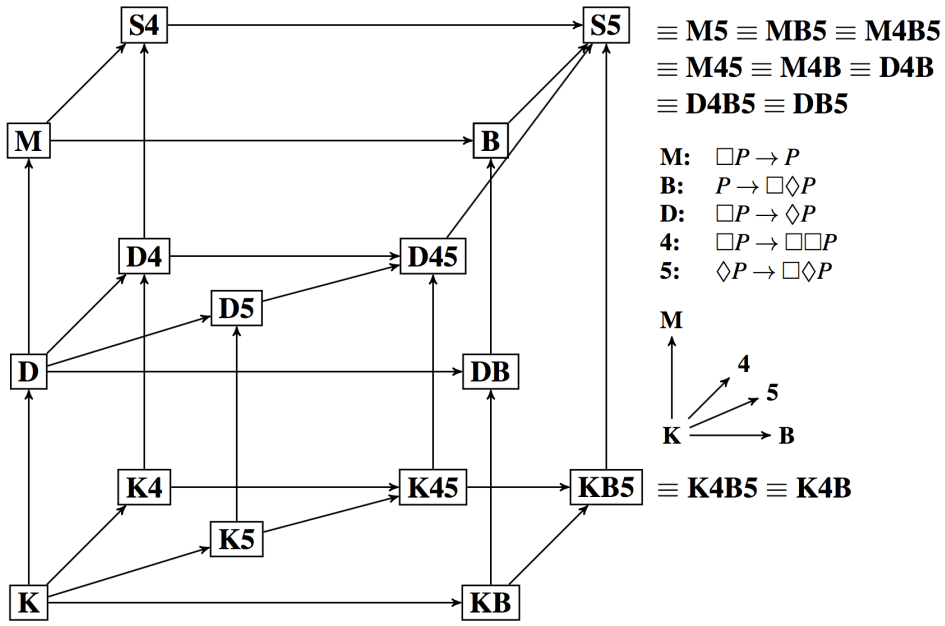


Figure 1: The modal logic cube: reasoning in modal logics is commonly done with respect to a certain set of basic axioms; different choices of basic axioms give rise to different modal logics. These modal logics can be arranged as vertices in a cube, such that the edges between them denote inclusion relations.

axioms and semantic properties of accessibility relations (i.e. Kripke models). These correspondences can themselves be elegantly formalized and effectively automated in our approach. Formalization of the modal axioms M, B, D, 4, and 5 requires quantification over propositional variables. This explains why an embedding of *quantified* modal logic in HOL is needed here, and not simply an embedding of propositional modal logic in HOL.

Our previous work (see the non-refereed, invited paper [3]) has already demonstrated the feasibility of the approach. However, instead of the development done there in pure TPTP THF [8], we here work with Isabelle/HOL [14] as the base environment, and fruitfully exploit various reasoning tools that are provided with it. This includes the Sledgehammer-based [15] interfaces from Isabelle/HOL to the external higher-order theorem provers LEO-II [9] and Satallax [1], as well as Isabelle/HOL's own reasoner Metis [11]. Moreover, the higher-order model finding capabilities of Nitpick [10] are heavily used in order to formulate and prove subsequent inclusion theorems in Isabelle/HOL. We also encountered some problems with interacting with the proof reconstruction available for LEO-II and Satallax in Isabelle/HOL.

This paper is a verified document in the sense that it has been automatically generated from Isabelle/HOL source code with the help of Isabelle's *build* tool (the entire source package is available from http://christoph-benzmueller.de/varia/pxtp2015.zip).

The paper is structured as follows: Section 2 presents an encoding of QML in HOL. This part reuses the theory provided by Benzmüller and Paulson [7], which has recently been further developed (to cover full higher-order QML) and applied for the verification of Gödel's ontological argument [5, 6]. Section 3 first establishes the well-known correspondence between properties of models and base axioms, and then investigates the equivalence of different axiomatizations. Subsequently, all inclusion relations as

depicted in the modal logic cube are shown to be proper. Finally, the minimal number of possible worlds that is required to obtain proper inclusions in each case is determined and verified. Section 4 presents a short evaluation and discussion of the conducted experiments, and Section 5 concludes the paper.

2 An Embedding of Quantified Multimodal Logics in HOL

In contrast to the monomodal case, in quantified multimodal logics both modalities $\Box$ and $\Diamond$ are parametrized, such that they refer to potentially different accessibility relations. We write $\Box^R$ and $\Diamond^R$ to refer to necessity and possibility wrt. a relation R. Furthermore, in terms of quantification, we only consider the constant-domain case: this means that all possible worlds share one common domain of discourse. More details on the embedding of QML in HOL are given earlier work [7, 6].

QML formulas are translated as HOL terms of type $i \Rightarrow bool$, where i is the type of possible worlds. This type is abbreviated as σ.

The classical connectives $\neg, \wedge, \rightarrow$, and $\forall$ (which quantifies over individuals and over sets of individuals) and $\exists$ (over individuals) are lifted to type σ. The lifted connectives are $\neg^m$, $\wedge^m$, $\vee^m$, $\rightarrow^m$, $\equiv^m$, $\forall$, and $\exists$ (the latter two are modeled as constant symbols). Other connectives can be introduced analogously. Moreover, the modal operators $\Box$ and $\Diamond$, parametric to R, are introduced. Note that in symbols like $\neg^m$, symbol m is simply part of the name, whereas in $\Box^R$ and $\Diamond^R$, symbol R is a parameter to the modality.

abbreviation *mnot* :: $\sigma \Rightarrow \sigma$ **where** $\neg^m\ \varphi \equiv (\lambda w.\ \neg\ \varphi\ w)$
abbreviation *mand* :: $\sigma \Rightarrow \sigma \Rightarrow \sigma$ **where** $\varphi \wedge^m \psi \equiv (\lambda w.\ \varphi\ w \wedge \psi\ w)$
abbreviation *mor* :: $\sigma \Rightarrow \sigma \Rightarrow \sigma$ **where** $\varphi \vee^m \psi \equiv (\lambda w.\ \varphi\ w \vee \psi\ w)$
abbreviation *mimplies* :: $\sigma \Rightarrow \sigma \Rightarrow \sigma$ **where** $\varphi \rightarrow^m \psi \equiv (\lambda w.\ \varphi\ w \longrightarrow \psi\ w)$
abbreviation *mequiv*:: $\sigma \Rightarrow \sigma \Rightarrow \sigma$ **where** $\varphi \equiv^m \psi \equiv (\lambda w.\ \varphi\ w \longleftrightarrow \psi\ w)$
abbreviation *mforall* :: $('a \Rightarrow \sigma) \Rightarrow \sigma$ **where** $\forall\ \Phi \equiv (\lambda w.\ \forall x.\ \Phi\ x\ w)$
abbreviation *mexists* :: $('a \Rightarrow \sigma) \Rightarrow \sigma$ **where** $\exists\ \Phi \equiv (\lambda w.\ \exists x.\ \Phi\ x\ w)$
abbreviation *mbox* :: $(i \Rightarrow i \Rightarrow bool) \Rightarrow \sigma \Rightarrow \sigma$ **where** $\Box^R\ \varphi \equiv (\lambda w.\ \forall v.\ (R\ w\ v) \longrightarrow \varphi\ v)$
abbreviation *mdia* :: $(i \Rightarrow i \Rightarrow bool) \Rightarrow \sigma \Rightarrow \sigma$ **where** $\Diamond^R\ \varphi \equiv (\lambda w.\ \exists v.\ R\ w\ v \wedge \varphi\ v)$

For grounding lifted formulas, the meta-predicate $[\cdot]$, read *valid*, is introduced.

abbreviation *valid* :: $\sigma \Rightarrow bool$ **where** $[p] \equiv \forall w.\ p\ w$

3 Reasoning about Modal Logics

3.1 Correspondence Results

Axioms of the modal cube correspond to constraints on the underlying accessibility relations. These constraints are as follows:

definition *refl* $\equiv \lambda R :: (i \Rightarrow i \Rightarrow bool).\ \forall S.\ R\ S\ S$ — reflexivity
definition *sym* $\equiv \lambda R :: (i \Rightarrow i \Rightarrow bool).\ \forall S\ T.\ (R\ S\ T \longrightarrow R\ T\ S)$ — symmetry
definition *ser* $\equiv \lambda R :: (i \Rightarrow i \Rightarrow bool).\ \forall S.\ \exists T.\ R\ S\ T$ — seriality
definition *trans* $\equiv \lambda R :: (i \Rightarrow i \Rightarrow bool).\ \forall S\ T\ U.\ (R\ S\ T \wedge R\ T\ U \longrightarrow R\ S\ U)$ — transitivity
definition *eucl* $\equiv \lambda R :: (i \Rightarrow i \Rightarrow bool).\ \forall S\ T\ U.\ (R\ S\ T \wedge R\ S\ U \longrightarrow R\ T\ U)$ — Euclidean

The corresponding axioms are defined next; note that they are parametric over accessibility relation R:

definition $M \equiv \lambda R$. *valid* $(\forall(\lambda P.\ (\Box^R\ P) \rightarrow^m P))$
definition $B \equiv \lambda R$. *valid* $(\forall(\lambda P.\ P \rightarrow^m \Box^R\Diamond^R\ P))$
definition $D \equiv \lambda R$. *valid* $(\forall(\lambda P.\ (\Box^R\ P) \rightarrow^m \Diamond^R\ P))$
definition $IV \equiv \lambda R$. *valid* $(\forall(\lambda P.\ (\Box^R\ P) \rightarrow^m \Box^R\Box^R\ P))$
definition $V \equiv \lambda R$. *valid* $(\forall(\lambda P.\ (\Diamond^R\ P) \rightarrow^m \Box^R\Diamond^R\ P))$

We will see below that *correspondence theorems* (between axioms and constraints on accessibility relations) can be elegantly expressed in HOL by exploiting the embedding used above. These correspondence theorems link a constraint to every axiom—for instance, *M* is linked to *refl*. Subsequently, in order to make statements about the relationship of two logics in the cube, it is sufficient to only look at the model constraints of their respective axiomatizations. Throughout the rest of this paper, all reasoning will be done on the model-theoretic side and then interpreted on the proof-theoretic side by the means of this correspondence.

3.1.1 Axiom M corresponds to Reflexivity

theorem *A1*: $(\forall R.\ (refl\ R) \longleftrightarrow (M\ R))$ **by** $(metis\ M\text{-}def\ refl\text{-}def)$

3.1.2 Axiom B corresponds to Symmetry

lemma *A2-a*: $(\forall R.\ (sym\ R) \longrightarrow (B\ R))$ **by** $(metis\ B\text{-}def\ sym\text{-}def)$
lemma *A2-b*: $(\forall R.\ (B\ R) \longrightarrow (sym\ R))$ **by** $(simp\ add{:}B\text{-}def\ sym\text{-}def, force)$
theorem *A2*: $(\forall R.\ (sym\ R) \longleftrightarrow (B\ R))$ **by** $(metis\ A2\text{-}a\ A2\text{-}b)$

3.1.3 Axiom D corresponds to Seriality

theorem *A3*: $(\forall R.\ (ser\ R) \longleftrightarrow (D\ R))$ **by** $(metis\ D\text{-}def\ ser\text{-}def)$

3.1.4 Axiom 4 corresponds to Transitivity

theorem *A4*: $(\forall R.\ (trans\ R) \longleftrightarrow (IV\ R))$ **by** $(metis\ IV\text{-}def\ trans\text{-}def)$

3.1.5 Axiom 5 corresponds to Euclideanness

lemma *A5-a*: $(\forall R.\ (eucl\ R) \longrightarrow (V\ R))$ **by** $(metis\ V\text{-}def\ eucl\text{-}def)$
lemma *A5-b*: $(\forall R.\ (V\ R) \longrightarrow (eucl\ R))$ **by** $(simp\ add{:}V\text{-}def\ eucl\text{-}def, force)$
theorem *A5*: $(\forall R.\ (eucl\ R) \longleftrightarrow (V\ R))$ **by** $(metis\ A5\text{-}a\ A5\text{-}b)$

3.2 Alternative Axiomatisations of Modal Logics

Often the same logic within the cube can be obtained through different axiomatizations. In this section we show how to prove different axiomatizations for logic **S5** resp. **KB5** to be equivalent. Using the correspondence theorems from the previous section, the equivalences can be elegantly formulated solely using the properties of accessibility relations. In Subsections 3.2.1 and 3.2.2 we also add the corresponding statements using the modal logic axioms; this could analogously be done also for the other theorems and lemmata presented in Sections 3.2 and 3.3.

The theorems below can be solved directly by Metis when it is provided the minimal set of necessary definitions. Sledgehammer (with the ATPs LEO-II and Satallax or with first-order provers) can also quickly solve these problems, in which case the manual selection of the required definitions is not necessary.

3.2.1 M5 $\Longleftrightarrow$ MB5

theorem *B1*: $\forall R.((refl\ R) \wedge (eucl\ R)) \longleftrightarrow ((refl\ R) \wedge (sym\ R) \wedge (eucl\ R))$
by (*metis eucl-def refl-def sym-def*)
theorem *B1-alt*: $\forall R.((M\ R) \wedge (V\ R)) \longleftrightarrow ((M\ R) \wedge (B\ R) \wedge (V\ R))$
by (*metis A1 A2 A5 B1*)

3.2.2 M5 $\Longleftrightarrow$ M4B5

theorem *B2*: $\forall R.((refl\ R) \wedge (eucl\ R)) \longleftrightarrow ((refl\ R) \wedge (trans\ R) \wedge (sym\ R) \wedge (eucl\ R))$
by (*metis eucl-def refl-def trans-def sym-def*)
theorem *B2-alt*: $\forall R.((M\ R) \wedge (V\ R)) \longleftrightarrow ((M\ R) \wedge (IV\ R) \wedge (B\ R) \wedge (V\ R))$
by (*metis A1 A4 A5 B1-alt B2*)

3.2.3 M5 $\Longleftrightarrow$ M45

theorem *B3*: $\forall R.((refl\ R) \wedge (eucl\ R)) \longleftrightarrow ((refl\ R) \wedge (trans\ R) \wedge (eucl\ R))$
by (*metis eucl-def refl-def trans-def*)

3.2.4 M5 $\Longleftrightarrow$ M4B

theorem *B4*: $\forall R.((refl\ R) \wedge (eucl\ R)) \longleftrightarrow ((refl\ R) \wedge (trans\ R) \wedge (sym\ R))$
by (*metis eucl-def refl-def sym-def trans-def*)

3.2.5 M5 $\Longleftrightarrow$ D4B

theorem *B5*: $\forall R.((refl\ R) \wedge (eucl\ R)) \longleftrightarrow ((ser\ R) \wedge (trans\ R) \wedge (sym\ R))$
by (*metis eucl-def refl-def ser-def sym-def trans-def*)

3.2.6 M5 $\Longleftrightarrow$ D4B5

theorem *B6*: $\forall R.((refl\ R) \wedge (eucl\ R)) \longleftrightarrow ((ser\ R) \wedge (trans\ R) \wedge (sym\ R) \wedge (eucl\ R))$
by (*metis eucl-def refl-def ser-def sym-def trans-def*)

3.2.7 M5 $\Longleftrightarrow$ DB5

theorem *B7*: $\forall R.((refl\ R) \wedge (eucl\ R)) \longleftrightarrow ((ser\ R) \wedge (sym\ R) \wedge (eucl\ R))$
by (*metis eucl-def refl-def ser-def sym-def*)

3.2.8 KB5 $\Longleftrightarrow$ K4B5

theorem *B8*: $\forall R.((sym\ R) \wedge (eucl\ R)) \longleftrightarrow ((trans\ R) \wedge (sym\ R) \wedge (eucl\ R))$
by (*metis eucl-def sym-def trans-def*)

3.2.9 KB5 $\Longleftrightarrow$ K4B

theorem *B9*: $\forall R.((sym\ R) \wedge (eucl\ R)) \longleftrightarrow ((trans\ R) \wedge (sym\ R))$
by (*metis eucl-def sym-def trans-def*)

3.3 Proper Inclusion Relations between Different Modal Logics

An edge within the cube denotes an inclusion between the connected logics. In the forward direction, these can be trivially shown valid through monotonicity of entailment and equivalence of the different

axiomatizations. For example, for the forward link from logic **K** to logic **B**, we need to show that every theorem of **K** is also a theorem of **B**; this simply means to disregard the additional axiom B. Below, the crucial backward directions are proved. Informally, it is shown that through moving further up in the cube (adding further axioms), theorems can be proved which were not provable before; this means that the inclusions are proper. We write $A > B$ to indicate that logic A can prove strictly more theorems than logic B.

It has to be noted that some logics are actually equivalent if the only models considered have few enough worlds; examples are given below. We introduce some useful abbreviations to formulate constraints on the number of worlds in a model.

abbreviation *one-world-model* :: $i \Rightarrow bool$ **where** $\#^1\ w1 \equiv \forall x.\ x = w1$
abbreviation *two-world-model* :: $i \Rightarrow i \Rightarrow bool$ **where** $\#^2\ w1\ w2 \equiv (\forall x.\ x = w1 \vee x = w2) \wedge w1 \neq w2$
abbreviation *three-world-model* :: $i \Rightarrow i \Rightarrow i \Rightarrow bool$ **where** $\#^3\ w1\ w2\ w3 \equiv (\forall x.\ x = w1 \vee x = w2 \vee x = w3) \wedge w1 \neq w2 \wedge w1 \neq w3 \wedge w2 \neq w3$

In what follows, we reserve the symbols *i1*, *i2* and *i3* for worlds, and *r* for an accessibility relation.

We applied the following methodology in the experiments reported in this section:

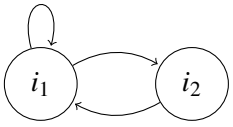
(Step A) First we deliberately made invalid conjectures about inclusion relations—e.g. for proving K4 $>$ K we first wrongly conjectured that K4 $\subseteq$ K, meaning that K4 entails K. We did this by conjecturing

$$\textit{lemma C1-A}\colon \forall R.\ (\textit{trans}\ R)$$

These wrongly-conjectured lemmata in Step A are uniformly named *C*$*$*-A*. Note that for the formulation of the *C*$*$*-A*-lemmata we again exploit the correspondence results given earlier, and we work with conditions on the accessibility relations instead of using the corresponding modal logic axioms. For each *C*$*$*-A*-lemma Nitpick quickly generates a countermodel, which it communicates in a specific syntax. For example, the countermodel it presents for *C1-A* is

$$R = (\lambda x.\ \text{-})(i_1 := (\lambda x.\ \text{-})(i_1 := \textit{True}, i_2 := \textit{True}), i_2 := (\lambda x.\ \text{-})(i_1 := \textit{True}, i_2 := \textit{False}))\ .$$

Diagrammatically this 2-world countermodel can be represented as follows



(Step B) Next, we systematically employed the arity information obtained from the countermodels for the *C*$*$*-A*-lemmata, reported by Nitpick, to formulate a corresponding lemma to be passed via Sledgehammer to the HOL-ATPs LEO-II, Satallax and/or CVC4 [2] (whenever it was not trivially provable by the automation tools *simp*, *force* and/or *blast* available within Isabelle/HOL). In our running example this lemma is

$$\textit{C1-B}\colon \#^2\ i1\ i2 \longrightarrow \forall R.\ \neg\ \textit{trans}\ R$$

All but four of these lemmata can actually be proved by either LEO-II or Satallax. Some of the easier problems can already be automated with *simp*, *force* and *blast*, which are preferred here. The four cases in which no automation attempts succeeded (we also tried all other integrated

ATPs in Isabelle) are named *C∗-ATP-challenge* below. Moreover, there are ten problems named *C∗-Isabelle-challenge*. For these problems LEO-II or Satallax found proofs, but their Metis-based integration into Isabelle failed. Hence, no verification was obtained for these problems. However, we found that five of these *C∗-Isabelle-challenge* problems can also be proved by CVC4, for which proof integration worked. Unfortunately, no other automation means (including the integrated first-order ATPs or SMT solvers) succeeded for the *C∗-Isabelle-challenge* problems.

(Step C) For the verification of the modal logic cube, the non-proved or non-integrated *C∗-challenge* problems of Step B are clearly unsatisfactory, since no proper verification in Isabelle is obtained. However, an easy solution for these (and all other) cases is possible by exploiting not only Nitpick's arity information on the countermodels, but by using all the information about the countermodels it presents, that is, the precise information on the accessibility relation. For example, Nitpick's countermodel for *C1-A* from above can be converted into the following theorem (where *r* denotes a fixed accessibility relation)

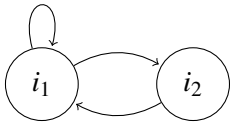
$$\textit{theorem C1-C}\colon \#^2\ i1\ i2 \wedge r\ i1\ i1 \wedge r\ i1\ i2 \wedge r\ i2\ i1 \wedge \neg r\ i2\ i2 \longrightarrow \neg\ \textit{trans}\ r.$$

The resulting theorems we generate are uniformly named *C∗-C*. It turns out that all *C∗-C*-theorems can be quickly verified in Isabelle by Metis. Thus, for each link in the modal logic we provide either a verified *C∗-B* theorem or, if this was not successful, a verified *C∗-C* theorem. Taken together, this confirms that the inclusion relation in the cube are indeed proper. Hence, these *C∗-B* resp. *C∗-C* theorems complete the verification of the modal logic cube. Below the *C∗-C* proof attempts are omitted if the corresponding *C∗-B* attempts were already successful.

(Step D) We additionally prove that the countermodels found by Nitpick in Step A are minimal (regarding the number of possible worlds). In other words, we prove here that the world model constraints as exploited in Step B are in fact minimal constraints under which the inclusion relations can be shown to be proper. Of course, if such a countermodel consists of one possible world only, nothing needs to be shown.

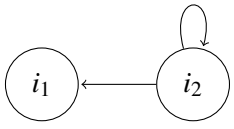
Note that the entire process sketched above, that is the schematic Steps A-D, could be fully automated, meaning that the formulation of the lemmata and theorems in each step could be obtained automatically by analyzing and converting Nitpick's output. In our experiments we still wrote and invoked the verification of each link in the modal cube manually however. Clearly, automation facilities could be very useful for the exploration of the meta-theory of other logics, for example, conditional logics [4], since the overall methodology is obviously transferable to other logics of interest.

3.3.1 K4 > K



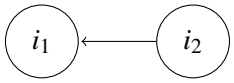
lemma *C1-A*: $\forall R.\ \textit{trans}\ R$ **nitpick oops**
theorem *C1-B*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ \textit{trans}\ R)$ **by** (*simp add*:*trans-def*, *force*)
lemma *C1-D*: $\#^1\ i1 \longrightarrow (\forall R.\ \textit{trans}\ R)$ **by** (*metis* (*lifting*, *full-types*) *trans-def*)

3.3.2 K5 > K



lemma *C2-A*: $\forall R.\ \textit{eucl}\ R$ **nitpick oops**
theorem *C2-B*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ \textit{eucl}\ R)$ **by** (*simp add*:*eucl-def*, *force*)
lemma *C2-D*: $\#^1\ i1 \longrightarrow (\forall R.\ \textit{eucl}\ R)$ **by** (*metis* (*lifting*, *full-types*) *eucl-def*)

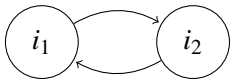
3.3.3 KB > K



lemma *C3-A*: $\forall R.\ sym\ R$ **nitpick oops**
theorem *C3-B*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ sym\ R)$ **by** (*simp add*:*sym-def*, *force*)
lemma *C3-D*: $\#^1\ i1 \longrightarrow (\forall R.\ sym\ R)$ **by** (*metis* (*full-types*) *sym-def*)

3.3.4 K45 > K4

lemma *C4-A*: $\forall R.\ ser\ R \longrightarrow (ser\ R \wedge eucl\ R)$ **nitpick oops**
lemma *C4-B-Isabelle-challenge*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ ser\ R \longrightarrow (ser\ R \wedge eucl\ R))$
— sledgehammer [remote_leo2](ser_def eucl_def)
— CPU time: 13.74 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] – timed out **oops**

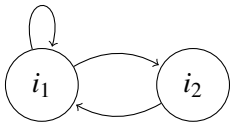


theorem *C4-C*: $\#^2\ i1\ i2\ \wedge \neg r\ i1\ i1 \wedge r\ i1\ i2 \wedge r\ i2\ i1 \wedge \neg r\ i2\ i2 \longrightarrow \neg\ (ser\ r \longrightarrow (ser\ r \wedge eucl\ r))$
by (*metis ser-def eucl-def*)
lemma *C4-D*: $\#^1\ i1\ \longrightarrow (\forall R.\ ser\ R \longrightarrow (ser\ R \wedge eucl\ R))$ **by** (*metis* (*full-types*) *eucl-def*)

3.3.5 K45 > K5

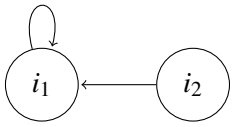
lemma *C5-A*: $\forall R.\ eucl\ R \longrightarrow (ser\ R \wedge eucl\ R)$
nitpick oops
lemma *C5-B-Isabelle-challenge*: $\#^1\ i1 \longrightarrow \neg\ (\forall R.\ (eucl\ R) \longrightarrow (ser\ R) \wedge (eucl\ R))$
— sledgehammer [remote_leo2](eucl_def ser_def) – CPU time: 14.61 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] – timed out **oops**
theorem *C5-C*: $\#^1\ i1 \wedge \neg r\ i1\ i1 \longrightarrow \neg\ (eucl\ r \longrightarrow (ser\ r \wedge eucl\ r))$ **by** (*metis* (*full-types*) *eucl-def ser-def*)

3.3.6 KB5 > KB



lemma *C6-A*: $\forall R.\ sym\ R \longrightarrow (sym\ R \wedge eucl\ R)$
nitpick oops
lemma *C6-B-Isabelle-challenge*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ sym\ R \longrightarrow (sym\ R \wedge eucl\ R))$
— sledgehammer [remote_leo2,timeout=200](sym_def eucl_def) – CPU time: 29.0 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] suggested following line:
by (*metis* (*full-types*) *A4 B8 C1-B IV-def sym-def*)
lemma *C6-D*: $\#^1\ i1 \longrightarrow (\forall R.\ sym\ R \longrightarrow (sym\ R \wedge eucl\ R))$
by (*metis* (*full-types*) *eucl-def*)

3.3.7 KB5 > K45



lemma *C7-A*: $\forall R.\ ser\ R \wedge eucl\ R \longrightarrow (sym\ R \wedge eucl\ R)$
nitpick oops
lemma *C7-B-Isabelle-challenge*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ ser\ R \wedge eucl\ R \longrightarrow (sym\ R \wedge eucl\ R))$
— sledgehammer [remote_leo2] (ser_def eucl_def sym_def) – CPU time: 11.15 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] – timed out **oops**
theorem *C7-C*: $\#^2\ i1\ i2 \wedge r\ i1\ i1 \wedge \neg\ r\ i1\ i2 \wedge r\ i2\ i1 \wedge \neg\ r\ i2\ i2 \longrightarrow \neg\ (ser\ r \wedge eucl\ r \longrightarrow (sym\ r \wedge eucl\ r))$
by (*metis* (*full-types*) *ser-def eucl-def sym-def*)
lemma *C7-D*: $\#^1\ i1 \longrightarrow (\forall R.\ ser\ R \wedge eucl\ R \longrightarrow (sym\ R \wedge eucl\ R))$ **by** (*metis* (*full-types*) *sym-def*)

3.3.8 D > K

lemma *C8-A*: $\forall R.\ ser\ R$ **nitpick oops**
lemma *C8-B*: $\#^1\ i1 \longrightarrow \neg(\forall R.\ (ser\ R))$ **by** (*simp add*:*ser-def*, *force*)
theorem *C8-C*: $\#^1\ i1 \wedge \neg r\ i1\ i1 \longrightarrow \neg(ser\ r)$ **by** (*metis* (*full-types*) *ser-def*)

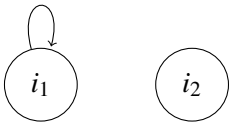
3.3.9 D4 > K4

lemma *C9-A*: $\forall R.\ trans\ R \longrightarrow (ser\ R \wedge trans\ R)$
nitpick oops
theorem *C9-B*: $\#^1\ i1 \longrightarrow \neg\ (\forall R.\ trans\ R \longrightarrow (ser\ R \wedge trans\ R))$
using *C1-D C8-B* **by** *blast*

3.3.10 D5 > K5

lemma *C10-A*: $\forall R.\ eucl\ R \longrightarrow (ser\ R \wedge eucl\ R)$ **nitpick oops**
theorem *C10-B*: $\#^1\ i1\ \longrightarrow \neg\ (\forall R.\ eucl\ R \longrightarrow (ser\ R \wedge eucl\ R))$ **using** *B9 C3-D C9-B* **by** *blast*

3.3.11 D45 > K45



lemma *C11-A*: $\forall R.\ trans\ R \wedge eucl\ R \longrightarrow (ser\ R \wedge trans\ R \wedge eucl\ R)$
nitpick oops
theorem *C11-B*: $\#^1\ i1 \longrightarrow \neg\ (\forall R.\ trans\ R \wedge eucl\ R \longrightarrow (ser\ R \wedge trans\ R \wedge eucl\ R))$
using *B9 C3-D C9-B* **by** *blast*

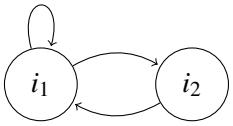
3.3.12 DB > KB

lemma *C12-A*: $\forall R.\ sym\ R \longrightarrow (ser\ R \wedge sym\ R)$
nitpick oops
theorem *C12-B*: $\#^1\ i1 \longrightarrow \neg\ (\forall R.\ sym\ R \longrightarrow (ser\ R \wedge sym\ R))$
using *C11-B C3-D* **by** *blast*

3.3.13 S5 > KB5

lemma *C13-A*: $\forall R.\ sym\ R \wedge eucl\ R \longrightarrow (refl\ R \wedge eucl\ R)$
nitpick oops
theorem *C13-B*: $\#^1\ i1 \longrightarrow \neg\ (\forall R.\ sym\ R \wedge eucl\ R \longrightarrow (refl\ R \wedge eucl\ R))$ **using** *B5 C12-B C6-D* **by** *blast*

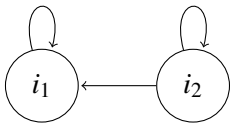
3.3.14 D4 > D



lemma *C14-A*: $\forall R.\ (ser\ R) \longrightarrow (ser\ R) \wedge (trans\ R)$
nitpick oops
theorem *C14-B-Isabelle-challenge*: $\#^2\ i1\ i2 \longrightarrow \neg(\forall R.\ ser\ R \longrightarrow (ser\ R \wedge trans\ R))$
— sledgehammer [remote_leo2] (ser_def trans_def) – CPU time: 13.08 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] suggested following line:
by (*metis* (*full-types*) *C1-B trans-def ser-def*)
lemma *C14-D*: $\#^1\ i1 \longrightarrow (\forall R.\ ser\ R \longrightarrow (ser\ R \wedge trans\ R))$ **by** (*metis* (*full-types*) *trans-def*)

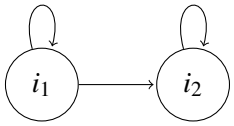
3.3.15 D5 > D

lemma *C15-A*: $\forall R.\ ser\ R \longrightarrow (ser\ R \wedge eucl\ R)$
nitpick oops
theorem *C15-B-Isabelle-challenge*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ ser\ R \longrightarrow (ser\ R \wedge eucl\ R))$
— sledgehammer [remote_leo2](ser_def eucl_def)
— CPU time: 12.9 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] suggested following line:
by (*metis* (*full-types*) *C14-B-Isabelle-challenge trans-def eucl-def*)
lemma *C15-D*: $\#^1\ i1 \longrightarrow (\forall R.\ ser\ R \longrightarrow (ser\ R \wedge eucl\ R))$ **by** (*metis* (*full-types*) *C2-D*)



3.3.16 DB > D

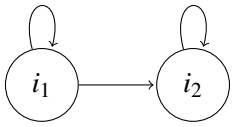
lemma *C16-A*: $\forall R.\ ser\ R \longrightarrow (ser\ R \wedge sym\ R)$
nitpick oops



lemma *C16-B*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ ser\ R \longrightarrow (ser\ R \wedge sym\ R))$ **by** (*simp add*:*ser-def sym-def*, *force*)
lemma *C16-D*: $\#^1\ i1 \longrightarrow (\forall R.\ ser\ R \longrightarrow (ser\ R \wedge sym\ R))$ **by** (*metis* (*full-types*) *sym-def*)

3.3.17 D45 > D4

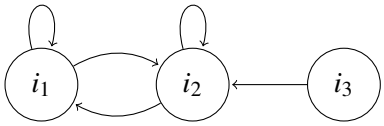
lemma *C17-A*: $\forall R.\ ser\ R \wedge trans\ R \longrightarrow (ser\ R \wedge trans\ R \wedge eucl\ R)$
nitpick oops



lemma *C17-B-ATP-challenge*: $\#^2\ i1\ i2 \longrightarrow \neg(\forall R.\ ser\ R \wedge trans\ R \longrightarrow (ser\ R \wedge trans\ R \wedge eucl\ R))$
oops — All ATPs time out
theorem *C17-C*: $\#^2\ i1\ i2 \wedge r\ i1\ i1 \wedge r\ i1\ i2 \wedge \neg\ r\ i2\ i1 \wedge r\ i2\ i2 \longrightarrow \neg\ (ser\ r \wedge trans\ r \longrightarrow (ser\ r \wedge trans\ r \wedge eucl\ r))$
by (*metis* (*full-types*) *ser-def trans-def eucl-def*)
lemma *C17-D*: $\#^1\ i1 \longrightarrow (\forall R.\ ser\ R \wedge trans\ R \longrightarrow (ser\ R \wedge trans\ R \wedge eucl\ R))$
by (*metis* (*full-types*) *eucl-def*)

3.3.18 D45 > D5

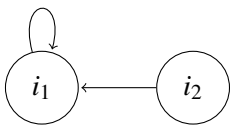
lemma *C18-A*: $\forall R.\ ser\ R \wedge eucl\ R \longrightarrow (ser\ R \wedge trans\ R \wedge eucl\ R)$
nitpick oops



lemma *C18-ATP-challenge*: $\#^3\ i1\ i2\ i3 \longrightarrow \neg\ (\forall R.\ ser\ R \wedge eucl\ R \longrightarrow (ser\ R \wedge trans\ R \wedge eucl\ R))$
oops — All ATPs time out
theorem *C18-C*: $\#^3\ i1\ i2\ i3 \wedge r\ i1\ i1 \wedge r\ i1\ i2 \wedge \neg\ r\ i1\ i3 \wedge r\ i2\ i1 \wedge r\ i2\ i2 \wedge \neg\ r\ i2\ i3 \wedge \neg\ r\ i3\ i1 \wedge r\ i3\ i2 \wedge \neg\ r\ i3\ i3 \longrightarrow \neg\ (ser\ r \wedge eucl\ r \longrightarrow (ser\ r \wedge trans\ r \wedge eucl\ r))$ **by** (*metis* (*full-types*) *eucl-def ser-def trans-def*)
lemma *C18-D*: $\#^2\ i1\ i2 \longrightarrow (\forall R.\ ser\ R \wedge eucl\ R \longrightarrow (ser\ R \wedge trans\ R \wedge eucl\ R))$
by (*metis* (*full-types*) *eucl-def trans-def*)

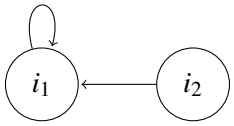
3.3.19 M > D

lemma *C19-A*: $\forall R.\ ser\ R \longrightarrow refl\ R$
nitpick oops



theorem *C19-B-Isabelle-challenge*: $\#^2$ *i1 i2* $\longrightarrow \neg\ (\forall R.\ ser\ R \longrightarrow refl\ R)$
— sledgehammer [remote_leo2,timeout=200] (ser_def refl_def) – CPU time: 29.15 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] suggested following line:
by (*metis* (*full-types*) *C14-B-Isabelle-challenge trans-def refl-def*)
lemma *C19-D*: $\#^1$ *i1* $\longrightarrow (\forall R.\ ser\ R \longrightarrow refl\ R)$ **by** (*metis* (*full-types*) *ser-def refl-def*)

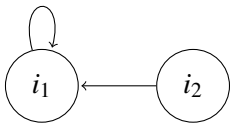
3.3.20 S4 > D4



lemma *C20-A*: $\forall R.\ ser\ R \wedge trans\ R \longrightarrow (refl\ R \wedge trans\ R)$
nitpick oops

lemma *C20-B-Isabelle-challenge*: $\#^2$ *i1 i2* $\longrightarrow \neg\ (\forall R.\ ser\ R \wedge trans\ R \longrightarrow (refl\ R \wedge trans\ R))$
— sledgehammer [remote_leo2](ser_def trans_def refl_def) – CPU time: 12.5 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] – timed out
oops
theorem *C20-C*: $\#^2$ *i1 i2* $\wedge$ *r i1 i1* $\wedge \neg$ *r i1 i2* $\wedge$ *r i2 i1* $\wedge \neg$ *r i2 i2* $\longrightarrow \neg\ (ser\ r \wedge trans\ r \longrightarrow (refl\ r \wedge trans\ r))$
by (*metis* (*full-types*) *ser-def refl-def trans-def*)
lemma *C20-D*: $\#^1$ *i1* $\longrightarrow (\forall R.\ ser\ R \wedge trans\ R \longrightarrow (refl\ R \wedge trans\ R))$
by (*metis* (*full-types*) *ser-def refl-def*)

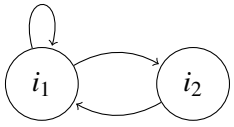
3.3.21 S5 > D45



lemma *C21-A*: $\forall R.\ ser\ R \wedge trans\ R \wedge eucl\ R \longrightarrow (refl\ R \wedge eucl\ R)$
nitpick oops

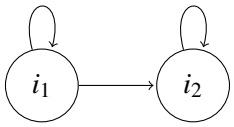
lemma *C21-B-Isabelle-challenge*: $\#^2$ *i1 i2* $\longrightarrow \neg\ (\forall R.\ ser\ R \wedge trans\ R \wedge eucl\ R \longrightarrow (refl\ R \wedge eucl\ R))$
— sledgehammer [remote_leo2](ser_def trans_def eucl_def refl_def) – CPU time: 12.51 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] – timed out
oops
theorem *C21-C*: $\#^2$ *i1 i2* $\wedge$ *r i1 i1* $\wedge \neg$ *r i1 i2* $\wedge$ *r i2 i1* $\wedge \neg$ *r i2 i2* $\longrightarrow \neg\ (ser\ r \wedge trans\ r \wedge eucl\ r \longrightarrow (refl\ r \wedge eucl\ r))$
by (*metis* (*full-types*) *ser-def trans-def eucl-def refl-def*)
lemma *C21-inclusion*: $\#^1$ *i1* $\longrightarrow (\forall R.\ ser\ R \wedge trans\ R \wedge eucl\ R \longrightarrow (refl\ R \wedge eucl\ R))$
by (*metis* (*full-types*) *ser-def refl-def*)

3.3.22 B > DB



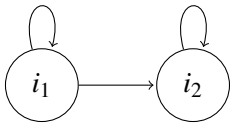
lemma *C22-A*: $\forall R.\ ser\ R \wedge sym\ R \longrightarrow (refl\ R \wedge sym\ R)$
nitpick oops
lemma *C22-B-Isabelle-challenge*: $\#^2$ *i1 i2* $\longrightarrow \neg\ (\forall R.\ ser\ R \wedge sym\ R \longrightarrow (refl\ R \wedge sym\ R))$
— sledgehammer [remote_leo2,timeout=200](ser_def sym_def refl_def) – CPU time: 31.18 s. Metis reconstruction failed.
— sledgehammer [cvc4,timeout=300] suggested following line:
— by (smt C14_B sym_def trans_def refl_def) **oops**
theorem *C22-C*: $\#^2$ *i1 i2* $\wedge$ *r i1 i1* $\wedge$ *r i1 i2* $\wedge$ *r i2 i1* $\wedge \neg$ *r i2 i2* $\longrightarrow \neg\ (ser\ r \wedge sym\ r \longrightarrow (refl\ r \wedge sym\ r))$
by (*metis* (*full-types*) *ser-def sym-def refl-def*)
lemma *C22-D*: $\#^1$ *i1* $\longrightarrow (\forall R.\ ser\ R \wedge sym\ R \longrightarrow (refl\ R \wedge sym\ R))$
by (*metis* (*full-types*) *ser-def refl-def*)

3.3.23 B > M



lemma *C23-A*: $\forall R.\ refl\ R \longrightarrow (refl\ R \wedge sym\ R)$ **nitpick oops**
lemma *C23-B-ATP-challenge*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ refl\ R \longrightarrow (refl\ R \wedge sym\ R))$
oops — All ATPs time out
theorem *C23-C*: $\#^2\ i1\ i2\ \wedge r\ i1\ i1 \wedge r\ i1\ i2 \wedge \neg\ r\ i2\ i1 \wedge r\ i2\ i2\ \longrightarrow \neg\ (refl\ r \longrightarrow (refl\ r \wedge sym\ r))$
by (*metis refl-def sym-def*)
lemma *C23-D*: $\#^1\ i1 \longrightarrow (\forall R.\ refl\ R \longrightarrow (refl\ R \wedge sym\ R))$ **by** (*metis* (*full-types*) *sym-def*)

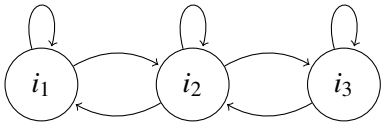
3.3.24 S5 > S4



lemma *C24-A*: $\forall R.\ refl\ R \wedge trans\ R \longrightarrow (refl\ R \wedge eucl\ R)$
nitpick oops

lemma *C24-B-ATP-challenge*: $\#^2\ i1\ i2 \longrightarrow \neg\ (\forall R.\ refl\ R \wedge trans\ R \longrightarrow (refl\ R \wedge eucl\ R))$
oops — All ATPs time out
theorem *C24-C*: $\#^2\ i1\ i2 \wedge r\ i1\ i1 \wedge r\ i1\ i2 \wedge \neg\ r\ i2\ i1 \wedge r\ i2\ i2 \longrightarrow \neg\ (refl\ r \wedge trans\ r \longrightarrow (refl\ r \wedge eucl\ r))$
by (*metis* (*full-types*) *trans-def refl-def eucl-def*)
lemma *C24-D*: $\#^1\ i1 \longrightarrow (\forall R.\ refl\ R \wedge trans\ R \longrightarrow (refl\ R \wedge eucl\ R))$ **by** (*metis* (*full-types*) *eucl-def*)

3.3.25 S5 > B



lemma *C25-A*: $\forall R.\ refl\ R \wedge sym\ R \longrightarrow (refl\ R \wedge eucl\ R)$
nitpick oops

lemma *C25-B-ATP-challenge*: $\#^3\ i1\ i2\ i3 \longrightarrow \neg\ (\forall R.\ (refl\ R \wedge sym\ R) \longrightarrow (refl\ R \wedge eucl\ R))$
oops — All ATPs time out
theorem *C25-C*: $\#^3\ i1\ i2\ i3 \wedge r\ i1\ i1 \wedge r\ i1\ i2 \wedge \neg\ r\ i1\ i3 \wedge r\ i2\ i1 \wedge r\ i2\ i2 \wedge r\ i2\ i3 \wedge \neg\ r\ i3\ i1 \wedge r\ i3\ i2 \wedge r\ i3\ i3$
$\longrightarrow \neg\ ((refl\ r \wedge sym\ r) \longrightarrow (refl\ r \wedge eucl\ r))$
by (*metis* (*full-types*) *eucl-def refl-def sym-def*)
lemma *C25-D*: $\#^2\ i1\ i2 \longrightarrow (\forall R.\ (refl\ R \wedge sym\ R) \longrightarrow (refl\ R \wedge eucl\ R))$
by (*metis* (*full-types*) *refl-def sym-def eucl-def*)

4 Discussion and Future Work.

The entire Isabelle document can be verified by Isabelle2014 in less than 60s on a semi-modern computer (2.4 GHz Core 2 Duo, 8 GB of memory). When including all (commented) remote calls to the external ATPs in the calculation the verification time sums up to a few minutes, which is still very reasonable.

The improvements in comparison to the first-order based verification of the modal logic cube done earlier by Rabe et al. [16], are: clarity and readability of the problem encodings, methodology, reliability (our proofs are verifiable in Isabelle/HOL) and, most importantly, automation performance. For the latter note that the experiments by Rabe et al. [16] required several days of reasoning time in first-order theorem provers. Most importantly, however, their solution relied on an enormous manual coding effort. However, we want to point again to the more general aims of their work.

Our solution instead requires a small amount of resources in comparison. In fact, as indicated before, the entire process (Steps A-D) is schematic, so that it should eventually be possible to fully automate

our method. For this it would be beneficial to have a flexible and accessible conversion of the countermodels delivered by Nitpick back into Isabelle/HOL input syntax. In fact, an automated conversion of Nitpick's countermodels into the corresponding *C∗-B* and *C∗-C* conjectures would eventually enable a truly automated exploration and verification of of the modal logic cube with no or minimal handcoding effort. Similarly, for the interactive user a more intuitive presentation of Nitpick's countermodels would be welcome (perhaps similar to the illustrations we used in this paper).

Using the first-order provers E [17], SPASS [19], Z3 [13] and Vampire [12] proved unsuccessful for all *C∗-Isabelle-challenge* problems (unless the right lemmas were given to them). Analyzing the reason for their weakness, as compared to the better performing higher-order automated theorem provers, remains future work. In contrast, the SMT solver CVC4 (via Sledgehammer) was quite successful and contributed five *C∗-Isabelle-challenge* proofs.

Our work motivates further improvements regarding the integration of LEO-II and Satallax: While these systems are able to prove all *∗-Isabelle-challenge* problems their proofs cannot yet be easily replayed or integrated in Isabelle/HOL. There have been recent improvements regarding the transformation of proofs from LEO-II and Satallax to Isabelle/HOL [18], using which all the proofs produced by Satallax and LEO-II in our work could be checked in Isabelle/HOL,[1] but this process still requires some manual work to adapt the output from the ATPs.

Our work also motivates further improvements in higher-order automated theorem provers. For example, for these systems it should be possible to also prove the remaining two *∗-ATP-challenge* problems. Moreover, they needed more than 10 seconds of CPU time in our experiments for the *∗-Isabelle-challenge* problems; it should be possible to prove these theorems much faster.

5 Conclusion

We have fully verified the modal logic cube in Isabelle/HOL. Our solution is simple, elegant, easy to follow, effective and efficient. Proof exchange between systems played a crucial role in our experiments. In particular, we have exploited and combined Nitpick's countermodel-finding capabilities with subsequent calls to the higher-order theorem provers LEO-II and Satallax and the SMT solver CVC4 via Isabelle's Sledgehammer tool. Our experiments also point to several improvement opportunities for Isabelle and the higher-order reasoners, in particular, regarding interaction and proof exchange.

Related experiments have been carried out earlier in collaboration with Geoff Sutcliffe. Similar to and improving on the work reported in [3], these unpublished experiments used the TPTP THF infrastructure directly. However, in that work we did not achieve a 'trusted verification' in the sense of the work presented in this paper. Another improvement in this article has been the use of schematic meta-level working steps (Steps A-D) to systematically convert (counter)models found by Nitpick into conjectures to be investigated.

Future work will explore and evaluate similar logic relationships for other non-classical logics, for example, conditional logics. Any improvements in the mentioned systems, as motivated above, would be very beneficial towards this planned work. Moreover, it would be useful to fully automate the schematic, meta-level working steps (Steps A-D) as applied in our experiments. This would produce a system that would explore logic relations truly automatically (for example, in conditional logics), analogous to what has been achieved here for the modal logic cube.

[1] The proofs and the evaluation workflow can be downloaded from http://christoph-benzmueller.de/papers/pxtp2015-eval.zip

Acknowledgements: We thank Florian Rabe and the anonymous reviewers of this paper for their valuable feedback.

References

[1] J. Backes & C.E. Brown (2010): *Analytic Tableaux for Higher-Order Logic with Choice*. In J. Giesl & R. Hähnle, editors: *Automated Reasoning*, *Lecture Notes in Computer Science* 6173, Springer Berlin Heidelberg, pp. 76–90, doi:10.1007/978-3-642-14203-1_7.

[2] C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds & C. Tinelli (2011): *CVC4*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification*, *LNCS* 6806, Springer Berlin Heidelberg, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[3] C. Benzmüller (2010): *Verifying the Modal Logic Cube is an Easy Task (for Higher-Order Automated Reasoners)*. In S. Siegler & N. Wasser, editors: *Verification, Induction, Termination Analysis - Festschrift for Christoph Walther on the Occasion of His 60th Birthday*, *LNCS* 6463, Springer, pp. 117–128, doi:10.1007/978-3-642-17172-7_7. Available at http://christoph-benzmueller.de/papers/B12.pdf.

[4] C. Benzmüller (2013): *Automating Quantified Conditional Logics in HOL*. In F. Rossi, editor: *23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*, Beijing, China, pp. 746–753. Available at http://ijcai.org/papers13/Papers/IJCAI13-117.pdf.

[5] C. Benzmüller & B. Woltzenlogel Paleo (2013): *Gödel's God in Isabelle/HOL*. *Archive of Formal Proofs*. http://afp.sf.net/entries/GoedelGod.shtml, Formal proof development.

[6] C. Benzmüller & B. Woltzenlogel Paleo (2014): *Automating Gödel's Ontological Proof of God's Existence with Higher-order Automated Theorem Provers*. In T. Schaub, G. Friedrich & B. O'Sullivan, editors: *ECAI 2014*, *Frontiers in Artificial Intelligence and Applications* 263, IOS Press, pp. 93 – 98, doi:10.3233/978-1-61499-419-0-93. Available at http://christoph-benzmueller.de/papers/C40.pdf.

[7] C. Benzmüller & L.C. Paulson (2013): *Quantified Multimodal Logics in Simple Type Theory*. *Logica Universalis (Special Issue on Multimodal Logics)* 7(1), pp. 7–20, doi:10.1007/s11787-012-0052-y. Available at http://christoph-benzmueller.de/papers/J23.pdf.

[8] C. Benzmüller, F. Rabe & G. Sutcliffe (2008): *THF0 – The Core of the TPTP Language for Classical Higher-Order Logic*. In A. Armando, P. Baumgartner & G. Dowek, editors: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, *LNCS* 5195, Springer, pp. 491–506, doi:10.1007/978-3-540-71070-7_41. Available at http://christoph-benzmueller.de/papers/C25.pdf.

[9] C. Benzmüller, F. Theiss, L.C. Paulson & A. Fietzke (2008): *LEO-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic (System Description)*. In A. Armando, P. Baumgartner & G. Dowek, editors: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, *LNCS* 5195, Springer, pp. 162–170, doi:10.1007/978-3-540-71070-7_14. Available at http://christoph-benzmueller.de/papers/C26.pdf.

[10] J.C. Blanchette & T. Nipkow (2010): *Nitpick: A counterexample generator for higher-order logic based on a relational model finder*. In M. Kaufmann & L. Paulson, editors: *Interactive Theorem Proving (ITP 2010)*, 6172, pp. 131–146, doi:10.1007/978-3-642-14052-5_11.

[11] J. Hurd (2003): *First-Order Proof Tactics in Higher-Order Logic Theorem Provers*. In M. Archer, B. Di Vito & C. Muñoz, editors: *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, NASA Technical Reports, pp. 56–68. Available at http://www.gilith.com/research/papers.

[12] L. Kovács & A. Voronkov (2013): *First-Order Theorem Proving and Vampire*. In N. Sharygina & H. Veith, editors: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, *LNCS* 8044, Springer, pp. 1–35, doi:10.1007/978-3-642-39799-8_1.

[13] L.M. de Moura & N. Bjørner (2008): *Z3: An Efficient SMT Solver*. In C.R. Ramakrishnan & J. Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[14] T. Nipkow, L.C. Paulson & M. Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer.

[15] L.C. Paulson & J.C. Blanchette (2012): *Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers*. In G. Sutcliffe, S. Schulz & E. Ternovska, editors: *IWIL 2010*, *EPiC Series* 2, EasyChair, pp. 1–11.

[16] F. Rabe, P. Pudlak, G. Sutcliffe & W. Shen (2009): *Solving the $ 100 modal logic challenge*. *Journal of Applied Logic* 7, pp. 113–130, doi:10.1016/j.jal.2007.07.007.

[17] S. Schulz (2013): *System Description: E 1.8*. In K.L. McMillan, A. Middeldorp & A. Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, *LNCS* 8312, Springer, pp. 735–743, doi:10.1007/978-3-642-45221-5_49.

[18] N. Sultana (2015): *Higher-order proof translation*. Ph.D. thesis, Computer Laboratory, University of Cambridge. Available as Tech Report UCAM-CL-TR-867.

[19] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda & P. Wischnewski (2009): *SPASS Version 3.5*. In R.A. Schmidt, editor: *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, *LNCS* 5663, Springer, pp. 140–145, doi:10.1007/978-3-642-02959-2_10.

The Common HOL Platform

Mark Adams
Proof Technologies Ltd, UK
Radboud University, Nijmegen, The Netherlands

The Common HOL project aims to facilitate porting source code and proofs between members of the HOL family of theorem provers. At the heart of the project is the Common HOL Platform, which defines a standard HOL theory and API that aims to be compatible with all HOL systems. So far, HOL Light and hol90 have been adapted for conformance, and HOL Zero was originally developed to conform. In this paper we provide motivation for a platform, give an overview of the Common HOL Platform's theory and API components, and show how to adapt legacy systems. We also report on the platform's successful application in the hand-translation of a few thousand lines of source code from HOL Light to HOL Zero.

1 Introduction

The HOL family of theorem provers started in the 1980s with HOL88 [5], and has since grown to include many systems, most prominently HOL4 [16], HOL Light [8], ProofPower HOL [3] and Isabelle/HOL [12]. These four main systems have developed their own advanced proof facilities and extensive theory libraries, and have been successfully employed in major projects in the verification of critical hardware and software [1, 11] and the formalisation of mathematics [7].

It would clearly be of benefit if these systems could "talk" to each other, specifically if theory, proofs and source code could be exchanged in a relatively seamless manner. This would reduce the considerable duplication of effort otherwise required for one system to benefit from the major projects and advanced capabilities developed on another. Work to date has concentrated on exchange of proofs via proof objects, with some degree of success, but little has been done to facilitate porting of source code.

The Common HOL Platform is part of the Common HOL project for facilitating the porting of source code and proofs between HOL systems. It defines a standard HOL theory compatible with the core theory of each HOL system, and an application programming interface (API) of programming components that is more-or-less common to all HOL systems. It has so far been supported in HOL Light, hol90 [15] and HOL Zero [19].

In this paper we give an overview of the platform. In Section 2, we further discuss motivation. In Section 3, we cover the platform's choice of components. In Section 4, we explain how to adapt legacy systems to conform to the platform. In Section 5, we report on its successful usage in assisting the manual porting of both new and legacy source code. In Section 6, we present our conclusions.

2 Motivation

By definition, all systems in the HOL family implement the HOL logic or a close variant. However, in practice their commonality stretches far beyond this. They have broadly similar axiomatisations of the logic, similar mechanisms for logical extension, similar formal language concrete syntax and build up similar foundational theory. Furthermore, in most basic usage at least, they each support

C.Kaliszyk and A. Paskevich (Eds.): 4th Workshop
on Proof eXchange for Theorem Proving (PxTP'15).
EPTCS 186, 2015, pp. 42–56, doi:10.4204/EPTCS.186.6

similar paradigms of user interaction, namely simple forwards-style application of inference rules and backwards-style tactic proofs via the subgoal goal package [14], performed in an interactive functional programming session. Also, their implementations are all written in variants of the ML functional programming language, all employ an LCF-style architecture [6] and are all built up from similar libraries of programming utilities, syntax utilities, inference rules and tactics.

Other than in these basic aspects, the systems branch off in their own respects. Each builds up considerable theory beyond the basic foundations in its own way. For example, real numbers in HOL Light are constructed quite differently from real numbers in ProofPower HOL. There is also much variation in their provision of user proof commands, especially for those relating to proof automation, with each system having its own strengths and idiosyncrasies. Most different is Isabelle/HOL, which is implemented as an instantiation of the Isabelle generic theorem prover [17] rather than by having its deductive system "hardwired" as source code, and supports a variant of the HOL logic that has axiomatic type classes. Also, the predominant mode of interaction with Isabelle has become the declarative proof language Isar in conjunction with a bespoke IDE, rather than the subgoal package in an interactive ML session.

Porting proofs between HOL systems by hand involves translating proofs scripts. These proof scripts typically involve heavy use of high-level proof commands that differ between systems. In cases where such commands are used to finish off subgoals, it is often possible to find a suitably powerful command to do the same in the target system, but in other cases proof scripts have to be recreated from scratch. Automatic proof porting, via recording of low-level proof steps and export to proof object files, is vastly preferable if it can be made sufficiently reliable. Such a capability requires a platform of common foundational theory, inference rules and logical extension mechanisms in both systems.

There have been notable successes in the large scale porting of legacy proofs between HOL systems via proof objects. Obua and Skalberg [13] developed a capability for porting proofs from HOL4 to Isabelle/HOL, using a theory platform based on the HOL4 inference kernel, and then adapted this for porting from HOL Light to Isabelle/HOL. Kaliszyk and Krauss [10] developed a capability for porting from HOL Light to Isabelle/HOL, based on the HOL Light inference kernel. The OpenTheory project [9] is based around the HOL Light axiomatisation, and establishes a common proof object format for porting proofs between various HOL systems, including HOL4, ProofPower/HOL and HOL Light, with ongoing work to support Isabelle/HOL. However, these capabilities would all struggle to port something as large as the entire Flyspeck project [7]. We believe that significant advances in capability can be achieved by exploiting a broader commonality that exists between HOL systems, using a platform at a somewhat higher level than the inference kernel of one system.

Porting source code from one system to another currently requires deep knowledge of both systems' implementations and can entail weeks of effort to replicate behaviour sufficiently closely. Naive porting of high-level routines will typically result in unreliable code due to the compounding of small and subtle differences in the theory or in ML function behaviour. We know of no pre-existing capability for supporting the systematic porting of source code between HOL systems.

We believe that if the existing HOL systems can be adapted to support a well-designed API that reflects the commonality of "primary functionality" (by which we mean functionality directly concerned with theorem proving) between the systems, then much of the pain of porting source code can be avoided. There is then a platform of precisely corresponding programming components, and source code built on this platform in one system can be trivially but accurately ported to another system conforming to the same platform. As is also the case for a proof porting capability, both ML components and foundational theory have to be taken into account when designing an effective platform.

3 Components

In this section, we give an overview of the components that make up version 0.5 of the Common HOL Platform. This is the latest version, and has been implemented for HOL Light and HOL Zero. An earlier version was implemented for hol90, but this has not yet been upgraded. Even though the platform has not yet been implemented for ProofPower HOL or HOL4, it has been carefully designed with knowledge of how these systems work. However, little consideration has so far been given to Isabelle/HOL, which presents greater challenges due to its greater differences. A significant redesign of the standard would probably be required to properly cater for Isabelle/HOL.

There is no space in this paper to list all the platform components, let alone to describe each one. Instead we provide various tables comparing some corresponding components from hol90, HOL4, ProofPower HOL, HOL Light and HOL Zero. For a given system, each platform component is either exactly represented in the system, or it is approximately represented, or it is not represented in the system. In our listings, those components only approximately corresponding are written in curly brackets.

There is not yet a single stand-alone document specifically for the purpose of precisely defining each platform component. However, part of the original motivation for the HOL Zero system was to act as a clear demonstration of the platform, and it has been designed to exactly conform to platform behaviour without adaption. Readers can download the HOL Zero source distribution [19], where source code file `commonhol.mli` gives a complete list of the API components, and the user manual appendices give a precise description of each API and theory component.

3.1 Considerations

Here we discuss some factors that should be taken into consideration when choosing the components.

Commonality Platform components should broadly reflect the commonality that exists between the systems. Including components that are only relevant in one system would entail extra effort to make the other systems conformant, and would be of little use to them. Not including components that are common to all systems would mean that basic components from one system would have to be needlessly considered when porting to a target system.

Usage Amount of usage in post-platform code should be taken into consideration when deciding the platform components. Heavily used components should almost qualify by default.

Level The components should be sufficiently high-level to be of likely use in post-platform source code. For example, including low-level subcomponents used to make a HOL term parser would be of little use, even if these components were common to all HOL systems.

Precision A platform without precisely defined components of course loses much of its purpose. In HOL systems, there are many small differences in the details of the behaviour of various corresponding basic functions. For each component, the platform should explicitly specify its exact behaviour or otherwise be clear about what is not specified. Non-conformant components must have platform-conformant variants defined as part of platform qualification.

Underspecification The API should allow some degree of flexibility in certain kinds of details about it components. For example, the ML names of the components, or the order in which function components take arguments and whether tuples or curried form is used. The API should seek to minimise the effort required to make legacy systems conformant by underspecifying these details, which are not the kinds of differences that make porting source code difficult.

Completeness The components should be complete in the sense that all primary functionality can be built from platform components alone. This becomes essential for the constructors and destructors of abstract datatypes (such as for HOL types, terms and theorems) because there is otherwise no way of manipulating such values.

Coherence The components should be chosen as a coherent set that categorise in a complete and consistent way and that composes robustly. This makes it easier to write new code based on the API, as well as helping portability.

Performance The API should not exclude components that are important to the performance of a system if this means they would otherwise need to be reimplemented in the outer platform in terms of API components to result in a significant degradation in performance.

Ease of Implementation The implementation effort required to conform to a platform is a significant consideration. Otherwise, in practice the platform will not get implemented for the full range of HOL systems, which defeats its purpose.

3.2 Theory Components

The theory components are the axioms, declarations and definitions that must exist in a conformant system's theory. They must form a sufficient basis for building up each HOL system's theory.

There is some variation in the systems' axiomatisations, especially between HOL Light and the other systems. Because each system implements the same formal logic, for our purposes of completeness it is sufficient to choose the core theory (i.e. the theory of the logical core) of one system as the theory platform, and to derive this in the other systems from their respective core theories. The outer platform (see Section 4.1) in these other systems can then "re-derive" the system's core theory using the theory platform. A platform theorem may be an axiom or definition theorem in one system and a derived theorem in another, but as far as the platform is concerned they are all just theorems.

Our theory platform features the axioms and definitions of ProofPower HOL, which we view as the most intuitive, and which are close to those of hol90 and HOL4. It also includes the HOL Light definition of the implication operator, which does not feature in the other systems because the behaviour of implication drops out from their primitive inference rules and the implication antisymmetry axiom. Including this definition means that any of the systems' primitive inference rule set suffices to complete the deductive system. A handful of fundamental theorems that are common to but derived in each system are included in the platform, such as the truth theorem and the Law of the Excluded Middle, because they are inevitably needed in implementing the platform and so may as well feature as components.

The type constants and constants declared in the theory platform include those from the basic theory about predicate logic and lambda calculus that is common to each HOL system, established in the logical core and initial derived theory of each system. This includes the function space type operator and the boolean base type, plus the equality, conjunction, disjunction, implication and logical negation operators, the universal, existential and unique existential quantifiers and the Hilbert choice operator.

Beyond this, each system builds up essentially equivalent theory of pairs, lists and natural numbers. To take advantage of this commonality, the platform also includes theory for pairs and natural numbers, including natural number numerals and 13 classic arithmetic operators including plus, multiply and exponentiation. Theory for lists does not currently feature, but is planned for inclusion in a future version.

The representation of natural number numerals varies between HOL systems: in HOL Light, HOL4 and HOL Zero, each numeral is constructed using compounding of two unary operators on the zero constant (one for multiplying by two and adding one, and one for multiplying by two and adding zero or

hol90	HOL4	ProofPower	HOL Light	HOL Zero
"bool"	"bool"	"BOOL"	"bool"	"bool"
"fun"	"fun"	"$\rightarrow$"	"fun"	"->"
"prod"	"prod"	"$\times$"	"prod"	"#"
"ind"	"ind"	"IND"	"ind"	"ind"
"num"	"num"	"$\mathbb{N}$"	"num"	"nat"
"T"	"T"	"T"	"T"	"true"
"F"	"F"	"F"	"F"	"false"
"="	"="	"="	"="	"="
"/\"	"/\"	"$\wedge$"	"/\"	"/\"
"\/"	"\/"	"$\vee$"	"\/"	"\/"
"$\sim$"	"$\sim$"	"$\neg$"	"$\sim$"	"$\sim$"
"!"	"!"	"$\forall$"	"!"	"!"
"?"	"?"	"$\exists$"	"?"	"?"
"?!"	"?!"	"$\exists_1$"	"?!"	"?!"
"@"	"@"	"ε"	"@"	"@"
IMP_ANTISYM_AX	IMP_ANTISYM_AX$^+$	$\Rightarrow$_antisym_axiom	-	imp_antisym_ax
ETA_AX	ETA_AX	η_axiom	ETA_AX	eta_ax
SELECT_AX	SELECT_AX	ε_axiom	SELECT_AX	select_ax
BOOL_CASES_AX	BOOL_CASES_AX	bool_cases_axiom	BOOL_CASES_AX$^+$	bool_cases_thm$^+$
INFINITY_AX	INFINITY_AX	infinity_axiom	INFINITY_AX	infinity_ax
T_DEF	T_DEF	t_def	T_DEF	true_def
F_DEF	F_DEF	f_def	F_DEF	false_def
AND_DEF	AND_DEF	$\wedge$_def	{AND_DEF}	conj_def
-	-	-	IMP_DEF	-
OR_DEF	OR_DEF	$\vee$_def	OR_DEF	disj_def
NOT_DEF	NOT_DEF	$\neg$_def	NOT_DEF	not_def
FORALL_DEF	FORALL_DEF	$\forall$_def	FORALL_DEF	forall_def
EXISTS_DEF	EXISTS_DEF	$\exists$_def	EXISTS_THM$^+$	exists_def
{UEXISTS_DEF}	{UEXISTS_DEF}	$\exists_1$_def	{UEXISTS_DEF}	uexists_def

Table 1: The type constants, some of the constants and some of the theorems (including all the axioms) of the theory platform. Derived theorems in a given system are marked with $^+$.

two depending on the system), whereas numerals in hol90 and ProofPower HOL form an infinite family of constants. However, beyond the definition of a set of basic numeral arithmetic evaluation inference rules, these differences do not surface in practice in the implementations of the systems. Thus we have abstracted away from the theory platform the detail of how numerals are defined.

3.3 API Components

The API components form the ML interface for programming primary functionality. There are approximately 475 components, mainly consisting of ML function and constant values, but also seven datatypes and three exceptions. Three configuration values are also provided, that hold the HOL system name and version and the Common HOL Platform version. In each conformant system, the API is provided as an ML module interface file, with components given the same ordering to aid comparison between systems.

Note that table components that have ML infix fixity in a given system are written in parentheses.

3.3.1 Functional Programming Library

There are around 100 functional programming library components (see Table 2 for a selection).

hol90	HOL4	ProofPower	HOL Light	HOL Zero
curry	curry	curry	curry	curry
uncurry	uncurry	uncurry	uncurry	uncurry
C	C	switch	C	swap_arg
I	I	I	I	id_fn
K	K	K	K	con_fn
W	W	-	W	dbl_arg
(o)	(o)	(o)	(o)	(<*)
(##)	(##)	(**)	(F_F)	pair_apply
map	map	map	map	map
map2	map2	-	map2	bimap
{funpow}	{funpow}	fun_pow	{funpow}	funpow
itlist	itlist	fold	itlist	foldr
rev_itlist	rev_itlist	revfold	rev_itlist	foldl
end_itlist	end_itlist	-	end_itlist	foldr1
-	-	-	-	foldl1

Table 2: Some of the functional programming library API components.

Included are many basic operations on ML pairs, lists and strings, such as selecting the first element of a pair, reversing the order of elements in a list, or turning an integer into a string. Association lists are also supported. Also included are various classic functional programming meta operations, e.g. for applying a function to each element in a set, or folding up a list into a single element by repeated application of a binary operator. There is also a collection of set operations on lists, such as set membership and set union, under either equality comparison or a supplied equivalence relation.

For coherence, we fill out the gaps that exist in the various legacy systems' libraries. For example, all kinds of folding operators and their inverses, unfolding operators, are provided, and all set operations are provided for both under equality and a supplied equivalence relation.

Three kinds of standard exception are catered for: normal failure, catastrophic failure and "local failure" (used for control flow within a function). The API underspecifies the form of the exception arguments and the textual content of error messages

Note that there is some variation in the behaviour of some library functions between systems. For example, `funpow`, which iterates a function application for the number of times specified by a supplied integer, does not fail in hol90, HOL4 or HOL Light if the integer is negative. Generally, platform functions are specified to fail if supplied with invalid arguments, and the platform version of `funpow` fails if its supplied integer is negative, as is done in ProofPower HOL and HOL Zero.

3.3.2 Type, Term and Theorem Utilities

Around 150 HOL type, term and theorem manipulation utilities are provided (see Table 3 for a selection).

The bulk of these utilities are syntax functions for HOL types or terms, for constructing, destructing and testing for a given syntactic category. Two levels of syntactic category are supported for both types and terms. Firstly, there are the primitive syntactic categories, namely the type variables and type constant applications for types, and variables, constants, function applications and lambda abstractions for terms. These are very widely used throughout the HOL implementations. Secondly, there are the basic syntactic categories associated with the type constants and constants of predicate logic and lambda calculus that feature in the theory platform. Some of these are also used heavily throughout the HOL implementations, but we include support for all such syntactic categories in the API for coherence with the theory platform and the API inference rules.

hol90	HOL4	ProofPower	HOL Light	HOL Zero
type_of	type_of	type_of	type_of	type_of
type_vars_in_term	type_vars_in_term	{term_tyvars}	type_vars_in_term	term_tyvars
aconv	aconv	(∼=$)	aconv	alpha_eq
-	rename_bvar	-	{alpha}	rename_bvar
free_vars	free_vars	frees	frees	free_vars
free_varsl	free_varsl	-	freesl	list_free_vars
-	var_occurs	is_free_in	{vfree_in}	var_free_in
{free_in}	free_in	-	free_in	term_free_in
all_vars	-	-	variables	all_vars
all_varsl	-	-	-	list_all_vars
inst	{inst}	{inst}	{inst}	tyvar_inst
-	rename_bvar	-	{alpha}	rename_bvar
-	-	{var_subst}	vsubst	var_inst
{subst}	{subst}	subst	subst	subst

Table 3: Some of the term utility API components.

There are various ML bindings for HOL constants and base types featured in the theory platform, and for commonly used HOL type variables. Also included are utilities for destructing a theorem into its assumptions and conclusion parts, and for equality and alpha-equivalence comparison of theorems. There are also various type and term operations defined that are essential for defining an inference kernel. These include calculating the type of a term, listing the type variables of a type, testing for the alpha equivalence of two terms, and performing variable and type variable instantiation.

The platform utilities for HOL terms are generally specified to work modulo alpha equivalence in their arguments. This was decided because different systems generate bound variable names differently when avoiding variable capture in type variable and variable instantiation, and so this measure makes the API functions more robust when ported. An arbitrary bound variable name used in an operation in one system could otherwise cause the equivalent operation in another system to fail. Note that hol90's `free_in`, which tests for one term occurring free in another, does not work modulo alpha equivalence, and so does not conform to the platform.

Note that there are various subtle differences between different systems' utilities that can trip up casually ported code. Examples include ProofPower HOL's `mk_const` constructor, which does not test that a constructed constant is well-formed, and hol90's and HOL4's `dest_imp` and `is_imp`, which work for logical negation as well as implication (although HOL4 has `dest_imp_only` and `is_imp_only` for implication only). The API chooses more conventional behaviour.

3.3.3 Theory Extension and Listing Commands

Around 40 theory extension and querying functions are provided. This includes primitive theory extension commands for type declaration, term declaration, constant definition, constant specification and type constant definition. On top of these, there are a few basic derived theory extension commands, for example the command to define a function constant using a universal quantifier for the function arguments instead of a lambda abstraction. Most systems have more sophisticated extension commands, but these are excluded from the platform because there is much variation in their capability between systems.

Each system also provides querying commands to access information about the theory extensions that have been made, although HOL Light omits support for querying about primitive type constant definitions. Such commands are essential for the approach for proof auditing advocated in [2], and a complete set features in the API.

3.3.4 Inference Rules

Around 100 basic inference rules are provided by the API (see Table 4 for a selection).

It is sufficient for the platform inference rules to include just a kernel of primitive rules[1] that suffice, when coupled with the axiom and definition theorems in the theory platform, to implement the HOL deductive system. Given our choice of theory platform, any of the systems' primitive inference rules would be sufficient. However, efficiency is also a consideration. If a primitive rule of a given system were missing from the API, it would have to be reimplemented in that system's outer platform in terms of the API inference rules, and which would in turn need to be implemented in terms of the system's primitives. An execution of such a recreated primitive could require 10 pre-platform rule applications or more, resulting in an unacceptable performance penalty. Thus we choose to include the union of primitive rules from each system in the platform (with the exception of one HOL Light primitive explained below). This principle qualifies around 35 rules for inclusion in the platform. Note that each system except HOL Zero has primitive rules that are derive able in terms of other primitives, but are included to improve the system's performance, which explains why the union includes as many as 35.

Also included are around 15 other inference rules at roughly the same level as the union of the primitive inference rules, including the equality symmetry rule and the cut rule, for using the conclusion of one theorem to eliminate an assumption in another. A further 25 rules are included for performing equality congruence over certain operators, in addition to the two that are present as a result of being primitive inference rules. For coherence, these fill out the patchy provision in existing HOL systems with full coverage for the HOL operators supported by the API syntax functions.

In addition, for natural arithmetic expressions there are conversions provided for performing evaluation of operators applied to numeral arguments for each of the 13 natural arithmetic operators featured in the theory platform. This is sufficient to provide complete coverage of the primitive natural numeral arithmetic inference rules provided by hol90 and ProofPower HOL (which represent numerals as constants). This allows the platform to keep abstract the underlying representation of numerals.

It is vital that the API specifies precise behaviour for each of its inference rules. There is a degree of variation in the behaviour of various rules between systems. We outline here some ways in which the platform promotes robustness in the details of the behaviour it specifies for its inference rules.

As with the API's term utilities, its inference rules also work modulo alpha equivalence, for the same reasons. Note that the successful execution of HOL Light's `BETA` rule (not to be confused with its `BETA_CONV` rule) can fail depending on the name used for a bound variable in one of its arguments, and because of this it is excluded from the API, despite being a primitive of HOL Light. Fortunately, the consequences on performance in HOL Light are minimal because `BETA` can be implemented purely in terms of `BETA_CONV`, which is in the API.

It was also decided that API inference rules should not depend on the presence of assumptions in their theorem arguments, also to help robustness. It is harmless for a rule to remove an assumption if it can, and this should not result in failure in rules composed with it. So, for example, the rule for discharging an assumption matching a supplied term should not fail if the assumption is not present in the theorem argument. Note that ProofPower's classical contradiction rule `c_contr_rule` breaks this principle, but other systems' equivalents do not.

There are also various other differences in behaviour between seemingly equivalent rules in different HOL systems. One particularly extreme case is the rule for instantiating type variables, called `INST` in hol90, HOL4 and HOL Light, which is a primitive of every HOL system. In hol90, only type variables in the conclusion are instantiated. In HOL Light and HOL4, non-variable types in the instantiation list

[1] In the paper, we occasionally abbreviate the term *inference rule* to *rule*.

hol90	HOL4	ProofPower	HOL Light	HOL Zero
ASSUME*	ASSUME*	asm_rule*	ASSUME*	assume_rule*
BETA_CONV*	BETA_CONV*	simple_β_conv*	BETA_CONV	beta_conv*
CCONTR*	CCONTR*	{c_contr_rule}	CCONTR	ccontr_rule
CHOOSE*	CHOOSE*	simple_$\exists$_elim	CHOOSE	choose_rule
CONJ*	CONJ*	$\wedge$_intro	CONJ	conj_rule
CONJUNCT1*	CONJUNCT1*	$\wedge$_left_elim	CONJUNCT1	conjunct1_rule
CONJUNCT2*	CONJUNCT2*	$\wedge$_right_elim	CONJUNCT2	conjunct2_rule
CONTR*	CONTR	contr_rule	CONTR	contr_rule
-	-	-	DEDUCT_ANTISYM_RULE*	deduct_anitsym_rule
DISCH*	DISCH*	$\Rightarrow$_intro*	DISCH	disch_rule*
DISJ1*	DISJ1*	$\vee$_right_intro	DISJ1	disj1_rule
DISJ2*	DISJ2*	$\vee$_left_intro	DISJ2	disj2_rule
DISJ_CASES*	DISJ_CASES*	$\vee$_elim	DISJ_CASES	disj_cases_rule

Table 4: Some of the inference rule API components. Primitive rules in a given system are marked with *.

argument do not cause failure. And in ProofPower HOL, any free variables that would otherwise become equal as a result of the instantiation are renamed. None of these idiosyncrasies exist in the API version.

3.3.5 Parsing and Pretty Printing

Around 20 functions supporting parsing and pretty printing are provided in the API. This includes functions for parsing strings into HOL types and terms, and printers for types, terms and theorems. There is also support for setting the fixity of HOL functions and type operators. The fixities supported exceed what is provided by hol90, ProofPower HOL and HOL Light, but do not extend to the full range of fixities supported by HOL4. There are plans to extend the platform to support all of HOL4's fixities.

4 Implementation

4.1 Architecture

For a legacy system to conform to an API, its source code must be adapted so that every component of the API is implemented in the system. For the Common HOL API, we use a software architecture for adapting legacy HOL systems that is designed with the three goals of minimising implementation effort, enabling API-level virtualisation, and facilitating the demonstration that the adapted system exhibits precisely the same behaviour as the legacy system.

To achieve this, we choose an appropriate point in the build of the legacy system that corresponds to the level of the API (the *platform level*), and insert an ML module for the API components (the *platform module*) at this point. All legacy source code occurs either before or after the platform level (respectively called the *pre-platform* and *post-platform* code) and stays exactly the same. Keeping the pre- and post-platform code the same makes it easier to argue that the system's behaviour has not been altered.

In the platform module, we define the API in terms of pre-platform functionality. Any API components not precisely implemented as a pre-platform component must be implemented here. This includes components missing from the legacy system, or with imprecisely corresponding equivalents in the pre-platform code or that are implemented as post-platform code. For any implemented as post-platform code, the full tree of post-platform code used to define it can be shifted into the platform module, or, if this is too big, then a more succinct version can be implemented specially for the platform. The code for

post-platform API components can then be deleted from its original position in the source code (thus the post-platform code remains the same except for deleted code that occurs in the platform module).

In our architecture, all post-platform code implementing primary functionality is implemented in terms of the API. This enables the API to act as a virtualisation layer through which all primary functionality is executed. This virtualisation layer can then be used for recording proofs as they are executed, before exporting them to proof objects. In order to achieve this and keep the post-platform code the same, we must somehow have a way of referring to pre-platform code that is used by post-platform code but is not in the API. We do this by implementing a module immediately after the platform module in the build that re-implements all such pre-platform code in terms of the platform, overwriting the pre-platform code. We call this the *outer platform* module.

In arguing that the system's behaviour has not altered in the API-adjusted version of the system, we must justify why any reimplementation of post-platform code in the platform module, and any reimplementation of pre-platform code in the outer platform module, preserves functionality.

Given that the API components correspond to classic basic components of a HOL system that tend to be implemented towards the start of the build of the system, finding an appropriate insertion point for the platform level tends to be fairly straightforward. It is to be found after the definition of the HOL type and term datatypes and basic utilities for manipulating them, the inference kernel, the initial theory and the parser and pretty printer. It is typically before the derived inference rules for predicate logic and the theory for pairs and natural numbers, which would need to be moved to or recreated in the platform module.

4.2 Adapting HOL Light

We now describe how we adapted HOL Light SVN release 197 to conform to the platform. The reader may find it instructive to download the adapted system [18].

The platform level in the HOL Light build file was chosen between the source files `parser.ml` and `equal.ml`. About 1,000 lines of post-platform code implementing platform components were moved into the platform module. Much of this was derived inference rules implemented using lemmas proved using HOL Light's automated proof facilities. Instead of recreating these facilities inside the platform module, we employed Common HOL proof porting to export the proofs of these lemmas as proof objects, which were then hand-translated into a total of around 400 lines of forwards style proof script in the platform module. An alternative approach was used to recreate the 13 evaluation rules for natural numeral arithmetic, whose implementation in `calc_num.ml` involves lemmas proved in hundreds of lines of proof script. Instead of exporting proof objects for these lemmas, the inference rules were given a completely different implementation in the platform module, ported from HOL Zero in about 800 lines.

About 1,000 lines of code were required to fill out platform components missing from HOL Light. For those components with an approximate equivalent already in HOL Light, the existing component was used in the implementation of the platform variant (e.g. see Figure 1), to ensure that the platform variant had roughly the same performance as the original. Those components with no approximate HOL Light version were ported from HOL Zero. In total, the components ported from HOL Zero required about 1,350 lines of supporting source code to be ported from HOL Zero, mainly involving forwards proof to prove lemmas. The platform module interface is written in about 500 lines of code.

For the outer platform, primitive inference rules and theory commands that do not correspond to platform components must be precisely recreated in terms of the platform. In HOL Light, this involves the `INST_TYPE` and `BETA` rules and all the theory commands. Also, non-platform theorems used to define platform theory needed to be recreated. In total, the outer platform required around 800 lines of code.

```
let INST_TYPE1 theta th =
  let () = if (forall (is_vartype o snd) theta)
             then failwith "INST_TYPE: Non-type-variable in instantiation domain" in
  INST_TYPE theta th;;
```

Figure 1: Using HOL Light's original `INST_TYPE` in the definition of the platform variant.

Overall, the platform and outer platform modules involved around 6,000 lines of source code, including the platform module interface. This took around two weeks of effort to create. The code was mostly systematically produced, being either moved from other parts of HOL Light, ported from HOL Zero, translated from proof object files, or simply a listing of platform components. The only code requiring creative thought was in the platform module variants of components with approximate equivalents already in HOL Light, and in much of the outer platform, totalling to around 1,000 lines.

5 Use Cases

In this section, we report on two use cases for the Common HOL Platform in assisting manual ports of source code between platform-adapted HOL systems. In both cases, the port was from HOL Light to HOL Zero. This is on the easy end of the difficulty spectrum in inter-HOL-system code porting, because both systems are implemented in the same dialect of ML, i.e. OCaml, and because the target system, HOL Zero, is almost a blank canvas with very little post-platform code to consider. Other HOL systems have considerable post-platform code, and porting should attempt to reuse any pre-existing code if it is straightforward to do so, to avoid creating an almost duplicate stack of supporting functionality in the target system. However, both ports described here would still be difficult without the support of the platform, and so the use cases provide useful insight.

5.1 Legacy Code Port: HOL Light Rewriting Mechanism to HOL Zero

In our first use case, we ported HOL Light's entire rewriting apparatus to HOL Zero. This is defined relatively early on in HOL Light's post-platform code, but provides vital functionality that is used throughout the rest of the system, and goes far beyond what HOL Zero is capable of in terms of proof automation. It is implemented in 360 lines of code, in the HOL Light source file `simp.ml`, and relies on 60 lines of code defining discrimination nets, and a further 300 lines of post-platform code defining supporting functionality such as conversion combinators. Thus there was a total of 720 lines to port, but this would probably be less if porting to another HOL system because it would already support conversion combinators. See Figures 2 and 3 for a sample of 32 lines from the port.

The manual port was carried out in about 2 hours 30 minutes of effort. Note that this time does not include approximately 30 minutes of effort required to extract out the 360 lines of HOL Light supporting code prior to the port. The porting itself involved systematically looking up HOL Zero equivalents of HOL Light platform functions, and renaming accordingly. HOL Light's uppercase names, that don't conform to normal OCaml lexical syntax, also needed to be converted to lowercase names. Instantiation lists, which have old-to-new ordering in HOL Zero but new-for-old ordering in HOL Light, needed to be switched around. The datatype constructors for types and terms, which are visible outside their defining module in HOL Light but not in HOL Zero, required some pattern matches to be replaced with abstract destructors and if-expressions. The function `term_match` name-clashed with a pre-existing HOL Zero function, and so was renamed to `hl_term_match`.

```
let mk_rewrites =
  let IMP_CONJ_CONV = REWR_CONV(ITAUT `p ==> q ==> r <=> p /\ q ==> r`)
  and IMP_EXISTS_RULE =
    let cnv = REWR_CONV(ITAUT `(!x. P x ==> Q) <=> (?x. P x) ==> Q`) in
    fun v th -> CONV_RULE cnv (GEN v th) in
  let collect_condition oldhyps th =
    let conds = subtract (hyp th) oldhyps in
    if conds = [] then th else
    let jth = itlist DISCH conds th in
    let kth = CONV_RULE (REPEATC IMP_CONJ_CONV) jth in
    let cond,eqn = dest_imp(concl kth) in
    let fvs = subtract (subtract (frees cond) (frees eqn)) (freesl oldhyps) in
    itlist IMP_EXISTS_RULE fvs kth in
  let rec split_rewrites oldhyps cf th sofar =
    let tm = concl th in
    if is_forall tm then
      split_rewrites oldhyps cf (SPEC_ALL th) sofar
    else if is_conj tm then
      split_rewrites oldhyps cf (CONJUNCT1 th)
        (split_rewrites oldhyps cf (CONJUNCT2 th) sofar)
    else if is_imp tm & cf then
      split_rewrites oldhyps cf (UNDISCH th) sofar
    else if is_eq tm then
      (if cf then collect_condition oldhyps th else th)::sofar
    else if is_neg tm then
      let ths = split_rewrites oldhyps cf (EQF_INTRO th) sofar in
      if is_eq (rand tm)
      then split_rewrites oldhyps cf (EQF_INTRO (GSYM th)) ths
      else ths
    else
      split_rewrites oldhyps cf (EQT_INTRO th) sofar in
  fun cf th sofar -> split_rewrites (hyp th) cf th sofar;;
```

Figure 2: A sample of legacy source code from HOL Light's `simp.ml`.

HOL Light non-conformant versions of platform functions, such as its `variant` function, required special attention. Unlike the platform equivalent, this function does not fail if its avoidance list contains non-variables, and so the code was adapted to either filter them out or check that non-variables are not possible from program context. Other complications included two uses of HOL Light's intuitionistic tautology prover, `ITAUT`. It was decided to keep this function outside the scope of the port, despite it being used to prove two lemmas, to reduce the amount of supporting code. For the HOL Zero version, one of the lemmas already existed in HOL Zero's small library of predicate logic theorems, and the other was proved in 10 minutes in a 16-line proof using HOL Zero's forward inference rules.

After the port was completed, it was tested on various rewriting examples, and one error was found. This took 45 minutes of debugging to track down and correct, and was due to a quirk in the failure exception returned by HOL Light's `rev_assoc` function, which has error message text `"find"` (instead of `"rev_assoc"`). This particular error message was explicitly trapped in the HOL Light code, but naively porting this to HOL Zero didn't work because its equivalent function, `inv_assoc`, uses error message text `"inv_assoc"`. As explained in Section 3.3.1, this aspect of porting is not catered for by the platform, and must be done manually.

```
let mk_rewrites =
  let imp_conj_conv = rewr_conv imp_imp_thm
  and imp_exists_rule =
    let cnv = rewr_conv imp_exists_rule_thm in
    fun v th -> conv_rule cnv (gen_rule v th) in
  let collect_condition oldhyps th =
    let conds = subtract (asms th) oldhyps in
    if conds = [] then th else
    let jth = foldr disch_rule conds th in
    let kth = conv_rule (repeatc imp_conj_conv) jth in
    let cond,eqn = dest_imp(concl kth) in
    let fvs = subtract (subtract (free_vars cond) (free_vars eqn))
                       (list_free_vars oldhyps) in
    foldr imp_exists_rule fvs kth in
  let rec split_rewrites oldhyps cf th sofar =
    let tm = concl th in
    if is_forall tm then
      split_rewrites oldhyps cf (spec_all_rule th) sofar
    else if is_conj tm then
      split_rewrites oldhyps cf (conjunct1_rule th)
        (split_rewrites oldhyps cf (conjunct2_rule th) sofar)
    else if is_imp tm & cf then
      split_rewrites oldhyps cf (undisch_rule th) sofar
    else if is_eq tm then
      (if cf then collect_condition oldhyps th else th)::sofar
    else if is_not tm then
      let ths = split_rewrites oldhyps cf (eqf_intro_rule th) sofar in
      if is_eq (rand tm)
      then split_rewrites oldhyps cf (eqf_intro_rule (gsym_rule th)) ths
      else ths
    else
      split_rewrites oldhyps cf (eqt_intro_rule th) sofar in
  fun cf th sofar -> split_rewrites (asms th) cf th sofar;;
```

Figure 3: The translation into HOL Zero of the legacy code sample from `simp.ml`.

5.2 New Code Port: HOL Light Proof Importer to HOL Zero

In the second use case, we used the platform to port HOL Light's importer for Common HOL proof objects. This was a fundamentally easier exercise because the proof importer is written specifically in terms of the API, and because Common HOL proof porting works at the level of platform inference rules itself. The proof importer is implemented in 2,200 lines of code.

It took about 1 hour 15 minutes to perform the porting. Despite the source code being three times longer than in the legacy code port, it took only half the time. The easier nature of the task meant that everything went smoothly first time. The effort consisted almost entirely of systematically applying search-and-replace to replace HOL Light platform function names with HOL Zero equivalents and carrying out manual adjustments for functions that take their arguments differently in the different systems.

The resulting source code was tested by importing into HOL Zero the text formalisation part of the Flyspeck project, as part of a partial audit of the project as described in [2]. This involved the tens of millions of platform-level inference rule steps. The import into HOL Zero worked first time, suggesting the code was ported correctly.

6 Conclusions

In defining a standard for basic theory and programming components, the Common HOL Platform is attempting to lay the foundation for much better portability between HOL systems, both in terms of porting proofs and porting source code. The feasibility of large scale proof porting has already been established by others, but arguably there is scope for doing better still, given a better foundation. However, the feasibility of quick and reliable source code porting has not been explored until now.

In this paper, we have given an overview of the platform's components and explained the reasons behind some of the careful design decisions made. We have also demonstrated using the platform in two use cases of manually porting source code from HOL Light to HOL Zero, one for legacy code and one for new code written specially for the platform. In both cases, several hundred lines of code were successfully and reliably ported within a few hours. Much of the effort normally involved in a manual port is removed, because almost all that needs to be considered is functionality implemented above the platform level. Finding corresponding low-level components in the two systems, and the subtle ways in which they can differ, has already been taken care of by the platform. As far as we are aware, this represents a leap in the productivity of source code porting between HOL systems, even when accounting for it being less challenging than the general porting case due to both systems being implemented in the same dialect of ML and due to HOL Zero effectively being a blank canvas.

It would be interesting to see how far HOL source code porting could be pushed. Certainly it is feasible to port more challenging parts of HOL Light to HOL Zero. Obvious candidates are the subgoal package, the intuitionistic tautology checker and the powerful `MESON_TAC`. Implementing the latest version of the platform for hol90, HOL4 and ProofPower HOL, and porting to these systems is another challenge worth pursuing. The platform has already been designed with these systems in mind, and it would at least enable Common HOL proof exporters and importers to be quickly ported to these systems.

One insight that comes from looking at code from the various HOL systems is how much the subgoal package is used in the implementation of other parts of HOL systems, suggesting that it should be part of the API. This should be a fairly easy extension to make, since beyond the implementation of an initial few tactics, code using it appears to operate at the abstract level using tacticals, rather than use the inner workings that differ between HOL systems. Another change worth making is to update the platform for the reform to primitive theory extension currently underway in various HOL systems [4]. And finally, catering for Isabelle/HOL must be a long term priority. This would probably require a significant overhaul of the platform to fit with such a different system, but if done well it would pay dividends to have good portability between the widest used HOL system and the rest of the family.

The systematic manner in which the porting can be carried out lends itself to automation, or at least to partial automation. The most difficult to automate is probably the intelligent use of the target system's legacy supporting code to avoid the ugly situation of creating two parallel stacks of code implementing effectively the same thing. Thus partial automation looks a more realistic prospect. We believe there are no fundamental difficulties in automatically porting between ML dialects, because the subsets of ML that tend to be used in the implementation of HOL systems are trivially corresponding between OCaml and SML. So we see there being good prospects for reducing further the time taken to reliably port source code, even in more challenging cases.

References

[1] M. Adams & P. Clayton (2005): *ClawZ: Cost-Effective Formal Verification for Control Systems*. In: *Proceedings of the 7th International Conference on Formal Methods and Software Engineering, Lecture Notes*

in Computer Science 3785, Springer, pp. 465–479, doi:10.1007/11576280_32.

[2] M. Adams (2015): *Proof Auditing Formalised Mathematics*. Available at `http://www.proof-technologies.com/flyspeck/qed_paper.pdf`. Accepted for publication in the Journal of Formalized Reasoning.

[3] R. Arthan & R. Jones (2005): *Z in HOL in ProofPower*. In Issue 2005-1 of the British Computer Society Specialist Group Newsletter on Formal Aspects of Computing Science.

[4] R. Arthan (2014): *HOL Constant Definition Done Right*. In: *Proceedings of the 5th International Conference on Interactive Theorem Proving*, *Lecture Notes in Computer Science* 8558, Springer, pp. 531–536, doi:10.1007/978-3-319-08970-6_34.

[5] M. Gordon & T. Melham (1993): *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

[6] M. Gordon, R. Milner & C. Wadsworth (1979): *Edinburgh LCF: A Mechanised Logic of Computation*. *Lecture Notes in Computer Science* 78, Springer, doi:10.1007/3-540-09724-4.

[7] T. Hales et al. (2015): *A Formal Proof of the Kepler Conjecture*. Preprint available at `arxiv.org`. ArXiv:1501.02155v1 [math.MG].

[8] J. Harrison (2009): *HOL Light: An Overview*. In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science* 5674, Springer, pp. 60–66, doi:10.1007/978-3-642-03359-9_4.

[9] J. Hurd (2011): *The OpenTheory Standard Theory Library*. In: *Proceedings of the Third International Symposium on NASA Formal Methods*, *Lecture Notes in Computer Science* 6617, Springer, pp. 177–191, doi:10.1007/978-3-642-20398-5_14.

[10] C. Kaliszyk & A. Krauss (2013): *Scalable LCF-Style Proof Translation*. In: *Proceedings of the 4th International Conference on Interactive Theorem Proving*, *Lecture Notes in Computer Science* 7998, Springer, pp. 51–66, doi:10.1007/978-3-642-39634-2_7.

[11] G. Klein et al. (2009): *seL4: Formal Verification of an OS Kernel*. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ACM, pp. 207–220, doi:10.1145/1629575.1629596.

[12] T. Nipkow, L. Paulson & M. Wenzel (2002): *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9.

[13] S. Obua & S. Skalberg (2006): *Importing HOL into Isabelle/HOL*. In: *Proceedings of the Third International Joint Conference on Automated Reasoning*, *Lecture Notes in Computer Science* 4130, Springer, pp. 298–302, doi:10.1007/11814771_27.

[14] L. Paulson (1987): *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press, doi:10.1017/CBO9780511526602.

[15] K. Slind (1991): *An Implementation of Higher Order Logic*. Technical Report 91-419-03, Computer Science Department, University of Calgary.

[16] K. Slind & M. Norrish (2008): *A Brief Overview of HOL4*. In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science* 5170, Springer, pp. 28–32, doi:10.1007/978-3-540-71067-7_6.

[17] M. Wenzel, L. Paulson & T. Nipkow (2008): *The Isabelle Framework*. In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science* 5170, Springer, pp. 33–38, doi:10.1007/978-3-540-71067-7_7.

[18] *HOL Light adaptation for Common HOL*. Available at `http://www.proof-technologies.com/commonhol/commonhol-0.5-hl-svn197.tgz`.

[19] *HOL Zero homepage*. Available at `http://www.proof-technologies.com/holzero/`.

Checking Zenon Modulo Proofs in Dedukti *

Raphaël Cauderlier Pierre Halmagrand
Cnam - Inria
Paris, France
raphael.cauderlier@inria.fr pierre.halmagrand@inria.fr

Dedukti has been proposed as a universal proof checker. It is a logical framework based on the $\lambda\Pi$-calculus modulo that is used as a backend to verify proofs coming from theorem provers, especially those implementing some form of rewriting. We present a shallow embedding into Dedukti of proofs produced by Zenon Modulo, an extension of the tableau-based first-order theorem prover Zenon to deduction modulo and typing. Zenon Modulo is applied to the verification of programs in both academic and industrial projects. The purpose of our embedding is to increase the confidence in automatically generated proofs by separating untrusted proof search from trusted proof verification.

1 Introduction

Program verification using deductive methods has become a valued technique among formal methods, with practical applications in industry. It guarantees a high level of confidence regarding the correctness of the developed software with respect to its specification. This certification process is generally based on the verification of a set of proof obligations, generated by deductive verification tools. Unfortunately, the number of proof obligations generated may be very high. To address this issue, deductive verification tools often rely on automated deduction tools such as first-order Automated Theorem Provers (ATP) or Satisfiability Modulo Theories solvers (SMT) to automatically discharge a large number of those proof obligations. For instance, Boogie is distributed with the SMT Z3 [4] and the Why3 platform with Alt-Ergo [16]. After decades of constant work, ATP and SMT have reached a high level of efficiency and now discharge more proof obligations than ever. At the end, many of these program verification tools use their corresponding ATP or SMT as oracles. The main concern here is the level of confidence users give to them. These programs are generally large software, consisting of dozens of thousands of lines of code, and using some elaborate heuristics, with some ad hoc proof traces at best, and with a simple "yes or no" binary answer at worst.

A solution, stated by Barendregt and Barendsen [3] and pursued by Miller [19] among others, relies on the concept of proof certificates. ATP and SMT should be seen as proof-certificate generators. The final "yes or no" answer is therefore left to an external proof checker. In addition, Barendregt and Barendsen proposed that proof checkers should satisfy two principles called the De Bruijn criterion and the Poincaré principle. The former states that proof checkers have to be built on a light and auditable kernel. The latter recommends that they distinguish reasoning and computing and that it should not be necessary to record pure computational steps.

Relying on an external proof checker to verify proofs strongly increases the trust we give them, but it also provides a common framework to express proofs. A profit made by using this common framework is the possibility to share proofs coming from different theorem provers, relying on different proof systems. But nothing comes for free, and using the same proof checker does not guarantee in general that we can

*This work has received funding from the BWare project (ANR-12-INSE-0010) funded by the INS programme of the French National Research Agency (ANR).

C.Kaliszyk and A. Paskevich (Eds.): 4th Workshop
on Proof eXchange for Theorem Proving (PxTP'15).
EPTCS 186, 2015, pp. 57–73, doi:10.4204/EPTCS.186.7

share proofs because formulæ and proofs can be translated in incompatible ways. Translation of proofs must rely on a shallow embedding in the sense proposed by Burel [10]: it reuses the features of the target language. It does not introduce new axioms and constants for logical symbols and inference rules. Connectives and binders of the underlying logic of ATP are translated to their corresponding connectives and binders in the target language. In addition, a shallow embedding preserves the computational behavior of the original ATP and the underlying type system of the logic.

In this paper, we present a shallow embedding of Zenon Modulo proofs into the proof checker Dedukti, consisting of an encoding of a typed classical sequent calculus modulo into the $\lambda\Pi$-calculus modulo ($\lambda\Pi^{\equiv}$ for short). Zenon Modulo [13] is an extension to deduction modulo [15] of the first-order tableau-based ATP Zenon [8]. It has also been extended to support ML polymorphism by implementing the TFF1 format [5]. Dedukti [7] is a proof checker that implements $\lambda\Pi^{\equiv}$, a proof language that has been proposed as a proof standard for proof checking and interoperability. This embedding is used to certify proofs in two different projects: FoCaLiZe [17], a programming environment to develop certified programs and based on a functional programming language with object-oriented features, and BWare [14], an industrial research project that aims at providing a framework for the automated verification of proof obligations coming from the B method [1]. The main benefit of Zenon Modulo and Dedukti relies on deduction modulo. Deduction modulo is an extension of first-order logic that allows reasoning modulo a congruence relation over propositions. It is well suited for automated theorem proving when dealing with theories since it turns axioms into rewrite rules. Using rewrite rules during proof search instead of reasoning on axioms lets provers focus on the challenging part of proofs, speeds up the tool and reduces the size of final proof trees [12].

Zenon was designed to support FoCaLiZe as its dedicated deductive tool and to generate proof certificates for Coq. Extension to deduction modulo constrains us to use a proof checker that can easily reason modulo rewriting. Dedukti is a good candidate to meet this specification. A previous embedding of Zenon Modulo proofs into Dedukti, based on a $\neg\neg$ translation [13], was implemented as a tool to translate classical proofs into constructive ones. This tool has the benefit to be shallower since it does not need add the excluded middle as an axiom into the target logic defined in Dedukti, but in return this transformation may be very time-consuming [12] and was not scalable to large proofs like those produced in BWare. The closest related work is the shallow embedding of resolution and superposition proofs into Dedukti proposed by Burel [10] and implemented in iProver Modulo [9]. Our embedding is close enough to easily share proofs of Zenon Modulo and iProver Modulo in Dedukti, at least for the subset of untyped formulæ.

The first contribution presented in this paper consists in the encoding into $\lambda\Pi^{\equiv}$ of typed deduction modulo and a set of translation functions into $\lambda\Pi^{\equiv}$ of theories expressed in this logic. Another contribution of this paper is the extension to deduction modulo and types of the sequent-like proof system LLproof which is the output format of Zenon Modulo proofs. The latter contribution is the embedding of this proof system into $\lambda\Pi^{\equiv}$ and the associated translation function for proofs coming from this system.

This paper is organized as follows: in Sec. 2, we introduce typed deduction modulo; in Sec. 3, we present $\lambda\Pi^{\equiv}$, its proof checker Dedukti, and a canonical encoding of typed deduction modulo in $\lambda\Pi^{\equiv}$; Sec. 4 introduces the ATP Zenon Modulo, the proof system LLproof used by Zenon Modulo to output proofs; and the translation scheme implemented as the new output of Zenon Modulo; finally, in Sec. 5, we present some examples and results to assess our implementation.

2 Typed Deduction Modulo

The Poincaré principle, as stated by Barendregt and Barendsen [3], makes a distinction between deduction and computation. Deduction may be defined using a set of inference rules and axioms, while computation consists mainly in simplification and unfolding of definitions. When dealing with axiomatic theories, keeping all axioms on the deduction side leads to inefficient proof search since the proof-search space grows with the theory. For instance, proving the following statement:

$$\mathsf{fst}(a,a) = \mathsf{snd}(a,a)$$

where a is a constant, and fst and snd are defined by:

$$\forall x,y.\ \mathsf{fst}(x,y) = x \qquad\qquad \forall x,y.\ \mathsf{snd}(x,y) = y$$

and with the reflexivity axiom:

$$\forall x.\ x = x$$

using a usual automated theorem proving method such as tableau, will generate some useless boilerplate proof steps, whereas a simple unfolding of definitions of fst and snd directly leads to the formula $a = a$.

Deduction modulo was introduced by Dowek, Hardin and Kirchner [15] as a logical formalism to deal with axiomatic theories in automated theorem proving. The proposed solution is to remove computational arguments from proofs by reasoning modulo a decidable congruence relation $\equiv$ on propositions. Such a congruence may be generated by a confluent and terminating system of rewrite rules (sometimes extended by equational axioms).

In our example, the two definitions may be replaced by the rewrite rules:

$$\mathsf{fst}(x,y) \longrightarrow x \qquad\qquad \mathsf{snd}(x,y) \longrightarrow y$$

And we obtain the following equivalence between propositions:

$$(\mathsf{fst}(a,a) = \mathsf{snd}(a,a)) \ \equiv\ (a = a)$$

Reasoning with several theories at the same time is often necessary in practice. For instance, in the BWare project, almost all proof obligations combine the theory of booleans, arithmetic and set theory. In this case, we have to introduce an expressive enough type system to ensure that an axiom about booleans, for instance $\forall x.\ x = \mathsf{true} \vee x = \mathsf{false}$, will not be used with a term that has another type. An input format for ATP called TFF1 [5] has been proposed recently by Blanchette and Paskevich to deal with first-order problems with polymorphic types. We propose to extend this format to deduction modulo.

We now introduce the notion of typed rewrite system, extending notations of Dowek et al. [15]. In the following, $\mathrm{FV}(t)$ stands for the set of free variables of t where t is either a TFF1 term or a TFF1 formula.

Definition (Typed Rewrite System)
A term rewrite rule is a pair of TFF1 terms l and r together with a TFF1 typing context Δ denoted by $l \longrightarrow_\Delta r$, where $\mathrm{FV}(r) \subseteq \mathrm{FV}(l) \subseteq \Delta$. It is well-typed in a theory $\mathscr{T}$ if l and r can be given the same type A in $\mathscr{T}$ using Δ to type free variables. A proposition rewrite rule is a pair of TFF1 formulæ l and r together with a typing context Δ denoted by $l \longrightarrow_\Delta r$, where l is an atomic formula and r is an arbitrary formula,

Types	τ	::=	α	(type variable)
		|	$T(\tau_1,\ldots,\tau_m)$	(type constructor)
Terms	e	::=	x	(term variable)
		|	$f(\tau_1,\ldots,\tau_m;e_1,\ldots,e_n)$	(function)
Formulæ	φ	::=	$\top \mid \bot$	(true, false)
		|	$\neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2$	(logical connectors)
		|	$e_1 =_\tau e_2$	(term equality)
		|	$P(\tau_1,\ldots,\tau_m;e_1,\ldots,e_n)$	(predicate)
		|	$\forall x:\tau.\ \varphi(x) \mid \exists x:\tau.\ \varphi(x)$	(term quantifiers)
		|	$\forall_{\mathsf{type}}\alpha : \mathsf{type}.\ \varphi(\alpha) \mid \exists_{\mathsf{type}}\alpha : \mathsf{type}.\ \varphi(\alpha)$	(type quantifiers)
Context	Δ	::=	$\emptyset$	(empty context)
		|	$\Delta, x:\tau$	(declaration)
Theory	$\mathscr{T}$	::=	$\emptyset$	(empty theory)
		|	$\mathscr{T}, T/m$	(m-ary type constructor declaration
		|	$\mathscr{T}, f : \Pi\vec{\alpha}.\ \vec{\tau} \to \tau$	(function declaration)
		|	$\mathscr{T}, P : \Pi\vec{\alpha}.\ \vec{\tau} \to o$	(predicate declaration)
		|	$\mathscr{T}, \text{name} : \varphi$	(axiom)
		|	$\mathscr{T}, l \longrightarrow_\Delta r$	(rewrite rule)

Figure 1: Syntax of TFF1$^{\equiv}$

and where $\mathrm{FV}(r) \subseteq \mathrm{FV}(l) \subseteq \Delta$. It is well-typed in a theory $\mathscr{T}$ if both l and r are well-formed formulæ in $\mathscr{T}$ using Δ to type free variables.

A typed rewrite system is a set $\mathscr{R}$ of proposition rewrite rules along with a set $\mathscr{E}$ of term rewrite rules. Given a rewrite system $\mathscr{R}\mathscr{E}$, the relation $=_{\mathscr{R}\mathscr{E}}$ denotes the congruence generated by $\mathscr{R}\mathscr{E}$. It is well-formed in a theory $\mathscr{T}$, if all its rewrite rules are well-typed in $\mathscr{T}$.

The notion of TFF1 theory can be extended with rewrite rules; we call the resulting logic TFF1$^{\equiv}$. Its syntax is given in Fig. 1.

3 Dedukti

The $\lambda\Pi$-calculus [2] is the simplest Pure Type System featuring dependent types. It is commonly used as a logical framework for encoding logics [18]. The $\lambda\Pi$-calculus modulo, presented in Fig. 2, is an extension of the $\lambda\Pi$-calculus with rewriting. The $\lambda\Pi$-calculus modulo (abbreviated as $\lambda\Pi^{\equiv}$) has successfully been used to encode many logical systems (Coq [6], HOL, iProver Modulo [10], FoCaLiZe) using shallow embeddings.

In $\lambda\Pi^{\equiv}$, conversion goes beyond simple β-equivalence since it is extended by a custom rewrite system. When this rewrite system is both strongly normalizing and confluent, each term gets a unique (up to α-conversion) normal form and both conversion and type-checking become decidable. Dedukti is an implementation of this decision procedure.

Burel [10] defines two encodings of deduction modulo in Dedukti: a deep encoding $|\varphi|$ in which logical connectives are simply declared as Dedukti constants and a shallow encoding $\|\varphi\| := \mathsf{prf}\ |\varphi|$ using a decoding function prf for translating connectives to their impredicative encodings. In Sec. 3.1 and Sec. 3.2, we extend these encodings to TFF1$^{\equiv}$.

Syntax

$$\begin{array}{lcl} s & ::= & \mathsf{Type} \mid \mathsf{Kind} \\ t & ::= & x \mid t\ t \mid \lambda x : t.t \mid \Pi x : t.t \mid s \\ \Delta & ::= & \emptyset \mid \Delta, x : t \\ \Gamma & ::= & \emptyset \mid \Gamma, x : t \mid \Gamma, t \hookrightarrow_\Delta t \end{array}$$

Well-formdness

$$\frac{}{\emptyset \vdash}\ (\text{Empty}) \qquad \frac{\Gamma \vdash \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash}\ (\text{Decl}) \qquad \frac{\Gamma,\Delta \vdash l : A \qquad \Gamma,\Delta \vdash A : \mathsf{Type} \qquad \Gamma,\Delta \vdash r : A \qquad FV(r) \subseteq FV(l) \subseteq \Delta}{\Gamma, l \hookrightarrow_\Delta r \vdash}\ (\text{Rew})$$

Typing

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}}\ (\text{Sort}) \qquad \frac{\Gamma \vdash \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\ (\text{Var})$$

$$\frac{\Gamma \vdash t_1 : \Pi x : A.B(x) \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\ t_2 : B(t_1)}\ (\text{App}) \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash t : B(x) \qquad \Gamma, x : A \vdash B(x) : s}{\Gamma \vdash \lambda x : A.t(x) : \Pi x : A.B(x)}\ (\text{Abs})$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B(x) : s}{\Gamma \vdash \Pi x : A.B(x) : s}\ (\text{Prod}) \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}\ (\text{Conv})$$

Figure 2: The $\lambda\Pi$-calculus modulo

3.1 Deep Embedding of Typed Deduction Modulo in Dedukti

In Fig. 3, for each symbol of our first-order typed logic, we declare its corresponding symbol into $\lambda\Pi^{\equiv}$. In $\lambda\Pi^{\equiv}$, types cannot be passed as arguments (no polymorphism) so we have to translate $\mathsf{TFF1}^{\equiv}$ types as Dedukti terms. The Dedukti type of translated $\mathsf{TFF1}^{\equiv}$ types is type and we can see an inhabitant of type as a Dedukti type thanks to the term function.

In Fig. 4, we define a direct translation of $\mathsf{TFF1}^{\equiv}$ in Dedukti. It is correct in the following sense:

- if the theory $\mathscr{T}$ is well-formed in $\mathsf{TFF1}^{\equiv}$, then $|\mathscr{T}| \vdash$.
- if τ is a well-formed $\mathsf{TFF1}^{\equiv}$ type in a theory $\mathscr{T}$, then $|\mathscr{T}| \vdash |\tau| : \mathsf{type}$.
- if t is a $\mathsf{TFF1}^{\equiv}$ term of type τ in a theory $\mathscr{T}$, then $|\mathscr{T}| \vdash |t| : \mathsf{term}\ |\tau|$.
- if φ is a well-formed $\mathsf{TFF1}^{\equiv}$ formula in a theory $\mathscr{T}$, then $|\mathscr{T}| \vdash |\varphi| : \mathsf{Prop}$.

3.2 From Deep to Shallow

Following Burel [10], we add rewrite rules defining the decoding function prf in Fig. 5 using the usual impredicative encoding of connectives. This transforms our deep encoding of $\mathsf{TFF1}^{\equiv}$ into a shallow encoding in which all connectives are defined by the built-in constructions of $\lambda\Pi^{\equiv}$.

This encoding is better suited for sharing proofs with other ATP because it is less sensible to small modifications of the logic. Any proof found, for example, by iProver Modulo is directly usable as an (untyped) proof in the shallow encoding.

Primitive Types

$$\mathsf{Prop} : \mathsf{Type} \qquad \mathsf{prf} : \mathsf{Prop} \to \mathsf{Type} \qquad \mathsf{type} : \mathsf{Type} \qquad \mathsf{term} : \mathsf{type} \to \mathsf{Type}$$

Primitive Connectives

$$\begin{array}{ll}
\top : \mathsf{Prop} & \bot : \mathsf{Prop} \\
\neg\text{-} : \mathsf{Prop} \to \mathsf{Prop} & \text{-} \wedge \text{-} : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\
\text{-} \vee \text{-} : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} & \text{-} \Rightarrow \text{-} : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\
\text{-} \Leftrightarrow \text{-} : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} & \forall \text{--} : \Pi\alpha : \mathsf{type}.(\mathsf{term}\ \alpha \to \mathsf{Prop}) \to \mathsf{Prop} \\
\forall_{\mathsf{type}} \text{-} : (\mathsf{type} \to \mathsf{Prop}) \to \mathsf{Prop} & \exists \text{--} : \Pi\alpha : \mathsf{type}.(\mathsf{term}\ \alpha \to \mathsf{Prop}) \to \mathsf{Prop} \\
\exists_{\mathsf{type}} \text{-} : (\mathsf{type} \to \mathsf{Prop}) \to \mathsf{Prop} & \text{-} =_{\text{-}} \text{-} : \Pi\alpha : \mathsf{type}.\mathsf{term}\ \alpha \to \mathsf{term}\ \alpha \to \mathsf{Prop}
\end{array}$$

Figure 3: Dedukti Declarations of TFF1$^{\equiv}$ Symbols

Translation Function for Types

$$|\alpha| := \alpha \qquad |T(\tau_1,\ldots,\tau_m)| := T\ |\tau_1|\ \ldots\ |\tau_m|$$

Translation Function for Terms

$$|x| := x \qquad |f(\tau_1,\ldots,\tau_m;e_1,\ldots,e_n)| := f\ |\tau_1|\ \ldots\ |\tau_m|\ |e_1|\ \ldots\ |e_n|$$

Translation Function for Formulæ

$$\begin{array}{rr}
|\top| := \top & |\bot| := \bot \\
|\neg\varphi| := \neg|\varphi| & |\varphi_1 \wedge \varphi_2| := |\varphi_1| \wedge |\varphi_2| \\
|\varphi_1 \vee \varphi_2| := |\varphi_1| \vee |\varphi_2| & |\varphi_1 \Rightarrow \varphi_2| := |\varphi_1| \Rightarrow |\varphi_2| \\
|\varphi_1 \Leftrightarrow \varphi_2| := |\varphi_1| \Leftrightarrow |\varphi_2| & |e_1 =_\tau e_2| := |e_1| =_{|\tau|} |e_2| \\
|\forall x : \tau.\ \varphi| := \forall\ |\tau|\ (\lambda x : \mathsf{term}\ |\tau|.\ |\varphi|) & |\exists x : \tau.\ \varphi| := \exists\ |\tau|\ (\lambda x : \mathsf{term}\ |\tau|.\ |\varphi|) \\
|\forall_{\mathsf{type}}\alpha : \mathsf{type}.\ \varphi| := \forall_{\mathsf{type}}\ (\lambda\alpha : \mathsf{type}\ .\ |\varphi|) & |\exists_{\mathsf{type}}\alpha : \mathsf{type}.\ \varphi| := \exists_{\mathsf{type}}\ (\lambda\alpha : \mathsf{type}\ .\ |\varphi|)
\end{array}$$

$$|P(\tau_1,\ldots,\tau_m;e_1,\ldots,e_n)| := P\ |\tau_1|\ \ldots\ |\tau_m|\ |e_1|\ \ldots\ |e_n|$$

Translation Function for Typing Contexts

$$|\emptyset| := \emptyset \qquad |\Delta, x : \tau| := |\Delta|, x : \mathsf{term}\ |\tau|$$

Translation Function for Theories

$$|\emptyset| := \Gamma_0 \text{ where } \Gamma_0 \text{ is the Dedukti context of Fig. 3}$$

$$|\mathscr{T}, T/m| := |\mathscr{T}|, T : \overbrace{\mathsf{type} \to \ldots \to \mathsf{type}}^{m \text{ times}} \to \mathsf{type}$$

$$|\mathscr{T}, f : \Pi(\alpha_1,\ldots,\alpha_m).\ (\tau_1,\ldots,\tau_n) \to \tau| := |\mathscr{T}|, f : \Pi\alpha_1 : \mathsf{type}.\ldots\Pi\alpha_m : \mathsf{type}.\ \mathsf{term}\ |\tau_1| \to \ldots \to \mathsf{term}\ |\tau_n| \to |\tau|$$

$$|\mathscr{T}, P : \Pi(\alpha_1,\ldots,\alpha_m).\ (\tau_1,\ldots,\tau_n) \to o| := |\mathscr{T}|, P : \Pi\alpha_1 : \mathsf{type}.\ldots\Pi\alpha_m : \mathsf{type}.\ \mathsf{term}\ |\tau_1| \to \ldots \to \mathsf{term}\ |\tau_n| \to \mathsf{Prop}$$

$$|\mathscr{T}, \mathrm{name} : \varphi| := |\mathscr{T}|, \mathrm{name} : \mathsf{prf}\ |\varphi|$$

$$|\mathscr{T}, l \longrightarrow_\Delta r| := |\mathscr{T}|, |l| \hookrightarrow_\Delta |r|$$

Figure 4: Translation Functions from TFF1$^{\equiv}$ to $\lambda\Pi^{\equiv}$

4 Zenon Modulo

Zenon Modulo [13] is an extension to deduction modulo [15] of the first-order tableau-based automated theorem prover Zenon [8]. It has also been improved to deal with typed formulæ and TFF1 input files. In this paper, we focus on the output format of Zenon Modulo. After finding a proof using its tableau-based proof-search algorithm [8], Zenon translates its proof tree into a *low level* format called LLproof, which

$$
\begin{array}{ll}
\mathsf{prf}\ \top & \hookrightarrow \Pi P : \mathsf{Prop}.\ \mathsf{prf}\ P \to \mathsf{prf}\ P \\
\mathsf{prf}\ \bot & \hookrightarrow \Pi P : \mathsf{Prop}.\ \mathsf{prf}\ P \\
\mathsf{prf}\ (\neg A) & \hookrightarrow \mathsf{prf}\ A \to \mathsf{prf}\ \bot \\
\mathsf{prf}\ (A \wedge B) & \hookrightarrow \Pi P : \mathsf{Prop}.\ (\mathsf{prf}\ A \to \mathsf{prf}\ B \to \mathsf{prf}\ P) \to \mathsf{prf}\ P \\
\mathsf{prf}\ (A \vee B) & \hookrightarrow \Pi P : \mathsf{Prop}.\ (\mathsf{prf}\ A \to \mathsf{prf}\ P) \to (\mathsf{prf}\ B \to \mathsf{prf}\ P) \to \mathsf{prf}\ P \\
\mathsf{prf}\ (A \Rightarrow B) & \hookrightarrow \mathsf{prf}\ A \to \mathsf{prf}\ B \\
\mathsf{prf}\ (A \Leftrightarrow B) & \hookrightarrow \mathsf{prf}\ ((A \Rightarrow B) \wedge (B \Rightarrow A)) \\
\mathsf{prf}\ (\forall\ \tau\ P) & \hookrightarrow \Pi x : \mathsf{term}\ \tau.\ \mathsf{prf}\ (P\ x) \\
\mathsf{prf}\ (\forall_{\mathsf{type}}\ P) & \hookrightarrow \Pi \alpha : \mathsf{type}.\ \mathsf{prf}\ (P\ \alpha) \\
\mathsf{prf}\ (\exists\ \tau\ P) & \hookrightarrow \Pi P : \mathsf{Prop}.\ (\Pi x : \mathsf{term}\ \tau.\ \mathsf{prf}\ (P\ x) \to \mathsf{prf}\ P) \to \mathsf{prf}\ P \\
\mathsf{prf}\ (\exists_{\mathsf{type}}\ P) & \hookrightarrow \Pi P : \mathsf{Prop}.\ (\Pi \alpha : \mathsf{type}.\ \mathsf{prf}\ (P\ \alpha) \to \mathsf{prf}\ P) \to \mathsf{prf}\ P \\
\mathsf{prf}\ (x =_\tau y) & \hookrightarrow \Pi P : (\mathsf{term}\ \tau \to \mathsf{Prop}).\ \mathsf{prf}\ (P\ x) \to \mathsf{prf}\ (P\ y)
\end{array}
$$

Figure 5: Shallow Definition of Logical Connectives in Dedukti

is a classical sequent-like proof system. This format is used for Zenon proofs before their automatic translation to Coq. LLproof is a one-sided sequent calculus with explicit contractions in every inference rule, which is close to an upside-down non-destructive tableau method.

We present in Figs. 6 and 7 the new proof system LLproof$^{\equiv}$, an adaptation of Zenon output format LLproof [8] to deduction modulo and TFF1 typing.

Normalization and deduction steps may interleave anywhere in the final proof tree. This leads to the introduction of the congruence relation $=_{\mathscr{RE}}$ inside rules of Figs. 6 and 7: if the formula P is in normal form (with respect to $\mathscr{RE}$), we denote by $[P]$ any formula congruent to P modulo $=_{\mathscr{RE}}$.

Extension of LLproof to TFF1 typing leads to the introduction of four new rules for quantification over type variables $\exists_{\mathsf{type}}$, $\neg\forall_{\mathsf{type}}$, $\forall_{\mathsf{type}}$ and $\neg\exists_{\mathsf{type}}$, and also to introduce some type information into

Closure and Quantifier-free Rules

$$\frac{}{\Gamma, [\bot] \vdash \bot}\ \bot \qquad \frac{}{\Gamma, [\neg\top] \vdash \bot}\ \neg\top \qquad \frac{}{\Gamma, [P], [\neg P] \vdash \bot}\ \mathrm{Ax} \qquad \frac{\Gamma, P \vdash \bot \qquad \Gamma, \neg P \vdash \bot}{\Gamma \vdash \bot}\ \mathrm{Cut}$$

$$\frac{}{\Gamma, [t \neq_\tau t] \vdash \bot}\ \neq \qquad \frac{}{\Gamma, [t =_\tau u], [u \neq_\tau t] \vdash \bot}\ \mathrm{Sym} \qquad \frac{\Gamma, \neg\neg P, P \vdash \bot}{\Gamma, [\neg\neg P] \vdash \bot}\ \neg\neg \qquad \frac{\Gamma, P \wedge Q, P, Q \vdash \bot}{\Gamma, [P \wedge Q] \vdash \bot}\ \wedge$$

$$\frac{\Gamma, P \vee Q, P \vdash \bot \qquad \Gamma, P \vee Q, Q \vdash \bot}{\Gamma, [P \vee Q] \vdash \bot}\ \vee \qquad \frac{\Gamma, P \Rightarrow Q, \neg P \vdash \bot \qquad \Gamma, P \Rightarrow Q, Q \vdash \bot}{\Gamma, [P \Rightarrow Q] \vdash \bot}\ \Rightarrow$$

$$\frac{\Gamma, P \Leftrightarrow Q, \neg P, \neg Q \vdash \bot \qquad \Gamma, P \Leftrightarrow Q, P, Q \vdash \bot}{\Gamma, [P \Leftrightarrow Q] \vdash \bot}\ \Leftrightarrow \qquad \frac{\Gamma, \neg(P \wedge Q), \neg P \vdash \bot \qquad \Gamma, \neg(P \wedge Q), \neg Q \vdash \bot}{\Gamma, [\neg(P \wedge Q)] \vdash \bot}\ \neg\wedge$$

$$\frac{\Gamma, \neg(P \vee Q), \neg P, \neg Q \vdash \bot}{\Gamma, [\neg(P \vee Q)] \vdash \bot}\ \neg\vee \qquad \frac{\Gamma, \neg(P \Rightarrow Q), P, \neg Q \vdash \bot}{\Gamma, [\neg(P \Rightarrow Q)] \vdash \bot}\ \neg\Rightarrow$$

$$\frac{\Gamma, \neg(P \Leftrightarrow Q), \neg P, Q \vdash \bot \qquad \Gamma, \neg(P \Leftrightarrow Q), P, \neg Q \vdash \bot}{\Gamma, [\neg(P \Leftrightarrow Q)] \vdash \bot}\ \neg\Leftrightarrow$$

Figure 6: LLproof$^{\equiv}$ Inference Rules of Zenon Modulo (part 1)

Quantifier Rules

$$\frac{\Gamma, \exists_{\mathsf{type}}\alpha : \mathsf{type}.\ P(\alpha), P(\tau) \vdash \bot}{\Gamma, [\exists_{\mathsf{type}}\alpha : \mathsf{type}.\ P(\alpha)] \vdash \bot}\ \exists_{\mathsf{type}} \qquad \frac{\Gamma, \neg\forall_{\mathsf{type}}\alpha : \mathsf{type}.\ P(\alpha), \neg P(\tau) \vdash \bot}{\Gamma, [\neg\forall_{\mathsf{type}}\alpha : \mathsf{type}.\ P(\alpha)] \vdash \bot}\ \neg\forall_{\mathsf{type}}$$

where τ is a fresh type constant

$$\frac{\Gamma, \forall_{\mathsf{type}}\alpha : \mathsf{type}.\ P(\alpha), P(\beta) \vdash \bot}{\Gamma, [\forall_{\mathsf{type}}\alpha : \mathsf{type}.\ P(\alpha)] \vdash \bot}\ \forall_{\mathsf{type}} \qquad \frac{\Gamma, \neg\exists_{\mathsf{type}}\alpha : \mathsf{type}.\ P(\alpha), \neg P(\beta) \vdash \bot}{\Gamma, [\neg\exists_{\mathsf{type}}\alpha : \mathsf{type}.\ P(\alpha)] \vdash \bot}\ \neg\exists_{\mathsf{type}}$$

where β is any closed type

$$\frac{\Gamma, \exists x : \tau.\ P(x), P(c) \vdash \bot}{\Gamma, [\exists x : \tau.\ P(x)] \vdash \bot}\ \exists \qquad \frac{\Gamma, \neg\forall x : \tau.\ P(x), \neg P(c) \vdash \bot}{\Gamma, [\neg\forall x : \tau.\ P(x)] \vdash \bot}\ \neg\forall$$

where $c : \tau$ is a fresh constant

$$\frac{\Gamma, \forall x : \tau.\ P(x), P(t) \vdash \bot}{\Gamma, [\forall x : \tau.\ P(x)] \vdash \bot}\ \forall \qquad \frac{\Gamma, \neg\exists x : \tau.\ P(x), \neg P(t) \vdash \bot}{\Gamma, [\neg\exists x : \tau.\ P(x)] \vdash \bot}\ \neg\exists$$

where $t : \tau$ is any closed term

Special Rules

$$\frac{\Delta, t_1 \neq_{\tau'_1} u_1 \vdash \bot \quad \dots \quad \Delta, t_n \neq_{\tau'_n} u_n \vdash \bot}{\Gamma, [P(\tau_1, \dots, \tau_m; t_1, \dots, t_n)], [\neg P(\tau_1, \dots, \tau_m; u_1, \dots, u_n)] \vdash \bot}\ \text{Pred}$$

where $\Delta = \Gamma \cup \{P(\tau_1, \dots, \tau_m; t_1, \dots, t_n), \neg P(\tau_1, \dots, \tau_m; u_1, \dots, u_n)\}$

$$\frac{\Delta, t_1 \neq_{\tau'_1} u_1 \vdash \bot \quad \dots \quad \Delta, t_n \neq_{\tau'_n} u_n \vdash \bot}{\Gamma, [f(\tau_1, \dots, \tau_m; t_1, \dots, t_n) \neq_\tau f(\tau_1, \dots, \tau_m; u_1, \dots, u_n)] \vdash \bot}\ \text{Fun}$$

where $\Delta = \Gamma \cup \{f(\tau_1, \dots, \tau_m; t_1, \dots, t_n) \neq_\tau f(\tau_1, \dots, \tau_m; u_1, \dots, u_n)\}$

$$\frac{\Delta, H_{11}, \dots, H_{1m} \vdash \bot \quad \dots \quad \Delta, H_{n1}, \dots, H_{nq} \vdash \bot}{\Gamma, [C_1], \dots, [C_p] \vdash \bot}\ \text{Ext}(\text{name}, \text{args}, C_1, \dots, C_p, H_{11}, \dots, H_{nq})$$

where $\Delta = \Gamma \cup \{C_1, \dots, C_p\}$

Figure 7: LLproof$^{\equiv}$ Inference Rules of Zenon Modulo (part 2)

other rules dealing with equality or quantification. For instance, equality of two closed terms t and u, both of type τ, is denoted by $t =_\tau u$. For predicate and function symbols, we first list types, then terms, separated by a semi-colon.

Finally, last difference regarding rules presented in [8] is the removal of rules "definition" and "lemma". Zenon Modulo, unlike Zenon, does not need to explicitly unfold definitions and the lemma constructions have been removed.

4.1 Translation of Zenon Modulo Proofs into $\lambda\Pi^{\equiv}$

We present in Fig. 8 a deep embedding of LLproof$^{\equiv}$ into $\lambda\Pi^{\equiv}$. We declare a constant for each inference rule, except for special rules Pred and Fun which have a dependency on the arity n of their underlying predicate and function. Fortunately, they can be expressed with the following Subst inference rule which corresponds to the substitution in a predicate P of a subterm $t : \tau'$ by another $u : \tau'$:

$$\frac{\Gamma, P(\vec{\tau}; t), t \neq_{\tau'} u \vdash \bot \qquad \Gamma, P(\vec{\tau}; t), P(\vec{\tau}; u) \vdash \bot}{\Gamma, P(\vec{\tau}; t) \vdash \bot}\ \text{Subst}$$

Zenon Modulo Rules

$\mathsf{R}_{\bot}$: $\mathsf{prf}\ \bot \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\top}$: $\mathsf{prf}\ (\neg\top) \to \mathsf{prf}\ \bot$
R_{Ax} : $\Pi P : \mathsf{Prop}.\ \mathsf{prf}\ P \to \mathsf{prf}\ (\neg P) \to \mathsf{prf}\ \bot$
R_{Cut} : $\Pi P : \mathsf{Prop}.\ (\mathsf{prf}\ P \to \mathsf{prf}\ \bot) \to (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neq}$: $\Pi \alpha : \mathsf{type}.\ \Pi t : \mathsf{term}\ \alpha.\ \mathsf{prf}\ (t \neq_\alpha t) \to \mathsf{prf}\ \bot$
R_{Sym} : $\Pi \alpha : \mathsf{type}.\ \Pi t,u : \mathsf{term}\ \alpha.\ \mathsf{prf}\ (t =_\alpha u) \to \mathsf{prf}\ (u \neq_\alpha t) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\neg}$: $\Pi P : \mathsf{Prop}.\ (\mathsf{prf}\ P \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (\neg\neg P) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\wedge}$: $\Pi P,Q : \mathsf{Prop}.\ (\mathsf{prf}\ P \to \mathsf{prf}\ Q \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (P \wedge Q) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\vee}$: $\Pi P,Q : \mathsf{Prop}.\ (\mathsf{prf}\ P \to \mathsf{prf}\ \bot) \to (\mathsf{prf}\ Q \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (P \vee Q) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\Rightarrow}$: $\Pi P,Q : \mathsf{Prop}.\ (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ \bot) \to (\mathsf{prf}\ Q \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (P \Rightarrow Q) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\Leftrightarrow}$: $\Pi P,Q : \mathsf{Prop}.\ (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ (\neg Q) \to \mathsf{prf}\ \bot) \to (\mathsf{prf}\ P \to \mathsf{prf}\ Q \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (P \Leftrightarrow Q) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\wedge}$: $\Pi P,Q : \mathsf{Prop}.\ (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ \bot) \to (\mathsf{prf}\ (\neg Q) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (\neg(P \wedge Q)) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\vee}$: $\Pi P,Q : \mathsf{Prop}.\ (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ (\neg Q) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (\neg(P \vee Q)) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\Rightarrow}$: $\Pi P,Q : \mathsf{Prop}.\ (\mathsf{prf}\ P \to \mathsf{prf}\ (\neg Q) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (\neg(P \Rightarrow Q)) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\Leftrightarrow}$: $\Pi P,Q : \mathsf{Prop}.\ (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ Q \to \mathsf{prf}\ \bot) \to (\mathsf{prf}\ P \to \mathsf{prf}\ (\neg Q) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (\neg(P \Leftrightarrow Q)) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\exists}$: $\Pi \alpha : \mathsf{type}.\ \Pi P : (\mathsf{term}\ \alpha \to \mathsf{Prop}).\ (\Pi t : \mathsf{term}\ \alpha.\ (\mathsf{prf}\ (P\ t) \to \mathsf{prf}\ \bot)) \to \mathsf{prf}\ (\exists\ \alpha\ P) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\forall}$: $\Pi \alpha : \mathsf{type}.\ \Pi P : (\mathsf{term}\ \alpha \to \mathsf{Prop}).\ \Pi t : \mathsf{term}\ \alpha.\ (\mathsf{prf}\ (P\ t) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (\forall\ \alpha\ P) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\exists}$: $\Pi \alpha : \mathsf{type}.\ \Pi P : (\mathsf{term}\ \alpha \to \mathsf{Prop}).\ \Pi t : \mathsf{term}\ \alpha.\ (\mathsf{prf}\ (\neg(P\ t)) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (\neg(\exists\ \alpha\ P) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\forall}$: $\Pi \alpha : \mathsf{type}.\ \Pi P : (\mathsf{term}\ \alpha \to \mathsf{Prop}).\ (\Pi t : \mathsf{term}\ \alpha.\ (\mathsf{prf}\ (\neg(P\ t)) \to \mathsf{prf}\ \bot)) \to \mathsf{prf}\ (\neg\forall\ \alpha\ P) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\exists_{\mathsf{type}}}$: $\Pi P : (\mathsf{type} \to \mathsf{Prop}).\ (\Pi \alpha : \mathsf{type}.\ (\mathsf{prf}\ (P\ \alpha) \to \mathsf{prf}\ \bot)) \to \mathsf{prf}\ (\exists_{\mathsf{type}}\ P) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\forall_{\mathsf{type}}}$: $\Pi P : (\mathsf{type} \to \mathsf{Prop}).\ \Pi \alpha : \mathsf{type}.\ (\mathsf{prf}\ (P\ \alpha) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (\forall_{\mathsf{type}}\ f) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\exists_{\mathsf{type}}}$: $\Pi P : (\mathsf{type} \to \mathsf{Prop}).\ \Pi \alpha : \mathsf{type}.\ (\mathsf{prf}\ (\neg(P\ \alpha)) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (\neg(\exists_{\mathsf{type}}\ P)) \to \mathsf{prf}\ \bot$
$\mathsf{R}_{\neg\forall_{\mathsf{type}}}$: $\Pi P : (\mathsf{type} \to \mathsf{Prop}).\ (\Pi \alpha : \mathsf{type}.\ (\mathsf{prf}\ (\neg(P\ \alpha)) \to \mathsf{prf}\ \bot)) \to \mathsf{prf}\ (\neg(\forall_{\mathsf{type}}\ P)) \to \mathsf{prf}\ \bot$
R_{Subst} : $\Pi \alpha : \mathsf{type}.\ \Pi P : (\mathsf{term}\ \alpha \to \mathsf{Prop}).\ \Pi t,u : \mathsf{term}\ \alpha.\ (\mathsf{prf}\ (t \neq_\alpha u) \to \mathsf{prf}\ \bot) \to (\mathsf{prf}\ (P\ u) \to \mathsf{prf}\ \bot) \to \mathsf{prf}\ (P\ t) \to \mathsf{prf}\ \bot$

Figure 8: $\mathsf{LLproof}^{\equiv}$ in $\lambda\Pi^{\equiv}$

The special rules Pred and Fun can be easily decomposed into n applications of the Subst rule. For instance, for a binary predicate P, from (we omit to repeat the context Γ)

$$\dfrac{\dfrac{\Pi_1}{t_1 \neq_{\tau'} u_1 \vdash \bot} \quad \dfrac{\Pi_2}{t_2 \neq_{\tau''} u_2 \vdash \bot}}{P(\vec{\tau}; t_1, t_2), \neg P(\vec{\tau}; u_1, u_2) \vdash \bot}\ \text{Pred}$$

we obtain

$$\dfrac{\dfrac{\Pi_1}{t_1 \neq_{\tau'} u_1 \vdash \bot} \quad \dfrac{\dfrac{\Pi_2}{t_2 \neq_{\tau''} u_2 \vdash \bot} \quad \dfrac{}{P(\vec{\tau}; u_1, u_2)}\ \text{Ax}}{P(\vec{\tau}; u_1, t_2)}\ \text{Subst}}{P(\vec{\tau}; t_1, t_2), \neg P(\vec{\tau}; u_1, u_2) \vdash \bot}\ \text{Subst}$$

In Fig. 9, we present the translation function for $\mathsf{LLproof}^{\equiv}$ sequents and proofs into $\lambda\Pi^{\equiv}$. Let us present a simple example. We want to translate this proof tree:

$$\Pi := \dfrac{\dfrac{\Pi_P}{\Gamma, P \vee Q, P \vdash \bot} \quad \dfrac{\Pi_Q}{\Gamma, P \vee Q, Q \vdash \bot}}{\Gamma, P \vee Q \vdash \bot}\ \vee(P,Q)$$

Translation Function for Sequents

$$|[\varphi_1],\ldots,[\varphi_n] \vdash \bot| := x_{\varphi_1} : \mathsf{prf}\ |\varphi_1|,\ldots,x_{\varphi_n} : \mathsf{prf}\ |\varphi_n|$$

Translation Function for Proofs

$$\left|\frac{\dfrac{\Pi_1}{\Delta,H_{11},\ldots,H_{1m}\vdash\bot}\quad\ldots\quad\dfrac{\Pi_n}{\Delta,H_{n1},\ldots,H_{nq}\vdash\bot}}{\Gamma,C_1,\ldots,C_p\vdash\bot}\ \mathrm{Rule}(\mathrm{Arg}_1,\ldots,\mathrm{Arg}_r)\right| :=$$

$$\begin{array}{l}\mathsf{R}_{\mathrm{Rule}}\ |\mathrm{Arg}_1|\ldots|\mathrm{Arg}_r| \\ \quad (\lambda x_{H_{11}} : \mathsf{prf}\ |H_{11}|.\ldots.\lambda x_{H_{1m}} : \mathsf{prf}\ |H|_{1m}.|\Pi_1|) \\ \quad \vdots \\ \quad (\lambda x_{H_{n1}} : \mathsf{prf}\ |H_{n1}|.\ldots.\lambda x_{H_{nq}} : \mathsf{prf}\ \left|H_{nq}\right|.|\Pi_n|) \\ \quad x_{C_1}\ldots x_{C_p}\end{array}$$

Figure 9: Translation Functions for LLproof$^{\equiv}$ Proofs into $\lambda\Pi^{\equiv}$

where Π_P and Π_Q are respectively proofs of sequents $\Gamma,P\vdash\bot$ and $\Gamma,Q\vdash\bot$, and where we annotate rule names with its parameters. Then, by applying the translation procedure of Figs. 4 and 9, we obtain the Dedukti term

$$\mathsf{R}_{\vee}\ |P|\ |Q|\ (\lambda x_P : \mathsf{prf}\ |P|.|\Pi_P|)\ (\lambda x_Q : \mathsf{prf}\ |Q|.|\Pi_Q|)\ x_{P\vee Q}$$

where the notation $|x|$ means the translation of x into $\lambda\Pi^{\equiv}$, and x_P is a variable declared of type $\mathsf{prf}\ |P|$. We then check that Π is a proof of the sequent $\Gamma,P\vee Q\vdash\bot$ in a TFF1$^{\equiv}$ theory $\mathscr{T}$, by checking that $|\mathscr{T}|,|\Gamma,P\vee Q|\vdash|\Pi| : \mathsf{prf}\ \bot$ in $\lambda\Pi^{\equiv}$.

More generally, for any LLproof$^{\equiv}$ proof Π and any sequent $\Gamma\vdash\bot$, we check that Π is a proof of $\Gamma\vdash\bot$ by checking the $\lambda\Pi^{\equiv}$ typing judgment $|\mathscr{T}|,|\Gamma|\vdash|\Pi| : \mathsf{prf}\ \bot$.

4.2 Shallow Embedding of LLproof$^{\equiv}$

The embedding of LLproof$^{\equiv}$ presented in Fig. 8 can also be lifted to a shallow embedding. In Fig. 13 of Appendix A, we present rewrite rules that prove all constants corresponding to LLproof$^{\equiv}$ inference rules into the logic presented in Sec. 3. This has been written in Dedukti syntax and successfully checked by Dedukti (see the file `modulogic.dk` distributed with the source code of Zenon Modulo[1]). The only remaining axiom is the law of excluded middle. This shows the soundness of LLproof$^{\equiv}$ relatively to the consistency of the logic of Sec. 3.

5 Experimental Results

Zenon Modulo helps to automatically discharge proof obligations in particular in the two projects FoCaLiZe [17] and BWare [14]. We present in this section some examples of theories, and simple related properties, that are handled successfully by Zenon Modulo, and its translation to Dedukti.

[1] https://www.rocq.inria.fr/deducteam/ZenonModulo/

Declarations

$\mathsf{bool}/0$

$\mathsf{false} : \mathsf{bool}$

$\mathsf{true} : \mathsf{bool}$

$\sim : \mathsf{bool} \to \mathsf{bool}$

$\_ \;\&\&\; \_ : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$

$\_ \;||\; \_ : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$

$\mathsf{if}\_ \;\mathsf{then}\; \_ \;\mathsf{else}\; \_ : \Pi\alpha.\ \mathsf{bool} \to \alpha \to \alpha \to \alpha$

Rewrite rules

$\mathsf{true} \;\&\&\; a \longrightarrow a$

$a \;\&\&\; \mathsf{true} \longrightarrow a$

$\mathsf{false} \;\&\&\; a \longrightarrow \mathsf{false}$

$a \;\&\&\; \mathsf{false} \longrightarrow \mathsf{false}$

$a \;\&\&\; a \longrightarrow a$

$a \;\&\&\; (b \;\&\&\; c) \longrightarrow (a \;\&\&\; b) \;\&\&\; c$

$\mathsf{true} \;||\; a \longrightarrow \mathsf{true}$

$a \;||\; \mathsf{true} \longrightarrow \mathsf{true}$

$\mathsf{false} \;||\; a \longrightarrow a$

$a \;||\; \mathsf{false} \longrightarrow a$

$a \;||\; a \longrightarrow a$

$a \;||\; (b \;||\; c) \longrightarrow (a \;||\; b) \;||\; c$

$\sim \mathsf{true} \longrightarrow \mathsf{false}$

$\sim \mathsf{false} \longrightarrow \mathsf{true}$

$\sim (\sim a) \longrightarrow a$

$\sim (a \;||\; b) \longrightarrow (\sim a) \;\&\&\; (\sim b)$

$\sim (a \;\&\&\; b) \longrightarrow (\sim a) \;||\; (\sim b)$

$a \;\&\&\; (b \;||\; c) \longrightarrow (a \;\&\&\; b) \;||\; (a \;\&\&\; c)$

$(a \;||\; b) \;\&\&\; c \longrightarrow (a \;\&\&\; c) \;||\; (b \;\&\&\; c)$

$\mathsf{if}_\alpha \;\mathsf{true}\; \mathsf{then}\; t\; \mathsf{else}\; e \longrightarrow t$

$\mathsf{if}_\alpha \;\mathsf{false}\; \mathsf{then}\; t\; \mathsf{else}\; e \longrightarrow e$

Extension deduction rules

$$\frac{\Gamma, \neg P(\mathsf{true}) \vdash \bot \qquad \Gamma, \neg P(\mathsf{false}) \vdash \bot}{\Gamma, [\neg\forall b : \mathsf{bool}.\ P(b)] \vdash \bot}\ \text{Ext(bool-case-}\neg\forall, P)$$

$$\frac{\Gamma, P(\mathsf{true}) \vdash \bot \qquad \Gamma, P(\mathsf{false}) \vdash \bot}{\Gamma, [\exists b : \mathsf{bool}.\ P(b)] \vdash \bot}\ \text{Ext(bool-case-}\exists, P)$$

Figure 10: A $\mathsf{TFF1}^{\equiv}$ Theory of Booleans

5.1 Application to FoCaLiZe

FoCaLiZe is a framework for specifying, developing and certifying programs. The specification language is first-order logic and proofs can be discharged to Zenon or Zenon Modulo. The FoCaLiZe compiler produces both a regular program written in OCaml and a certificate written either in Coq or in Dedukti (but only the Dedukti output can be used from Zenon Modulo).

In FoCaLiZe, specifications usually rely a lot on the primitive type bool so it is important that Zenon Modulo deals with booleans efficiently. In order to prove all propositional tautologies, it is enough to add the following rules for reasoning by case on booleans (together with truth tables of connectives):

$$\frac{\Gamma, \neg P(\mathsf{true}) \vdash \bot \qquad \Gamma, \neg P(\mathsf{false}) \vdash \bot}{\Gamma, [\neg\forall b : \mathsf{bool}.\ P(b)] \vdash \bot}\ \text{Ext(bool-case-}\neg\forall, P)$$

$$\frac{\Gamma, P(\mathsf{true}) \vdash \bot \qquad \Gamma, P(\mathsf{false}) \vdash \bot}{\Gamma, [\exists b : \mathsf{bool}.\ P(b)] \vdash \bot}\ \text{Ext(bool-case-}\exists, P)$$

However, we get a much smaller proof-search space and smaller proofs by adding common algebraic laws as rewrite rules. In Fig. 10, we define a theory of booleans in $\mathsf{TFF1}^{\equiv}$. This theory handles idempotency and associativity of conjunction and disjunction but not commutativity because the rule

```
λx1 : prf(¬(∀ bool (λx : term bool. ∀ bool (λy : term bool. x && y =bool y && x)))).
 Rbool-case-¬∀
  (λx : term bool. ∀ bool (λy : term bool. x && y =bool y && x))
  (λx2 : prf(¬(∀ bool (λy : term bool. y =bool y)))).
   R¬∀ bool
    (λy : term bool. y ≠bool y)
    (λa : term bool.
     λx3 : prf(a ≠bool a).
      R≠ bool a x3)
    x2)
  (λx4 : prf(¬(∀ bool (λy : term bool. false =bool false)))).
   R¬∀ bool
    (λy : term bool. false =bool false)
    (λa : term bool.
     λx5 : prf(false ≠bool false).
      R≠ bool false x5)
    x4)
  x1
```

Figure 11: Proof Certificate for Commutativity of Conjunction in Dedukti

$a \,\&\&\, b \hookrightarrow b \,\&\&\, a$ would lead to a non terminating rewrite system; therefore, commutativity is a lemma with the following proof:

$$\dfrac{\dfrac{\dfrac{}{a \neq_{\mathsf{bool}} a \vdash \bot}\neq}{\neg\forall y : \mathsf{bool}.\ y =_{\mathsf{bool}} y \vdash \bot}\neg\forall \quad \dfrac{\dfrac{}{\mathsf{false} \neq_{\mathsf{bool}} \mathsf{false} \vdash \bot}\neq}{\neg\forall y : \mathsf{bool}.\ \mathsf{false} =_{\mathsf{bool}} \mathsf{false} \vdash \bot}\neg\forall}{\neg\forall x,y : \mathsf{bool}.\ x \,\&\&\, y =_{\mathsf{bool}} y \,\&\&\, x \vdash \bot}\ \text{Ext(bool-case-}\neg\forall\text{)}$$

The translation of this proof in Dedukti is shown in Fig. 11.

5.2 Application to Set Theory

The BWare project is an industrial research project that aims to provide a framework to support the automated verification of proof obligations coming from the development of industrial applications using the B method [1]. The B method relies on a particular set theory with types. In the context of the BWare project, this typed set theory has been encoded into WhyML, the native language of Why3 [16]. To call Zenon Modulo, Why3 translates proof obligations and the B theory into TFF1 format. If it succeeds in proving the proof obligation, Zenon Modulo produces a proof certificate containing both the theory and the term, following the model presented in Fig. 12.

The BWare project provides a large benchmark made of 12,876 proof obligations coming from industrial projects. The embedding presented in this paper allowed us to verify with Dedukti all the 10,340 proof obligations that are proved by Zenon Modulo.

Let us present a small subset of this set theory, and a simple example of LLproof$^{\equiv}$ proof produced by Zenon Modulo. The theory consists of three axioms that have been turned into rewrite rules. We define constructors: a type constructor set, the membership predicate $\in$, equality on sets $=_{\mathsf{set}}$, the empty set $\emptyset$ and difference of sets $-$. For readability, we use an infix notation and let type parameters of functions and predicates in subscript. We want to prove the property

$$\forall_{\mathsf{type}}\alpha : \mathsf{type}.\ \forall s : \mathsf{set}\alpha.\ s -_\alpha s =_{\mathsf{set}\alpha} \emptyset_\alpha$$

$$
\begin{array}{rl}
\mathsf{set}\text{-}: & \mathsf{type} \to \mathsf{type} \\
\text{-} \in_{\text{-}} \text{-}: & \Pi\alpha : \mathsf{type}.\ (\mathsf{term}\ \mathsf{set}\alpha) \to (\mathsf{term}\ \mathsf{set}\alpha) \to \mathsf{Prop} \\
\text{-} =_{\mathsf{set}\text{-}} \text{-}: & \Pi\alpha : \mathsf{type}.\ (\mathsf{term}\ \mathsf{set}\alpha) \to (\mathsf{term}\ \mathsf{set}\alpha) \to \mathsf{Prop} \\
\emptyset_{\text{-}}: & \Pi\alpha : \mathsf{type}.\ (\mathsf{term}\ \mathsf{set}\alpha) \\
\text{-} -_{\text{-}} \text{-}: & \Pi\alpha : \mathsf{type}.\ (\mathsf{term}\ \mathsf{set}\alpha) \to (\mathsf{term}\ \mathsf{set}\alpha) \to (\mathsf{term}\ \mathsf{set}\alpha)
\end{array}
$$

$$
\begin{array}{rl}
s =_{\mathsf{set}\alpha} t \hookrightarrow & \forall\ (\alpha)\ (\lambda x : (\mathsf{term}\ \alpha).\ x \in_\alpha s \Leftrightarrow x \in_\alpha t) \\
x \in_\alpha \emptyset_\alpha \hookrightarrow & \bot \\
x \in_\alpha s -_\alpha t \hookrightarrow & x \in_\alpha s \wedge x \notin_\alpha t
\end{array}
$$

$$
\begin{array}{rl}
\mathit{Goal}: & \mathsf{prf}\ (\neg(\forall_{\mathsf{type}}\ (\lambda\alpha : \mathsf{type}.\ (\forall\ (\mathsf{set}\alpha)\ (\lambda s : (\mathsf{term}\ \mathsf{set}\alpha).\ s -_\alpha s =_{\mathsf{set}\alpha} \emptyset_\alpha)))) \to \mathsf{prf}\ \bot) \\
[]\ \mathit{Goal} \hookrightarrow & \lambda x_1 : \mathsf{prf}\ (\neg(\forall_{\mathsf{type}}\ (\lambda\alpha : \mathsf{type}.\ (\forall\ (\mathsf{set}\alpha)\ (\lambda s : (\mathsf{term}\ \mathsf{set}\alpha).\ s -_\alpha s =_{\mathsf{set}\alpha} \emptyset_\alpha))))).
\end{array}
$$

$$
\begin{array}{l}
\mathsf{R}_{\neg\forall_{\mathsf{type}}}\ (\lambda\alpha : \mathsf{type}.\ (\forall\ (\mathsf{set}\alpha)\ \lambda s : (\mathsf{term}\ \mathsf{set}\alpha).\ s -_\alpha s =_{\mathsf{set}\alpha} \emptyset_\alpha)) \\
\quad (\lambda\tau : \mathsf{type}. \\
\quad\ \lambda x_2 : \mathsf{prf}\ (\neg(\forall\ (\mathsf{set}\tau)\ (\lambda s : (\mathsf{term}\ \mathsf{set}\tau).\ s -_\tau s =_{\mathsf{set}\tau} \emptyset_\tau))). \\
\quad\ \mathsf{R}_{\neg\forall}\ (\mathsf{set}\tau) \\
\quad\quad (\lambda s : (\mathsf{term}\ \mathsf{set}\tau).\ s -_\tau s =_{\mathsf{set}\tau} \emptyset_\tau) \\
\quad\quad (\lambda c_1 : (\mathsf{term}\ \mathsf{set}\tau). \\
\quad\quad\ \lambda x_3 : \mathsf{prf}\ (c_1 -_\tau c_1 \neq_{\mathsf{set}\tau} \emptyset_\tau). \\
\quad\quad\ \mathsf{R}_{\neg\forall}\ (\tau) \\
\quad\quad\quad (\lambda x : (\mathsf{term}\ \tau).\ (x \in_\tau c_1 -_\tau c_1) \Leftrightarrow (x \in_\tau \emptyset_\tau)) \\
\quad\quad\quad (\lambda c_2 : (\mathsf{term}\ \tau). \\
\quad\quad\quad\ \lambda x_4 : \mathsf{prf}\ (\neg((c_2 \in_\tau c_1 -_\tau c_1) \Leftrightarrow (c_2 \in_\tau \emptyset_\tau))). \\
\quad\quad\quad\ \mathsf{R}_{\neg\Leftrightarrow}\ (c_2 \in_\tau c_1 -_\tau c_1) \\
\quad\quad\quad\quad (c_2 \in_\tau \emptyset_\tau) \\
\quad\quad\quad\quad (\lambda x_5 : \mathsf{prf}\ (\neg(c_2 \in_\tau c_1 -_\tau c_1)). \\
\quad\quad\quad\quad\ \lambda x_6 : \mathsf{prf}\ (c_2 \in_\tau \emptyset_\tau). \\
\quad\quad\quad\quad\ \mathsf{R}_\bot\ x_6) \\
\quad\quad\quad\quad (\lambda x_7 : \mathsf{prf}\ (c_2 \in_\tau c_1 -_\tau c_1). \\
\quad\quad\quad\quad\ \lambda x_8 : \mathsf{prf}\ (\neg(c_2 \in_\tau \emptyset_\tau)). \\
\quad\quad\quad\quad\ \mathsf{R}_\wedge\ (c_2 \in_\tau c_1) \\
\quad\quad\quad\quad\quad (c_2 \notin_\tau c_1) \\
\quad\quad\quad\quad\quad (\lambda x_9 : \mathsf{prf}\ (c_2 \in_\tau c_1). \\
\quad\quad\quad\quad\quad\ \lambda x_9 : \mathsf{prf}\ (c_2 \notin_\tau c_1). \\
\quad\quad\quad\quad\quad\ \mathsf{R}_{Ax}\ (c_2 \notin_\tau c_1) \\
\quad\quad\quad\quad\quad\quad x_8 \\
\quad\quad\quad\quad\quad\quad x_9) \\
\quad\quad\quad\quad\quad x_7) \\
\quad\quad\quad\quad x_4) \\
\quad\quad\quad x_3) \\
\quad\quad x_2) \\
\quad x_1)
\end{array}
$$

Figure 12: Proof Certificate for a B Set Theory Property in Dedukti

with the theory:

$$\begin{array}{rl}
\mathsf{set}\text{-}\ /1 & \\
\text{-} \in_{\text{-}} \text{-} : \Pi\alpha.\ \mathsf{set}\alpha \to \mathsf{set}\alpha \to \mathsf{Prop} & s =_{\mathsf{set}\alpha} t \longrightarrow \forall x : \alpha.\ x \in_\alpha s \Leftrightarrow x \in_\alpha t \\
\text{-} =_{\mathsf{set}\text{-}} \text{-} : \Pi\alpha.\ \mathsf{set}\alpha \to \mathsf{set}\alpha \to \mathsf{Prop} & x \in_\alpha \emptyset_\alpha \longrightarrow \bot \\
\emptyset_{\text{-}} : \Pi\alpha.\ \mathsf{set}\alpha & x \in_\alpha s -_\alpha t \longrightarrow x \in_\alpha s \wedge x \notin_\alpha t \\
\text{-} -_{\text{-}} \text{-} : \Pi\alpha.\ \mathsf{set}\alpha \to \mathsf{set}\alpha \to \mathsf{set}\alpha &
\end{array}$$

The LLproof proof tree generated by Zenon Modulo is (we omit to repeat context Γ):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(c_2 \in_\tau c_1 -_\tau c_1), c_2 \in_\tau \emptyset_\tau \vdash \bot}\,\bot \quad \dfrac{\dfrac{}{c_2 \in_\tau c_1, c_2 \notin_\tau c_1 \vdash \bot}\,\mathrm{Ax}}{c_2 \in_\tau c_1 -_\tau c_1, \neg(c_2 \in_\tau \emptyset_\tau) \vdash \bot}\,\wedge}{\neg((c_2 \in_\tau c_1 -_\tau c_1) \Leftrightarrow (c_2 \in_\tau \emptyset_\tau)) \vdash \bot}\,\neg\Leftrightarrow}{\neg(c_1 -_\tau c_1 =_{\mathsf{set}\tau} \emptyset_\tau) \vdash \bot}\,\neg\forall}{\neg(\forall s : \mathsf{set}\tau.\ s -_\tau s =_{\mathsf{set}\tau} \emptyset_\tau) \vdash \bot}\,\neg\forall}{\neg(\forall_{\mathsf{type}}\alpha : \mathsf{type}.\ \forall s : \mathsf{set}\alpha.\ s -_\alpha s =_{\mathsf{set}\alpha} \emptyset_\alpha) \vdash \bot}\,\neg\forall_{\mathsf{type}}$$

We obtain the proof certificate of Fig. 12 checkable by Dedukti, using the file `modulogic.dk`, and that is successfully checked.

6 Conclusion

We have presented a shallow embedding of Zenon Modulo proofs into Dedukti. For this encoding, we have needed to embed into $\lambda\Pi^{\equiv}$ an extension to deduction modulo of the underlying logic of the TFF1 format, denoted by TFF1$^{\equiv}$. We then defined LLproof$^{\equiv}$, the extension to TFF1$^{\equiv}$ of the proof system LLproof, which is the output format of Zenon Modulo. Finally, we have embedded LLproof$^{\equiv}$ into $\lambda\Pi^{\equiv}$ by giving the translation function for proofs. This embedding is shallow in the sense that we have reused the features of the target language and have not declared new constants for connectives and inference rules. The only axiom that we have added is the law of excluded middle.

This embedding has helped us to verify a large set of proof obligations coming from two different projects. FoCaLiZe can now benefit from deduction modulo to improve program verification when dealing with theories. In BWare, this work allowed us to certify all the proofs generated by Zenon Modulo.

Our work is closely related to the embedding of iProver Modulo proofs into Dedukti [10]. The two main differences are the assumption of the excluded middle and the extension of the logic to deal with ML-style polymorphism. Because these shallow encodings are close, we could easily share proofs of untyped formulæ with iProver Modulo.

We do not have to trust the full implementation of Zenon Modulo but only the translation of TFF1$^{\equiv}$ problems to $\lambda\Pi^{\equiv}$ discussed in Sec. 3 and, of course, Dedukti. In the case of FoCaLiZe, we go even further by using an external translator, Focalide [11]. Hence Zenon Modulo requires no confidence in that context. As future work, we want export this model. To achieve that, deduction tools must be able to read Dedukti in addition to write some. This model improves the confidence on automated deduction tools because it is no more possible to introduce inconsistency inside a proof certificate. In addition, in case of the verification of several formulæ, it should be possible to inject terms coming from different tools inside the same Dedukti file. A first experiment with Zenon Modulo and iProver Modulo in FoCaLiZe would be an interesting proof of concept.

References

[1] Jean-Raymond Abrial (1996): *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, doi:10.1017/CBO9780511624162.

[2] Hendrik Pieter Barendregt, Wil Dekkers & Richard Statman (2013): *Lambda calculus with types*. Cambridge University Press, doi:10.1017/CBO9781139032636.

[3] Henk Barendregt & Erik Barendsen (2002): *Autarkic Computations in Formal Proofs*. *Journal of Automated Reasoning (JAR)* 28, doi:10.1023/A:1015761529444.

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs & K Rustan M Leino (2006): *Boogie: A modular reusable verifier for object-oriented programs*. In: *Formal Methods for Components and Objects*, Springer, doi:10.1007/11804192_17.

[5] Jasmin Christian Blanchette & Andrei Paskevich (2013): *TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism*. In: *Conference on Automated Deduction (CADE), LNCS* 7898, Springer, doi:10.1007/978-3-642-38574-2_29.

[6] Mathieu Boespflug & Guillaume Burel (2012): *CoqInE: translating the calculus of inductive constructions into the λΠ-calculus modulo*. In: *Proof Exchange for Theorem Proving (PxTP)*.

[7] Mathieu Boespflug, Quentin Carbonneaux & Olivier Hermant (2012): *The λΠ-Calculus Modulo as a Universal Proof Language*. In: *Proof Exchange for Theorem Proving (PxTP)*.

[8] Richard Bonichon, David Delahaye & Damien Doligez (2007): *Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs*. In: *Logic for Programming Artificial Intelligence and Reasoning (LPAR), LNCS/LNAI* 4790, Springer, doi:10.1007/978-3-540-75560-9_13.

[9] Guillaume Burel (2011): *Experimenting with Deduction Modulo*. In: *Conference on Automated Deduction (CADE), LNCS/LNAI* 6803, Springer, doi:10.1007/978-3-642-22438-6_14.

[10] Guillaume Burel (2013): *A Shallow Embedding of Resolution and Superposition Proofs into the λΠ-Calculus Modulo*. In: *International Workshop on Proof Exchange for Theorem Proving (PxTP)*.

[11] Raphaël Cauderlier: *Focalide*. https://www.rocq.inria.fr/deducteam/Focalide.

[12] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand & Olivier Hermant (2013): *Proof Certification in Zenon Modulo: When Achilles Uses Deduction Modulo to Outrun the Tortoise with Shorter Steps*. In: *International Workshop on the Implementation of Logics (IWIL)*.

[13] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand & Olivier Hermant (2013): *Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo*. In: *Logic for Programming Artificial Intelligence and Reasoning (LPAR), LNCS/ARCoSS* 8312, Springer, doi:10.1007/978-3-642-45221-5_20.

[14] David Delahaye, Catherine Dubois, Claude Marché & David Mentré (2014): *The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations*. In: *Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, LNCS, Springer, doi:10.1007/978-3-662-43652-3_26.

[15] Gilles Dowek, Thérèse Hardin & Claude Kirchner (2003): *Theorem Proving Modulo*. *Journal of Automated Reasoning (JAR)* 31, doi:10.1023/A:1027357912519.

[16] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 - Where Programs Meet Provers*. In: *European Symposium on Programming (ESOP)*, doi:10.1007/978-3-642-37036-6_8.

[17] Thérèse Hardin, François Pessaux, Pierre Weis & Damien Doligez (2009): *FoCaLiZe reference manual*.

[18] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. *Journal of the ACM* 40, doi:10.1145/138027.138060.

[19] Dale Miller (2011): *A proposal for broad spectrum proof certificates*. In: *Certified Programs and Proofs (CPP)*, Springer, doi:10.1007/978-3-642-25379-9_6.

A Appendix: Shallow Embedding of LLproof$^{\equiv}$ System into Dedukti

Law of Excluded Middle and Lemmas

$ExMid(P : \mathsf{Prop}) : \Pi Z : \mathsf{Prop}.\ (\mathsf{prf}\ P \to \mathsf{prf}\ Z) \to (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ Z) \to \mathsf{prf}\ Z$

$NNPP(P : \mathsf{Prop}) : \mathsf{prf}\ (\neg\neg P) \to \mathsf{prf}\ P$
$:= \lambda H_1 : \mathsf{prf}\ (\neg\neg P).\ ExMid\ P\ P\ (\lambda H_2 : \mathsf{prf}\ P.\ H_2)\ (\lambda H_3 : \mathsf{prf}\ (\neg P).\ H_1\ H_3\ P)\ \square$

$Contr(P : \mathsf{Prop}, Q : \mathsf{Prop}) : (\mathsf{prf}\ (P \Rightarrow Q) \to \mathsf{prf}\ (\neg Q \to \neg P))$
$:= \lambda\ H_1 : \mathsf{prf}\ (P \Rightarrow Q).\ \lambda H_2 : \mathsf{prf}\ (\neg Q).\ \lambda H_3 : \mathsf{prf}\ P.\ H_2\ (H_1\ H_3)\ \square$

LLproof Inference Rules

$[]\ \mathsf{R}_{\bot} \hookrightarrow \lambda H : \mathsf{prf}\ \bot.\ H\ \square$

$[]\ \mathsf{R}_{\neg\top} \hookrightarrow \lambda H_1 : \mathsf{prf}\ (\neg\top).\ H_1\ (\lambda Z : \mathsf{Prop}.\ \lambda H_2 : \mathsf{prf}\ Z.\ H_2)\ \square$

$[P : \mathsf{Prop}]\ \mathsf{R}_{Ax}\ P \hookrightarrow \lambda H_1 : \mathsf{prf}\ P.\ \lambda H_2 : \mathsf{prf}\ (\neg P).\ H_2\ H_1\ \square$

$[\alpha : \mathsf{type}, t : \mathsf{term}\ \alpha]\ \mathsf{R}_{\neq}\ \alpha\ t \hookrightarrow \lambda H_1 : \mathsf{prf}\ (t \neq_\alpha t).\ H_1\ (\lambda z : (\mathsf{term}\ \alpha \to \mathsf{Prop}).\ \lambda H_2 : \mathsf{prf}\ (z\ t).\ H_2)\ \square$

$[\alpha : \mathsf{type}, t : \mathsf{term}\ \alpha, u : \mathsf{term}\ \alpha]\ \mathsf{R}_{Sym}\ \alpha\ t\ u \hookrightarrow \lambda H_1 : \mathsf{prf}\ (t =_\alpha u).\ \lambda H_2 : \mathsf{prf}\ (u \neq_\alpha t).\ H_2$
$(\lambda z : (\mathsf{term}\ \alpha \to \mathsf{Prop}).\ \lambda H_3 : \mathsf{prf}\ (z\ u).\ H_1\ (\lambda x : \mathsf{term}\ \alpha.\ (z\ x) \Rightarrow (z\ t))\ (\lambda H_4 : \mathsf{prf}\ (z\ t).\ H_4)\ H_3)\ \square$

$[P : \mathsf{Prop}]\ \mathsf{R}_{Cut}\ P \hookrightarrow \lambda H_1 : (\mathsf{prf}\ P \to \mathsf{prf}\ \bot).\ \lambda H_2 : (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ \bot).\ H_2\ H_1\ \square$

$[P : \mathsf{Prop}]\ \mathsf{R}_{\neg\neg}\ P \hookrightarrow \lambda H_1 : (\mathsf{prf}\ P \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\neg\neg P).\ H_2\ H_1\ \square$

$[P : \mathsf{Prop}, Q : \mathsf{Prop}]\ \mathsf{R}_{\wedge}\ P\ Q \hookrightarrow \lambda H_1 : (\mathsf{prf}\ P \to \mathsf{prf}\ Q \to \mathsf{prf}\bot).\ \lambda H_2 : \mathsf{prf}\ (P \wedge Q).\ H_2\ \bot\ H_1\ \square$

$[P : \mathsf{Prop}, Q : \mathsf{Prop}]\ \mathsf{R}_{\vee}\ P\ Q \hookrightarrow \lambda H_1 : (\mathsf{prf}\ P \to \mathsf{prf}\bot).\ \lambda H_2 : (\mathsf{prf}\ Q \to \mathsf{prf}\ \bot).\ \lambda H_3 : \mathsf{prf}\ (P \vee Q).\ H_3\ \bot\ H_1\ H_2\ \square$

$[P : \mathsf{Prop}, Q : \mathsf{Prop}]\ \mathsf{R}_{\Rightarrow}\ P\ Q \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ \bot).\ \lambda H_2 : (\mathsf{prf}\ Q \to \mathsf{prf}\ \bot).\ \lambda H_3 : \mathsf{prf}\ (P \Rightarrow Q).\ H_1$
$(Contr\ P\ Q\ H_3\ H_2)\ \square$

$[P : \mathsf{Prop}, Q : \mathsf{Prop}]\ \mathsf{R}_{\Leftrightarrow}\ P\ Q \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ (\neg Q) \to \mathsf{prf}\ \bot).\ \lambda H_2 : (\mathsf{prf}\ P \to \mathsf{prf}\ Q \to \mathsf{prf}\ \bot).$
$\lambda H_3 : \mathsf{prf}\ (P \Leftrightarrow Q).\ H_3\ \bot\ (\lambda H_4 : (\mathsf{prf}\ P \to \mathsf{prf}\ Q).\ \lambda H_5 : (\mathsf{prf}\ Q \to \mathsf{prf}\ P).\ (H_1\ (Contr\ P\ Q\ H_4$
$(\lambda H_6 : \mathsf{prf}\ Q.\ (H_2\ (H_5\ H_6))\ H_6)))\ (\lambda H_7 : \mathsf{prf}\ Q.\ (H_2\ (H_5\ H_7))\ H_7))\ \square$

$[P : \mathsf{Prop}, Q : \mathsf{Prop}]\ \mathsf{R}_{\neg\wedge}\ P\ Q \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ \bot).\ \lambda H_2 : (\mathsf{prf}\ (\neg Q) \to \mathsf{prf}\ \bot).\ \lambda H_3 : \mathsf{prf}\ (\neg(P \wedge Q)).$
$H_1\ (\lambda H_5 : \mathsf{prf}\ P.\ H_2\ (\lambda H_6 : \mathsf{prf}\ Q.\ H_3\ (\lambda Z : \mathsf{Prop}.\ \lambda H_4 : (\mathsf{prf}\ P \to \mathsf{prf}\ Q \to \mathsf{prf}\ Z).\ H_4\ H_5\ H_6)))\ \square$

$[P : \mathsf{Prop}, Q : \mathsf{Prop}]\ \mathsf{R}_{\neg\vee}\ P\ Q \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (\neg P)\mathsf{prf}\ (\neg Q) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\neg(P \vee Q)).\ H_1\ (Contr\ P\ (P \vee Q)$
$(\lambda H_3 : \mathsf{prf}\ P.\ \lambda Z : \mathsf{Prop}.\ \lambda H_4 : (\mathsf{prf}\ P \to \mathsf{prf}\ Z).\ \lambda H_5 : (\mathsf{prf}\ P \to \mathsf{prf}\ Z).\ H_4\ H_3)\ H_2)\ (Contr\ Q\ (P \vee Q)$
$(\lambda H_6 : \mathsf{prf}\ Q.\ \lambda Z : \mathsf{Prop}.\ \lambda H_7 : (\mathsf{prf}\ P \to \mathsf{prf}\ Z).\ \lambda H_8 : (\mathsf{prf}\ Q \to \mathsf{prf}\ Z).\ H_8\ H_6)\ H_2)\ \square$

$[P : \mathsf{Prop}, Q : \mathsf{Prop}]\ \mathsf{R}_{\neg\Rightarrow}\ P\ Q \hookrightarrow \lambda H_1 : (\mathsf{prf}\ P \to \mathsf{prf}\ (\neg Q) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\neg(P \Rightarrow Q)).\ H_2\ (\lambda H_3 : \mathsf{prf}\ P.$
$(H_1\ H_3)\ (\lambda H_4 : \mathsf{prf}\ Q.\ H_2\ (\lambda H_5 : \mathsf{prf}\ P.\ H_4))\ Q)\ \square$

$[P : \mathsf{Prop}, Q : \mathsf{Prop}]\ \mathsf{R}_{\neg\Leftrightarrow}\ P\ Q \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (\neg P) \to \mathsf{prf}\ (\neg Q)).\ \lambda H_2 : (\mathsf{prf}\ P \to \mathsf{prf}\ (\neg\neg Q)).$
$\lambda H_3 : \mathsf{prf}\ (\neg(P \Leftrightarrow Q)).\ (\lambda H_4 : \mathsf{prf}\ (\neg P).\ H_3\ (\lambda Z : \mathsf{Prop}.\ \lambda H_5 : (\mathsf{prf}\ (P \Rightarrow Q) \to \mathsf{prf}\ (Q \Rightarrow P) \to \mathsf{prf}\ Z).$
$H_5\ (\lambda H_6 : \mathsf{prf}\ P.\ H_4\ H_6\ Q)\ (\lambda H_7 : \mathsf{prf}\ Q.\ H_1\ H_4\ H_7\ P)))\ (\lambda H_8 : \mathsf{prf}\ P.\ H_2\ H_8\ (\lambda H_9 : \mathsf{prf}\ Q.\ H_3\ (\lambda Z : \mathsf{Prop}.$
$\lambda H_{10} : (\mathsf{prf}\ (P \Rightarrow Q) \to \mathsf{prf}\ (Q \Rightarrow P) \to \mathsf{prf}\ Z).\ H_{10}\ (\lambda H_{11} : \mathsf{prf}\ P.\ H_9)\ (\lambda H_{12} : \mathsf{prf}\ Q.\ H_8))))\ \square$

$[\alpha : \mathsf{type}, P : \mathsf{term}\ \alpha \to \mathsf{Prop}]\ \mathsf{R}_{\exists}\ \alpha\ P \hookrightarrow \lambda H_1 : (t : \mathsf{term}\ \alpha \to \mathsf{prf}\ (P\ t) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\exists\ \alpha\ P).\ H_2\ \bot\ H_1\ \square$

$[\alpha : \mathsf{type}, P : \mathsf{term}\ \alpha \to \mathsf{Prop}, t : \mathsf{term}\ \alpha]\ \mathsf{R}_{\forall}\ \alpha\ P\ t \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (P\ t) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\forall\ \alpha\ P).\ H_1\ (H_2\ t)\ \square$

$[\alpha : \mathsf{type}, P : \mathsf{term}\ \alpha \to \mathsf{Prop}, t : \mathsf{term}\ \alpha]\ \mathsf{R}_{\neg\exists}\ \alpha\ P\ t \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (\neg(P\ t)) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\neg(\exists\ \alpha\ P)).\ H_1$
$(\lambda H_4 : \mathsf{prf}\ (P\ t).\ H_2\ (\lambda Z : \mathsf{Prop}.\ \lambda H_3 : (x : \mathsf{term}\ \alpha \to \mathsf{prf}\ (P\ x) \to \mathsf{prf}\ Z).\ H_3\ t\ H_4))\ \square$

$[\alpha : \mathsf{type}, P : \mathsf{term}\ \alpha \to \mathsf{Prop}]\ \mathsf{R}_{\neg\forall}\ \alpha\ P \hookrightarrow \lambda H_1 : (t : \mathsf{term}\ \alpha \to \mathsf{prf}\ (\neg(P\ t)) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\neg(\forall\ \alpha\ P)).$
$H_2\ (\lambda t : \mathsf{term}\ \alpha.\ NNPP\ (P\ t)\ (H_1\ t))\ \square$

$[P : \mathsf{type} \to \mathsf{Prop}]\ \mathsf{R}_{\exists_{\mathsf{type}}}\ P \hookrightarrow \lambda H_1 : (\alpha : \mathsf{type} \to \mathsf{prf}\ (P\ \alpha) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\exists_{\mathsf{type}}\ P).\ H_2\ \bot\ H_1\ \square$

$[P : \mathsf{type} \to \mathsf{Prop}, \alpha : \mathsf{type}]\ \mathsf{R}_{\forall_{\mathsf{type}}}\ P\ \alpha \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (P\ \alpha) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\forall_{\mathsf{type}}\ P).\ H_1\ (H_2\ \alpha)\ \square$

$[P : \mathsf{type} \to \mathsf{Prop}, \alpha : \mathsf{type}]\ \mathsf{R}_{\neg\exists_{\mathsf{type}}}\ P\ \alpha \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (\neg(P\ \alpha)) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\neg(\exists_{\mathsf{type}}\ P)).\ H_1$
$(\lambda H_4 : \mathsf{prf}\ (P\ \alpha).\ H_2\ \lambda Z : \mathsf{Prop}.\ \lambda H_3 : (\beta : \mathsf{type} \to \mathsf{prf}\ (P\ \beta) \to \mathsf{prf}\ Z).\ H_3\ \alpha\ H_4)\ \square$

$[P : \mathsf{type} \to \mathsf{Prop}]\ \mathsf{R}_{\neg\forall_{\mathsf{type}}}\ P \hookrightarrow \lambda H_1 : (\alpha : \mathsf{type} \to \mathsf{prf}\ (\neg(P\ \alpha)) \to \mathsf{prf}\ \bot).\ \lambda H_2 : \mathsf{prf}\ (\neg(\forall_{\mathsf{type}}\ P)).\ H_2$
$(\lambda\alpha : \mathsf{type}.\ NNPP\ (P\ \alpha)\ (H_1\ \alpha))\ \square$

$[\alpha : \mathsf{type}, P : \mathsf{term}\ \alpha \to \mathsf{Prop}, t_1 : \mathsf{term}\ \alpha, t_2 : \mathsf{term}\ \alpha]\ \mathsf{R}_{Subst}\ \alpha\ P\ t_1\ t_2 \hookrightarrow \lambda H_1 : (\mathsf{prf}\ (t_1 \neq_\alpha t_2) \to \mathsf{prf}\ \bot).$
$\lambda H_2 : (\mathsf{prf}\ (P\ t_2) \to \mathsf{prf}\ \bot).\ \lambda H_3 : \mathsf{prf}\ (P\ t_1).\ H_1\ (\lambda H_4 : \mathsf{prf}\ (t_1 \neq_\alpha t_2).\ H_2\ (H_4\ P\ H_3))\ \square$

Figure 13: Shallow Embedding of LLproof into Dedukti

The deep embedding of LLproof$^{\equiv}$ presented in Sec. 4.1 is well-typed with respect to the deep em-

bedding of typed deduction modulo presented in Sec. 3.1. Using the shallow embedding presented in Sec. 3.2, we can prove all the rules declared in Fig. 8 by rewriting the $\mathsf{R}_{\mathsf{rule}}$ symbols using only one axiom: the law of excluded middle. These proofs are listed in Fig. 13 where the □ symbol is used to delimit proofs.

Translating HOL to Dedukti

Ali Assaf
Inria, Paris, France
École Polytechnique, Palaiseau, France

Guillaume Burel
ENSIIE/Cédric, Évry, France

Dedukti is a logical framework based on the $\lambda\Pi$-calculus modulo rewriting, which extends the $\lambda\Pi$-calculus with rewrite rules. In this paper, we show how to translate the proofs of a family of HOL proof assistants to Dedukti. The translation preserves binding, typing, and reduction. We implemented this translation in an automated tool and used it to successfully translate the OpenTheory standard library.

1 Introduction

Dedukti is a logical framework for defining logics and expressing proofs in those logics [8]. Following the LF legacy [17], it is based on the $\lambda\Pi$-*calculus modulo rewriting*, which extends the $\lambda\Pi$-*calculus* with rewrite rules. Cousineau and Dowek [11] showed that functional *pure type systems* (PTS), a large class of calculi that are at the basis of many proof systems, can be embedded in the $\lambda\Pi$-calculus modulo rewriting in a way that is complete and that preserves reductions (i.e. program evaluation). This led to propose Dedukti as a universal proof framework.

In this paper, we focus on translating the proofs of HOL to Dedukti. HOL refers to a family of theorem provers built on a common logical system known as *higher-order logic* or *simple type theory* [10]. It includes systems such as HOL Light, HOL4, and ProofPower-HOL. These systems are fairly popular and a large number of important mathematical results have been formalized in them [15, 16, 29].

Universal proof checking

Using Dedukti as a logical framework serves two goals. First, in the short term, it serves as an alternative, independent proof checker, providing an additional layer of confidence over each system. The second, longer term goal, is interoperability. Proof systems are becoming increasingly important, both in the formalization of mathematics and in software engineering. However, they are usually developed separately, with very little interoperability in mind. As a result, it is currently very difficult to reuse a proof from one system in another one. Embedding these different systems in a single unified framework is the first step to bring them closer together, and opens the way for theory management systems [18, 27] to combine their proofs in order to construct and verify larger theories.

The $\lambda\Pi$-calculus as a logical framework

The $\lambda\Pi$-calculus, also known as LF, is a typed λ-calculus with dependent types. Through the *Curry–Howard correspondence*, it can express a wide variety of logics [17]. Several formalizations of HOL in LF have been proposed [2, 28, 26].

C.Kaliszyk and A. Paskevich (Eds.): 4th Workshop
on Proof eXchange for Theorem Proving (PxTP'15).
EPTCS 186, 2015, pp. 74–88, doi:10.4204/EPTCS.186.8

The main concept behind this correspondence is the "*propositions as types*" principle. Typically, we define a context declaring the types, terms, and judgments of the original logic, in such a way that provability in the logic corresponds to type inhabitation in the context. For HOL, the signature would be:

$$\begin{array}{lcl}
\mathsf{type} & : & \mathsf{Type} \\
\mathsf{bool} & : & \mathsf{type} \\
\mathsf{arrow} & : & \mathsf{type} \to \mathsf{type} \to \mathsf{type} \\
\mathsf{term} & : & \mathsf{type} \to \mathsf{Type} \\
\mathsf{lam} & : & (\mathsf{term}\,\alpha \to \mathsf{term}\,\beta) \to \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta) \\
\mathsf{app} & : & \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta) \to \mathsf{term}\,\alpha \to \mathsf{term}\,\beta \\
\mathsf{proof} & : & \mathsf{term}\,\mathsf{bool} \\
\mathsf{rule_1} & : & \ldots \\
\mathsf{rule_2} & : & \ldots
\end{array}$$

For each proposition ϕ of the logic, we assign a type $\|\phi\|$ in the $\lambda\Pi$-calculus. The provability of the proposition ϕ corresponds to the inhabitation of the type $\|\phi\|$. Similarly, we translate proofs as terms inhabiting those types, and the correctness of the proof corresponds to the well-typedness of the term.

However, because the $\lambda\Pi$-calculus does not have polymorphism, we cannot translate propositions directly as types, as doing so would prevent us from quantifying over propositions for example. Instead, for each proposition ϕ, we have two translations: one translation $|\phi|$ *as a term*, and another $\|\phi\| = \mathsf{proof}\,|\phi|$ *as a type*. This correspondence has been successfully used to embed logics in the LF framework [17, 14], implemented in Twelf [25].

The $\lambda\Pi$-calculus vs. the $\lambda\Pi$-calculus modulo rewriting

An important limitation of LF is that these encodings do not preserve reduction (i.e. program evaluation), and therefore it does not preserve equivalence: if $M \equiv_\beta M'$ then $|M| \not\equiv_\beta |M'|$. For example, the term $(\lambda x : \alpha.x)\,x$ is encoded as $\mathsf{app}\,(\mathsf{lam}\,(\lambda x : \mathsf{term}\,\alpha.x))\,x$ which is not equivalent to x. This is problematic not only because it makes the representation larger and hence less efficient but also because conversion proofs may be very long.

By extending the $\lambda\Pi$-calculus with rewrite rules such as

$$\mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta) \hookrightarrow \mathsf{term}\,\alpha \to \mathsf{term}\,\beta\ ,$$

we can identify the type $\mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta)$ with the type $\mathsf{term}\,\alpha \to \mathsf{term}\,\beta$ and thus define a translation that is lighter and that preserves reductions. The encoding of the terms becomes more compact, as we represent λ-abstractions by λ-abstractions, applications by applications, etc. For example, the term $(\lambda x : \alpha.x)\,x$ is encoded as $(\lambda x : \mathsf{term}\,\alpha.x)\,x$. Such an encoding is impossible in LF for higher-order theories such as system F, HOL, or the calculus of constructions.

Moreover, our translation is modular enough so that we can extend the notion of reduction to the proofs of HOL and recover the pure type system nature of HOL [5]. This might be beneficial for several reasons:

1. It gives a reduction semantics for the proofs of HOL.
2. It allows compressing the proofs further by replacing conversion proofs with reflexivity.
3. Several other proof systems (Coq, Agda, etc.) are based on pure type systems, so expressing HOL as a PTS fits in the large scale of interoperability.

HOL and OpenTheory

The theorem provers of the HOL family (HOL Light, HOL4, ProofPower-HOL, etc.) are built on a common logical formalism known as *higher-order logic*, and have fairly similar core implementations.

A recurrent issue when trying to retrieve proofs from these systems is that they do not keep a trace of their proofs [18, 20, 24]. Following the LCF architecture, they represent their theorems using an abstract datatype and thus guarantee their safety without the need to remember their proofs. This approach reduces memory consumption but hinders their ability to share proofs.

Fortunately, several proposals have already been made to solve this problem [18, 24]. Among them is the OpenTheory project. It defines a standard format called the *article format* for recording and sharing HOL theorems. An article file contains a sequence of elementary commands to reconstruct proofs. Importing a theorem requires only a mechanical execution of the commands.

The format is limited to the HOL family, and cannot be used to communicate the proofs of Coq for example. However, it is an excellent starting point for our translation. Choosing OpenTheory as a front-end has several advantages:

- We cover all the systems of the HOL family that can export their proofs to OpenTheory with a single implementation. As of today, this includes HOL Light, HOL4, and ProofPower-HOL.[1]
- The implementation of a theorem prover can change, so the existence of this standard, documented proof format is extremely helpful, if not necessary.
- The OpenTheory project also defines a large common standard theory library, covering the development of common datatypes and mathematical theories such as lists and natural numbers. This substantial body of theories was used as a benchmark for our implementation.

Related work

Several formalizations of HOL in LF have been proposed [2, 26, 28]. To our knowledge, they lack an actual implementation of the translation. Other translations have been proposed to automatically extract the proofs of HOL to other systems such as Isabelle/HOL [19, 24], Nuprl [23], or Coq [20]. With the exception of the implementation of Kalyszyk and Krauss [19], these tools suffer from scalability problems. Our translation is lightweight enough to be scalable and provides promising results. The implementation of Kalyszyk and Krauss is the first efficient and scalable translation of HOL Light proofs, but its target is Isabelle/HOL, a system that, unlike Dedukti, is foundationally very close to HOL Light.

ProofCert [9] is another project like Dedukti that aims at providing a universal framework for checking proofs. Unlike Dedukti, it is based on sequent calculus. It can handle linear, intuitionistic, and classical logics. To our knowledge, there are no automated translations of systems like HOL to ProofCert that have been implemented yet.

A project complementary to ours is Coqine [7], which proposes a translation of the *calculus of inductive constructions* (CIC), the formalism behind Coq, to Dedukti. The translation has been implemented in an automated tool that translates the proofs compiled by Coq to Dedukti. It can handle most of the features of Coq, and has been used to translate a part of its standard library.

[1] Isabelle/HOL can currently read from but not write to OpenTheory.

Contributions

We define a translation of the types, terms and proofs of HOL to Dedukti. We use the rewriting techniques of Cousineau and Dowek [11] to obtain a shallow embedding that is lightweight and modular. We implemented this translation in an automated tool called Holide, which automatically translates the proofs of HOL written in the OpenTheory format to Dedukti. We used it to successfully translate the OpenTheory standard library.

Outline

The rest of this paper is organized as follows. Section 2 presents Dedukti and the $\lambda\Pi$-calculus modulo rewriting. Section 3 presents HOL and the logical system behind it. Section 4 defines the translation of HOL to Dedukti. In Section 5, we show that the translation is correct. Section 6 discusses the details of our implementation and the results obtained by translating the OpenTheory standard library. Section 7 discusses some additional applications of rewriting. Finally, Section 8 summarizes and considers future work.

2 Dedukti

Dedukti is essentially a type checker for the $\lambda\Pi$-calculus modulo rewriting [8], which extends the $\lambda\Pi$-calculus with rewrite rules. We choose a presentation based on pure type systems [5], which makes no syntactic distinction between terms, usually denoted by M or N, and types, usually denoted by A or B.

We assume countably infinite sets of variables and constants. There are two sorts, Type and Kind. The sort Type is the type of types and the sort Kind is the type of Type. We write $\lambda x : A.M$ for abstractions and MN for applications. The type of functions is written $\Pi x : A.B$, or $A \to B$ when x does not appear free in B. Application is left-associative while the arrow $\to$ is right-associative. Terms are considered up to α-equivalence. Contexts contain the types of variables while signatures contain the types of constants and their rewrite rules. Each rewrite rule is accompanied by a context Γ to ensure it is well-typed.

Definition 2.1. The syntax of the $\lambda\Pi$-calculus modulo rewriting is:

variables	x, y		
constants	c		
sorts	s	$::=$	$\mathsf{Type} \mid \mathsf{Kind}$
terms	M, N, A, B	$::=$	$x \mid c \mid s \mid \Pi x : A.B \mid \lambda x : A.M \mid MN$
contexts	Γ, Δ	$::=$	$\cdot \mid \Gamma, x : A$
signatures	Σ	$::=$	$\cdot \mid \Sigma, c : A \mid \Sigma, [\Gamma]\ M \rightsquigarrow N$

If R is a set of rewrite rules, we write $\longrightarrow_R$ for the induced reduction relation, $\longrightarrow_R^+$ for its transitive closure, $\longrightarrow_R^*$ for its reflexive transitive closure, and $\equiv_R$ for its reflexive symmetric transitive closure. Given a signature Σ, we write $\beta\Sigma$ for the union of the β rule with the rewrite rules of Σ.

The typing judgments $\Sigma \mid \Gamma \vdash M : A$ are accompanied by context formation judgments $\Sigma \mid \Gamma$ *context* and signature formation judgments Σ *signature*. We write $\Gamma \vdash M : A$ and Γ *context* instead of $\Sigma \mid \Gamma \vdash M : A$ and $\Sigma \mid \Gamma$ *context* when the signature is not ambiguous. The rules are presented in Figure 1.

Example 2.2. Let Σ be the signature containing

$$\alpha : \mathsf{Type}, c : \alpha, f : \alpha \to \mathsf{Type}$$

$$\frac{\Gamma \ \mathit{context} \qquad (x:A)\in\Gamma}{\Gamma\vdash x:A}\ \textsc{Var} \qquad \frac{\Gamma \ \mathit{context} \qquad (c:A)\in\Sigma}{\Gamma\vdash c:A}\ \textsc{Const} \qquad \frac{\Gamma \ \mathit{context}}{\Gamma\vdash \mathsf{Type}:\mathsf{Kind}}\ \textsc{Type}$$

$$\frac{\Gamma\vdash A:\mathsf{Type} \qquad \Gamma,x:A\vdash B:s}{\Gamma\vdash \Pi x:A.B:s}\ \textsc{Prod} \qquad \frac{\Gamma\vdash A:\mathsf{Type} \qquad \Gamma,x:A\vdash M:B}{\Gamma\vdash \lambda x:A.M:\Pi x:A.B}\ \textsc{Abs}$$

$$\frac{\Gamma\vdash M:\Pi x:A.B \qquad \Gamma\vdash N:A}{\Gamma\vdash MN:[N/x]B}\ \textsc{App} \qquad \frac{\Gamma\vdash M:A \qquad \Gamma\vdash B:\mathsf{Type} \qquad A\equiv_{\beta\Sigma} B}{\Gamma\vdash M:B}\ \textsc{Conv}$$

$$\frac{\Sigma \ \mathit{signature}}{\cdot \ \mathit{context}}\ \textsc{EmptyCtx} \qquad \frac{\Gamma\vdash A:\mathsf{Type} \qquad x\notin\Gamma}{\Gamma,x:A \ \mathit{context}}\ \textsc{VarCtx}$$

$$\frac{}{\cdot \ \mathit{signature}}\ \textsc{EmptySig} \qquad \frac{\Sigma \mid \cdot\vdash A:s \qquad c\notin\Sigma}{\Sigma,c:A \ \mathit{signature}}\ \textsc{ConstSig}$$

$$\frac{\Sigma \mid \Gamma\vdash M:A \qquad \Sigma \mid \Gamma\vdash N:A}{\Sigma,[\Gamma]\ M\rightsquigarrow N \ \mathit{signature}}\ \textsc{RewriteSig}$$

Figure 1: Typing rules of the $\lambda\Pi$-calculus

and the rewrite rule

$$[\cdot]\ f\,c \rightsquigarrow \Pi y:\alpha.\ f\,y\to f\,y.$$

The term $\lambda x:f\,c.\ x\,c\,x$ is well-typed in Σ and has the type $f\,c\to f\,c$. Notice that this term would not be well-typed without the rewrite rule, even if we replace all occurences of $f\,c$ by $\Pi y:\alpha.\ f\,y\to f\,y$.

Dedukti imposes some additional restrictions on the rewrite rules to keep type-checking decidable. In particular, the left side of a rewrite rule must belong to the higher-order pattern fragment [21, 22] and the free variables of the right side must appear on the left side. Moreover, the reduction relation $\longrightarrow_{\beta\Sigma}$ should be confluent and strongly normalizing. This property is not verified by the system and it is up to the user to ensure that it is indeed the case. We discuss this in Section 5.

3 HOL

There are many different formulations for higher-order logic. The intuitionistic formulation is based on implication and universal quantification as primitive connectives, but the current systems generally use a formulation called Q_0 [1] based on equality as a primitive connective. We take as reference the logical system used by OpenTheory [18], which we will now briefly present.

The terms of the logic are terms of the simply typed λ-calculus, with a base type bool representing the type of propositions and a type ind of individuals. The terms can contain constant symbols such as $(=)$, the symbol for equality, or select, the symbol of choice. The logic supports a restricted form of polymorphism, known as ML-style polymorphism, by allowing type variables, such as α or β, to appear in types. For example, the type of $(=)$ is $\alpha\to\alpha\to\mathsf{bool}$.

$$\frac{}{\vdash M = M}\ \textsc{Refl}\ M \qquad \frac{\Gamma \vdash M = N}{\Gamma \vdash \lambda x : A.M = \lambda x : A.N}\ \textsc{AbsThm}\ x \qquad \frac{\Gamma \vdash F = G \qquad \Delta \vdash M = N}{\Gamma \cup \Delta \vdash FM = GN}\ \textsc{AppThm}$$

$$\frac{}{\vdash (\lambda x : A.M)\, x = M}\ \textsc{Beta}\ x\ M \qquad \frac{}{\{\phi\} \vdash \phi}\ \textsc{Assume} \qquad \frac{\Gamma \vdash \phi = \psi \qquad \Delta \vdash \phi}{\Gamma \cup \Delta \vdash \psi}\ \textsc{EqMp}$$

$$\frac{\Gamma \vdash \phi \qquad \Delta \vdash \psi}{(\Gamma - \{\psi\}) \cup (\Delta - \{\phi\}) \vdash \phi = \psi}\ \textsc{DeductAntiSym} \qquad \frac{\Gamma \vdash \phi}{\Gamma[\sigma] \vdash \phi[\sigma]}\ \textsc{Subst}\ \sigma$$

Figure 2: Derivation rules of HOL

Types can be parameterized through type operators of the form $p(A_1, \ldots, A_n)$. For example, list is a type operator of arity 1, and $\mathsf{list}(\mathsf{bool})$ is the type of lists of booleans. Type variables and type operators are enough to describe all the types of HOL, because bool can be seen as a type operator of arity 0, and the arrow $\to$ as a type operator of arity 2. Hence the type of $(=_\alpha)$ is in fact $\to(\alpha, \to(\alpha, \mathsf{bool}()))$. We still write $A \to B$ instead of $\to(A,B)$ for arrow types, p instead of $p()$ for type operators of arity 0, and $M = N$ instead of $(=)MN$ when it is more convenient.

Definition 3.1. The syntax of HOL is:

type variables	α, β		
type operators	p		
types	A, B	$::=$	$\alpha \mid p(A_1, \ldots, A_n)$
term variables	x, y		
term constants	c		
terms	M, N	$::=$	$x \mid \lambda x : A.M \mid MN \mid c$

The propositions of the logic are the terms of type bool and the predicates are the terms of type $A \to \mathsf{bool}$. We use letters such as ϕ or ψ to denote propositions. The contexts, denoted by Γ or Δ, are sets of propositions, and the judgments of the logic are of the form $\Gamma \vdash \phi$. The derivation rules are presented in Figure 2.

Example 3.2. Here is a derivation of the transitivity of equality: if $\Gamma \vdash x = y$ and $\Delta \vdash y = z$, then $\Gamma \cup \Delta \vdash x = z$.

$$\dfrac{\dfrac{\dfrac{}{\vdash ((=)x) = ((=)x)}\ \textsc{Refl} \qquad \Delta \vdash y = z}{\Delta \vdash (x = y) = (x = z)}\ \textsc{AppThm} \qquad \Gamma \vdash x = y}{\Gamma \cup \Delta \vdash x = z}\ \textsc{EqMp}$$

HOL supports mechanisms for defining new types and constants in a conservative way. We will not consider them here. In addition to the core derivation rules, three axioms are assumed:

- η-equality, which states that $\lambda x : A.\, Mx = M$,
- the axiom of choice, with a predeclared symbol of choice called select,
- the axiom of infinity, which states that the type ind is infinite.

It is important to note that from η-convertibility and the axiom of choice, we can derive the excluded middle [6], making HOL a classical logic.

4 Translation

In this section we show how to translate HOL to Dedukti. We define a signature Σ containing primitive declarations and definitions, and a translation function assigning, to every construct of the logic, a term that is well-typed in the signature Σ.

HOL Types

To translate the simple types of HOL, we declare a new Dedukti type called type and three constructors bool, ind and arrow.

$$\begin{array}{lcl} \mathsf{type} & : & \mathsf{Type} \\ \mathsf{bool} & : & \mathsf{type} \\ \mathsf{ind} & : & \mathsf{type} \\ \mathsf{arrow} & : & \mathsf{type} \to \mathsf{type} \to \mathsf{type} \end{array}$$

One should not confuse type, which is the type of Dedukti terms that represent HOL types, with Type, which is the type of Dedukti types. The translation of a HOL type as a Dedukti term is defined inductively on the structure of the type.

Definition 4.1 (Translation of a HOL type as a Dedukti term)**.** For any HOL type A, we define $|A|$, the translation of A as a term, to be

$$\begin{array}{lcl} |\alpha| & = & \alpha \\ |\mathsf{bool}| & = & \mathsf{bool} \\ |\mathsf{ind}| & = & \mathsf{ind} \\ |A \to B| & = & \mathsf{arrow}\,|A|\,|B| \ . \end{array}$$

More generally, if we have an n-ary HOL type operator p, we declare a constant p of type $\underbrace{\mathsf{type} \to \ldots \to \mathsf{type}}_{n} \to \mathsf{type}$, and we translate an instance $p\,(A_1, \ldots, A_n)$ of this type operator to the term $p\,|A_1| \cdots |A_n|$.

HOL Terms

We declare a new dependent type called term indexed by a type, and we identify the terms of type $\mathsf{term}(\mathsf{arrow}\,A\,B)$ with the functions of type $\mathsf{term}\,A \to \mathsf{term}\,B$ by adding a rewrite rule. We also declare a constant eq for HOL equality and a constant select for the choice operator.

$$\begin{array}{lcl} \mathsf{term} & : & \mathsf{type} \to \mathsf{Type} \\ \mathsf{eq} & : & \Pi\alpha : \mathsf{type}.\ \mathsf{term}\,(\mathsf{arrow}\,\alpha\,(\mathsf{arrow}\,\alpha\,\mathsf{bool})) \\ \mathsf{select} & : & \Pi\alpha : \mathsf{type}.\ \mathsf{term}\,(\mathsf{arrow}\,(\mathsf{arrow}\,\alpha\,\mathsf{bool})\,\alpha) \end{array}$$

$$[\alpha : \mathsf{type}, \beta : \mathsf{type}]\ \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta) \rightsquigarrow \mathsf{term}\,\alpha \to \mathsf{term}\,\beta$$

The symbol term can be seen as a decoding function that assigns a Dedukti type to every HOL type. The translation of a term M of type A will then be a term of type $\mathsf{term}\,|A|$.

Definition 4.2 (Translation of a HOL type as a Dedukti type)**.** For any HOL type A, we define

$$\|A\| = \mathsf{term}\,|A| \,.$$

Definition 4.3 (Translation of a HOL term as a Dedukti term)**.** For any HOL term M, we define $|M|$, the translation of M as a term to be

$$\begin{array}{lcl} |x| & = & x \\ |MN| & = & |M|\ |N| \\ |\lambda x : A.M| & = & \lambda x : \|A\|\ .\ |M| \\ |(=_A)| & = & \mathsf{eq}\,|A| \\ |\mathsf{select}_A| & = & \mathsf{select}\,|A|\ . \end{array}$$

More generally, for every HOL constant c of type A, if $\alpha_1,\ldots,\alpha_n$ are the free type variables that appear in A, we declare a new constant c of type

$$\Pi\alpha_1 : \mathsf{type}.\ \ldots \Pi\alpha_n : \mathsf{type}.\ \|A\|$$

and we translate an instance $c_{A_1,\ldots,A_n}$ of this constant by the term $c\,|A_1|\cdots|A_n|$.

Example 4.4. The term $(\lambda x : \alpha.x)\,x$ is translated to

$$|(\lambda x : \alpha.x)\,x| \quad = \quad (\lambda x : \mathsf{term}\,\alpha.x)\,x$$

which is convertible to x.

HOL Proofs

We declare a new type proof, to express the proof judgments of HOL. It is a dependent type, indexed by the proposition ϕ that it is proving.

$$\mathsf{proof} \quad : \quad \mathsf{term}\,\mathsf{bool} \to \mathsf{Type}$$

Definition 4.5 (Translation of HOL propositions as Dedukti types)**.** For any HOL proposition ϕ (i.e. a HOL term of type bool), we define

$$\|\phi\| = \mathsf{proof}\,|\phi|\,.$$

For any HOL context $\Gamma = \phi_1,\ldots,\phi_n$, we define

$$\|\Gamma\| = h_{\phi_1} : \|\phi_1\|, \ldots, h_{\phi_n} : \|\phi_n\|$$

where $h_{\phi_1},\ldots,h_{\phi_n}$ are fresh variables.

We now take care of the derivation rules of HOL (Figure 2). In the following, we write $\Pi x,y : A.B$ as a shortcut for $\Pi x : A.\Pi y : A.B$.

Equality proofs

We declare Refl, FunExt, and AppThm:

$$\begin{array}{lcl} \mathsf{Refl} & : & \Pi\alpha : \mathsf{type}.\,\Pi x : \mathsf{term}\,\alpha.\ \mathsf{proof}\,(\mathsf{eq}\,\alpha\,x\,x) \\ \mathsf{FunExt} & : & \Pi\alpha,\beta : \mathsf{type}.\,\Pi f,g : \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta)\,. \\ & & (\Pi x : \mathsf{term}\,\alpha.\ \mathsf{proof}\,(\mathsf{eq}\,\beta\,(f\,x)\,(g\,x))) \to \mathsf{proof}\,(\mathsf{eq}\,(\mathsf{arrow}\,\alpha\,\beta)\,f\,g) \\ \mathsf{AppThm} & : & \Pi\alpha,\beta : \mathsf{type}.\,\Pi f,g : \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta)\,.\,\Pi x,y : \mathsf{term}\,\alpha. \\ & & \mathsf{proof}\,(\mathsf{eq}\,(\mathsf{arrow}\,\alpha\,\beta)\,f\,g) \to \mathsf{proof}\,(\mathsf{eq}\,\alpha\,x\,y) \to \mathsf{proof}\,(\mathsf{eq}\,\beta\,(f\,x)\,(g\,y)) \end{array}$$

The constant FunExt corresponds to *functional extensionality*, which states that if two functions f and g of type $A \to B$ are equal on all values x of type A, then f and g are equal. We can use it to translate both the ABSTHM rule and the η axiom. Finally, since our encoding is shallow, β-equality can be proved by reflexivity.

Definition 4.6. The rules REFL, ABSTHM, APPTHM, and BETA are translated to

$$\left|\frac{}{\vdash M = M}\ \textsc{Refl}\right| = \mathsf{Refl}\,|A|\ |M| \quad \text{(where } A \text{ is the type of } M\text{)}$$

$$\left|\frac{\mathscr{D}}{\Gamma \vdash \lambda x : A.M = \lambda x : A.N}\ \textsc{AbsThm}\right| = \mathsf{FunExt}\,|A|\ |B|\ |\lambda x : A.M|\ |\lambda x : A.N|\ (\lambda x : |A|\,.\,|\mathscr{D}|)$$

$$\left|\frac{\mathscr{D}_1 \qquad \mathscr{D}_2}{\Gamma \cup \Delta \vdash F\,M = G\,N}\ \textsc{AppThm}\right| = \mathsf{AppThm}\,|A|\ |B|\ |F|\ |G|\ |M|\ |N|\ |\mathscr{D}_1|\ |\mathscr{D}_2|$$

$$\left|\frac{}{(\lambda x : A.M)\,x = M}\ \textsc{Beta}\right| = \mathsf{Refl}\,|B|\ |M| \quad \text{(where } B \text{ is the type of } M\text{)}\ .$$

Boolean proofs

We declare the constants PropExt and EqMp:

$$\begin{array}{lcl} \mathsf{PropExt} & : & \Pi p, q : \mathsf{term\,bool}. \\ & & (\mathsf{proof}\,q \to \mathsf{proof}\,p) \to (\mathsf{proof}\,q \to \mathsf{proof}\,p) \to \mathsf{proof}\,(\mathsf{eq\,bool}\,p\,q) \\ \mathsf{EqMp} & : & \Pi p, q : \mathsf{term\,bool}.\ \mathsf{proof}\,(\mathsf{eq\,bool}\,p\,q) \to \mathsf{proof}\,p \to \mathsf{proof}\,q \end{array}$$

The constant PropExt corresponds to *propositional extensionality* and, together with EqMp, states that equality on booleans in HOL behaves like the connective "*if and only if*".

Definition 4.7. The rules ASSUME, DEDUCTANTISYM, and EQMP are translated to

$$\left|\frac{}{\{\phi\} \vdash \phi}\ \textsc{Assume}\right| = h_\phi \quad \text{(where } h_\phi \text{ is a fresh variable)}$$

$$\left|\frac{\mathscr{D}_1 \qquad \mathscr{D}_2}{(\Gamma - \{\psi\}) \cup (\Delta - \{\phi\}) \vdash \phi = \psi}\ \textsc{DeductAntiSym}\right| =$$

$$\mathsf{PropExt}\,|\phi|\ |\psi|\ \left(\lambda h_\psi : \|\psi\|\,.\,|\mathscr{D}_1|\right)\ \left(\lambda h_\phi : \|\phi\|\,.\,|\mathscr{D}_2|\right)$$

$$\left|\frac{\mathscr{D}_1 \qquad \mathscr{D}_2}{\Gamma \cup \Delta \vdash \psi}\ \textsc{EqMp}\right| = \mathsf{EqMp}\,|\phi|\ |\psi|\ |\mathscr{D}_1|\ |\mathscr{D}_2|\ .$$

Substitution proofs

The HOL rule SUBST derives $\Gamma[\sigma] \vdash \phi[\sigma]$ from $\Gamma \vdash \phi$. In OpenTheory, the substitution can substitute for both term and type variables but type variables are instantiated first. For the sake of clarity, we split this rule in two steps: one for term substitution of the form $\sigma = M_1/x_1, \ldots, M_n/x_n$, and one for type substitution of the form $\theta = A_1/\alpha_1, \ldots, A_m/\alpha_m$. In Dedukti, we have to rely on β-reduction to express substitution. We can correctly translate a parallel substitution $M[M_1/x_1, \ldots, M_n/x_n]$ as

$$(\lambda x_1 : B_1. \ldots \lambda x_n : B_n.M)\,M_1 \ldots M_n$$

where B_i is the type of M_i.

Definition 4.8. The rule SUBST is translated to

$$\left|\frac{\mathscr{D}}{\Gamma[\theta] \vdash \phi[\theta]}\ \textsc{TypeSubst}\right| = (\lambda\, \alpha_1 : \mathsf{type}. \ldots \lambda\, \alpha_m : \mathsf{type}.\ |\mathscr{D}|)\, |A_1| \ldots |A_m|$$

$$\left|\frac{\mathscr{D}}{\Gamma[\sigma] \vdash \phi[\sigma]}\ \textsc{TermSubst}\right| = (\lambda x_1 : \|B_1\| . \ldots \lambda x_n : \|B_n\| .\ |\mathscr{D}|)\, |M_1| \ldots |M_n|$$

5 Correctness

The correctness of the translation is expressed by two properties: *completeness* and *soundness*. The first states that all the generated terms have the correct type. For example, the translation of a term of type A has type $\|A\|$ while the translation of a proof of ϕ has type $\|\phi\|$. The second states that if a proof term is well-typed in Dedukti, then the proof is correct in the original logic. These two properties ensure that we can use Dedukti as an independent proof checker: we can use it to re-verify the proofs of OpenTheory, and moreover we can be sure that, if a proof is accepted by Dedukti, then it is also valid in OpenTheory.

Completeness

Let Σ be the signature of HOL containing the declarations and rewrite rules of the previous sections.

Lemma 5.1. *For any HOL type A,*

$$\Sigma \mid \alpha_1 : \mathsf{type}, \ldots, \alpha_n : \mathsf{type} \vdash |A| : \mathsf{type}$$

where $\alpha_1, \ldots, \alpha_n$ are the free type variables appearing in A.

Lemma 5.2. *For any HOL term M of type A,*

$$\Sigma \mid \alpha_1 : \mathsf{type}, \ldots, \alpha_n : \mathsf{type}, x_1 : \|A_1\|, \ldots x_n : \|A_n\| \vdash |M| : \|A\|$$

where $\alpha_1, \ldots, \alpha_n$ are the free type variables and $x_1 : A_1, \ldots, x_n : A_n$ are the free term variables appearing in M.

Theorem 5.3. *For any HOL proof $\mathscr{D}$ of $\Gamma \vdash \phi$,*

$$\Sigma \mid \alpha_1 : \mathsf{type}, \ldots, \alpha_n : \mathsf{type}, x_1 : \|A_1\|, \ldots x_n : \|A_n\|, \|\Gamma\| \vdash |\mathscr{D}| : \|\phi\|$$

where $\alpha_1, \ldots, \alpha_n$ are the free type variables and $x_1 : A_1, \ldots, x_n : A_n$ are the free term variables appearing in $\mathscr{D}$.

Proof. By induction on the structure of $\mathscr{D}$. □

Soundness

Proving the soundness of the embedding is less straightforward than proving completeness. In fact, it is closely related to the confluence and normalization properties of the system. We state the results here and refer the reader to the works of Assaf, Cousineau, and Dowek [3, 11, 12] for the complete proofs.[2]

Lemma 5.4. *The reduction relation $\longrightarrow_{\beta\Sigma}$ is confluent.*

Lemma 5.5. *The reduction relation $\longrightarrow_{\beta\Sigma}$ is strongly normalizing.*

Theorem 5.6. *If $\Sigma \mid \|\Gamma\| \vdash M : \|A\|$ then M corresponds to a valid proof of $\Gamma \vdash A$ in HOL.*

[2] The terms *soundness* and *completeness* are interchanged in Cousineau and Dowek's paper [11].

Package	Size (kB)		Time (s)	
	OpenTheory	Dedukti	Translation	Verification
unit	5	13	0.2	0.0
function	16	53	0.3	0.2
pair	38	121	0.8	0.5
bool	49	154	0.9	0.5
sum	84	296	2.1	1.1
option	93	320	2.2	1.2
relation	161	620	4.6	2.8
list	239	827	5.7	3.2
real	286	945	6.5	3.1
natural	343	1065	6.8	3.2
set	389	1462	10.2	5.8
Total	1702	5877	40.3	21.6

Table 1: Translation of the OpenTheory standard library

6 Implementation

We implemented our translation in an automated tool called Holide. It works as an OpenTheory virtual machine that additionally keeps track of the corresponding proof terms for theorems. The program reads a HOL proof written in the OpenTheory article format (`.art`) and outputs a Dedukti file (`.dk`) containing its translation. We can run Dedukti on the generated file to verify it. All generated files are linked with a hand-written file `hol.dk` containing the signature Σ that we defined in Section 4. Our software is available online at https://www.rocq.inria.fr/deducteam/Holide/.

HOL proofs are known to be very large [19, 20, 24], and we needed to implement sharing of proofs, terms, and types in order to reduce them to a manageable size. OpenTheory already provides some form of proof sharing but we found it easier to completely factorize the derivations into individual steps.

We used Holide to translate the OpenTheory standard library. The library is organized into logical packages, each corresponding to a theory such as lists or natural numbers. We were able to verify all of the generated files. The results are summarized in Table 1. We list the size of both the source files and the files generated by the translation after compression using gzip. The reason we use the size of the compressed files for comparison is because it provides a more reasonable measure that is less affected by syntax formatting and whitespace. We also list the time it takes to translate and verify each package. These tests were done on a 64-bit Intel Xeon(R) CPU @ 2.67GHz × 4 machine with 4 GB of RAM.

Overall, the size of the generated files is about 3 to 4 times larger than the source files. Given that this is an encoding in a logical framework, an increase in the size is to be expected, and we find that this factor is very reasonable. There are no similar translations to compare to except the one of Keller and Werner [20]. The comparison is difficult because they work with a slightly different form of input, but they produce several hundred megabytes of proofs. Similary, an increase in verification time is to be expected compared to verifying OpenTheory directly, but our results are still very reasonable given the nature of the translation. Our time is about 4 times larger than OpenTheory, which takes about 5 seconds to verify the standard library. It is in line with the scalable translation of Kalyszyk and Krauss to Isabelle/HOL, which takes around 30 seconds [19]. In comparison, Keller and Werner's translation takes several hours, although we should note that our work greatly benefited from their experience.

7 Extensions

In this section we show some additional advantages of having a translation which preserves reduction.

Compressing conversion proofs

One of the reasons why HOL proofs are so large is that conversion proofs have to traverse the terms using the congruence rules AbsThm and AppThm. Since we now prove β-reduction using reflexivity, large conversion proofs could be reduced to a single reflexivity step, therefore reducing the size of the proofs.[3]

Example 7.1. The following proof of $f(g((\lambda x : A.x)x)) = f(g(x))$,

$$\dfrac{\dfrac{}{\vdash f = f}\ \textsc{Refl}\ f \qquad \dfrac{\dfrac{}{\vdash g = g}\ \textsc{Refl}\ g \qquad \dfrac{}{\vdash (\lambda x : A.x)x = x}\ \textsc{Beta}}{\vdash g((\lambda x : A.x)x) = gx}\ \textsc{AppThm}}{\vdash f(g((\lambda x : A.x)x)) = f(gx)}$$

can be translated simply as $\mathsf{Refl}\, C(f(gx))$, where $A \to B$ is the type of g and $B \to C$ is the type of f.

HOL as a pure type system

It turns out that HOL can be seen as a pure type system called λ_{HOL} with three sorts [5, 13]. This formulation corresponds to intuitionistic higher-order logic. However, this structure is lost in the Q_0 formulation used by the HOL systems. Our shallow embedding can be adapted to recover this structure, and thus obtain a constructive and computational version of HOL.

Instead of equality, we declare implication and universal quantification as primitive connectives, and we define what provability means through rewriting.

$$\begin{array}{lcl} \mathsf{imp} & : & \mathsf{term}\,(\mathsf{arrow}\,\mathsf{bool}\,(\mathsf{arrow}\,\mathsf{bool}\,\mathsf{bool})) \\ \mathsf{forall} & : & \Pi\alpha : \mathsf{type}.\ \mathsf{term}\,(\mathsf{arrow}\,(\mathsf{arrow}\,\alpha\,\mathsf{bool})\,\mathsf{bool}) \end{array}$$

$$\begin{array}{llcl} [p : \mathsf{term}\,\mathsf{bool}, q : \mathsf{term}\,\mathsf{bool}] & \mathsf{proof}\,(\mathsf{imp}\,p\,q) & \rightsquigarrow & \mathsf{proof}\,p \to \mathsf{proof}\,q \\ [\alpha : \mathsf{type}, p : \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\mathsf{bool})] & \mathsf{proof}\,(\mathsf{forall}\,p) & \rightsquigarrow & \Pi x : \mathsf{term}\,\alpha.\ \mathsf{proof}\,(p\,x) \end{array}$$

However, this time we do not even need to declare constants like Refl and AppThm for the derivation rules, because they are derivable. Here is a derivation of the introduction and elimination rules for implication for example:

$$\begin{array}{lcl} \mathsf{imp\_intro} & : & \Pi p, q : \mathsf{term}\,\mathsf{bool}.\ (\mathsf{proof}\,p \to \mathsf{proof}\,q) \to \mathsf{proof}\,(\mathsf{imp}\,p\,q) \\ & = & \lambda p, q : \mathsf{term}\,\mathsf{bool}.\,\lambda h : (\mathsf{proof}\,p \to \mathsf{proof}\,q).\,h \\ \mathsf{imp\_elim} & : & \Pi p, q : \mathsf{term}\,\mathsf{bool}.\ \mathsf{proof}\,(\mathsf{imp}\,p\,q) \to \mathsf{proof}\,p \to \mathsf{proof}\,q \\ & = & \lambda p, q : \mathsf{term}\,\mathsf{bool}.\,\lambda h : \mathsf{proof}\,(\mathsf{imp}\,p\,q).\,\lambda x : \mathsf{proof}\,p.\ hx \end{array}$$

By translating the introduction rules as λ-abstractions, and the elimination rules as applications, we recover the reduction of the proof terms, which corresponds to *cut elimination* in the original proofs.

[3] This also applies to conversions involving constant definitions, which we did not cover here but are also assumed as an axiom in HOL.

As for equality, it is also possible to define it in terms of these connectives. For example, we could use the Leibniz definition of equality, which is the one used by Coq:

$$\begin{array}{rcl} \mathsf{eq} & : & \Pi\alpha : \mathsf{type}.\ \mathsf{term}\,(\mathsf{arrow}\,\alpha\,(\mathsf{arrow}\,\alpha\,\mathsf{bool})) \\ & = & \lambda\alpha : \mathsf{type}.\,\lambda x : \mathsf{term}\,\alpha.\,\lambda y : \mathsf{term}\,\alpha. \\ & & \mathsf{forall}\,(\mathsf{arrow}\,\alpha\,\mathsf{bool})\ (\Pi p : \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\mathsf{bool}).\ \mathsf{imp}\,(p\,x)\ (p\,y)) \end{array}$$

We would still need to assume some axioms to prove all the rules of OpenTheory, namely FunExt and PropExt [20], but at least this definition is closer to that of Coq. Since the λ_{HOL} PTS is a strict subset of the calculus of inductive constructions, we can adapt our translation to inject HOL directly into an embedding of Coq in Dedukti [7], or to combine HOL proofs with Coq proofs in Dedukti [4]. Further research into ways to eliminate these axioms (and thus maintain the constructive aspect) when possible is the subject of ongoing work.

8 Conclusion

We showed how to translate HOL to Dedukti by adapting techniques from Cousineau and Dowek [11] to define an embedding that is sound, complete, and reduction preserving. Using our implementation, we were able to translate the OpenTheory standard library and verify it in Dedukti.

Future work

The translation we have presented can be improved in several ways. The current implementation suffers from a lack of linking: if a package makes use of a type, constant, or theorem defined in another package, we do not have a reference to the original definition. This is due to a limitation of the OpenTheory article format. In OpenTheory, this problem is resolved by adding a theory management layer, which is responsible for composing and linking theories together [18]. It would be beneficial to integrate this layer in our translation so that we can properly link the resulting files together.

While we used several optimizations including term sharing in our implementation, there is still room for reducing the time and memory consumption of the translation and the size of the generated files. The caching techniques of Kaliszyk and Krauss [19] could be used in this regard to handle larger libraries and formalizations.

Finally, we can study how to combine the proofs obtained by this translation with the proofs obtained from the translation of Coq [7]. That will require a careful examination of the compatibility of the two embeddings. First, the types of the two theories must coincide, so that a natural number from HOL is the same as a natural number from Coq for example. Second, we must make sure that the resulting theory is consistent. For instance, we know that every type in HOL is inhabited, which is inconsistent with the existence of empty types in Coq, so we will need to modify the translations to avoid this. A solution is to parameterize each HOL type variable by a witness ensuring that it is non-empty. Our translation can be adapted for this solution without much trouble. Some work has already been done in this direction [4].

Acknowledgments

We thank Gilles Dowek for his support, as well as Mathieu Boespflug and Chantal Keller for their comments and suggestions.

References

[1] Peter B. Andrews (1986): *An introduction to mathematical logic and type theory: to truth through proof*. Academic Press Professional, Inc., San Diego, CA, USA.

[2] Andrew W. Appel (2001): *Foundational Proof-Carrying Code*. In: *LICS*, IEEE Computer Society, Washington, DC, USA, p. 247–256, doi:10.1109/LICS.2001.932501.

[3] Ali Assaf (2015): *Conservativity of embeddings in the lambda-Pi calculus modulo rewriting*. Available at https://hal.archives-ouvertes.fr/hal-01084165. To appear in TLCA 2015.

[4] Ali Assaf & Raphaël Cauderlier (2015): *Mixing HOL and Coq in Dedukti (Rough Diamond)*. Available at https://hal.inria.fr/hal-01141789. To appear in PxTP 2015.

[5] H. P. Barendregt (1992): *Lambda Calculi with Types, Handbook of Logic in Computer Science Vol. II*. Oxford University Press.

[6] Michael Beeson (1985): *Foundations of Constructive Mathematics*. Springer-Verlag, doi:10.1007/978-3-642-68952-9.

[7] M. Boespflug & G. Burel (2012): *CoqInE: Translating the calculus of inductive constructions into the lambda-Pi-calculus modulo*. In: *PxTP*, pp. 44–50.

[8] M. Boespflug, Q. Carbonneaux & O. Hermant (2012): *The lambda-Pi-calculus modulo as a universal proof language*. In: *PxTP*, pp. 28–43.

[9] Zakaria Chihani, Dale Miller & Fabien Renaud (2013): *Foundational proof certificates in first-order Logic*. In Maria Paola Bonacina, editor: *Automated Deduction – CADE-24*, *Lecture Notes in Computer Science* 7898, Springer Berlin Heidelberg, pp. 162–177, doi:10.1007/978-3-642-38574-2_11.

[10] Alonzo Church (1940): *A formulation of the simple theory of types*. *Journal of Symbolic Logic* 5(02), pp. 56–68, doi:10.2307/2266170.

[11] Denis Cousineau & Gilles Dowek (2007): *Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo*. In Simona Ronchi Della Rocca, editor: *TLCA*, *LNCS* 4583, Springer Berlin Heidelberg, pp. 102–117, doi:10.1007/978-3-540-73228-0_9.

[12] Gilles Dowek (2014): *Models and termination of proof-reduction in the $\lambda\Pi$-calculus modulo theory*. Available at https://who.rocq.inria.fr/Gilles.Dowek/Publi/superpi.pdf.

[13] Herman Geuvers (1993): *Logics and type systems*. PhD thesis, University of Nijmegen.

[14] Herman Geuvers & Erik Barendsen (1999): *Some logical and syntactical observations concerning the first-order dependent type system lambda-P*. *Mathematical Structures in Computer Science* 9(04), pp. 335–359, doi:10.1017/S0960129599002856.

[15] Thomas C. Hales (2007): *The Jordan Curve Theorem, Formally and Informally*. *American Mathematical Monthly* 114(10), pp. 882–894.

[16] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua & Roland Zumkeller (2011): *A Revision of the Proof of the Kepler Conjecture*. In Jeffrey C. Lagarias, editor: *The Kepler Conjecture*, Springer New York, pp. 341–376, doi:10.1007/978-1-4614-1129-1_9.

[17] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A framework for defining logics*. *J. ACM* 40(1), p. 143–184, doi:10.1145/138027.138060.

[18] Joe Hurd (2011): *The OpenTheory Standard Theory Library*. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *NFM*, *LNCS* 6617, Springer, pp. 177–191, doi:10.1007/978-3-642-20398-5_14.

[19] Cezary Kaliszyk & Alexander Krauss (2013): *Scalable LCF-Style Proof Translation*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *ITP*, *LNCS* 7998, Springer Berlin Heidelberg, pp. 51–66, doi:10.1007/978-3-642-39634-2_7.

[20] Chantal Keller & Benjamin Werner (2010): *Importing HOL Light into Coq*. In Matt Kaufmann & Lawrence C. Paulson, editors: *ITP*, *LNCS* 6172, Springer Berlin Heidelberg, pp. 307–322, doi:10.1007/978-3-642-14052-5_22.

[21] Dale Miller (1991): *Unification of simply typed lambda-terms as logic programming*. *Technical Reports (CIS)*.

[22] Dale A. Miller (2004): *Proofs in higher-order logic*. Ph.D. thesis, University of Pennsylvania.

[23] Pavel Naumov, Mark-Oliver Stehr & José Meseguer (2001): *The HOL/NuPRL Proof Translator*. In Richard J. Boulton & Paul B. Jackson, editors: *TPHOLs*, *LNCS* 2152, Springer Berlin Heidelberg, pp. 329–345, doi:10.1007/3-540-44755-5_23.

[24] Steven Obua & Sebastian Skalberg (2006): *Importing HOL into Isabelle/HOL*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning*, *LNCS* 4130, Springer Berlin Heidelberg, pp. 298–302, doi:10.1007/11814771_27.

[25] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In: *CADE-16*, *LNCS* 1632, Springer Berlin Heidelberg, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[26] Florian Rabe (2010): *Representing Isabelle in LF*. *EPTCS* 34, pp. 85–99, doi:10.4204/EPTCS.34.8. arXiv: 1009.2794.

[27] Florian Rabe & Michael Kohlhase (2013): *A scalable module system*. *Inf. Comput.* 230, pp. 1–54, doi:10.1016/j.ic.2013.06.001.

[28] Carsten Schürmann & Mark-Oliver Stehr (2006): *An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf*. In Miki Hermann & Andrei Voronkov, editors: *LPAR*, *LNCS* 4246, Springer Berlin Heidelberg, pp. 150–166, doi:10.1007/11916277_11.

[29] Freek Wiedijk (2007): *The QED manifesto revisited*. *Studies in Logic, Grammar and Rhetoric* 10(23), pp. 121–133.

Mixing HOL and Coq in Dedukti (Extended Abstract)

Ali Assaf
INRIA Paris-Rocquencourt, France
École Polytechnique, France

Raphaël Cauderlier
INRIA Paris-Rocquencourt, France
Laboratoire CEDRIC, CNAM, France

We use Dedukti as a logical framework for interoperability. We use automated tools to translate different developments made in HOL and in Coq to Dedukti, and we combine them to prove new results. We illustrate our approach with a concrete example where we instantiate a sorting algorithm written in Coq with the natural numbers of HOL.

1 Introduction

Interoperability is an emerging problem in the world of proof systems. Interactive theorem provers are developed independently and cannot usually be used together effectively. The theorems of one system can rarely be used in another, and it can be very expensive to redo the proofs manually. Obstacles for a large-scale interoperability are many, ranging from differences in the logical theory and the representation of data types, to the lack of a standard and effective way of retrieving proofs. For systems based on a common logical formalism, exchange formats for proofs have appeared like the TPTP derivation format [26] for traces of automated first-order theorem provers and OpenTheory [17] for HOL interactive theorem provers. However, combining systems working in different logical theories is harder.

A solution to this problem is to use a logical framework. The idea is to have a small and simple language that is expressive and flexible enough to define various logics and to faithfully express proofs in those logics, at a relatively low cost. Translating all the different systems to this common framework is a first step in bringing them closer together. This is the idea behind LF [14], implemented in Twelf [22], which has been used as a framework for interoperability in various projects [25, 16].

We propose to use a variant of Twelf called Dedukti. The reason for using Dedukti is that it implements an extension of LF called the $\lambda\Pi$-*calculus modulo rewriting* [5, 10], which adds term rewriting to the calculus. This extension not only allows for a more compact representation of proofs, but also enables the encoding of richer theories, such as the calculus of constructions. This cannot be done in LF efficiently because computation would have to be represented as a relation and every conversion made explicit. We thus use Dedukti as our logical framework.

Several tools have been developed to translate the proofs of various systems to Dedukti [6, 3, 11, 8]. The translations are based on the encodings of Cousineau and Dowek in the $\lambda\Pi$-calculus modulo rewriting [10]. The proofs, represented as terms of the $\lambda\Pi$-calculus modulo rewriting, can be checked independently by Dedukti, adding another layer of confidence over the original systems. This approach has been successfully used to verify the formalization of several libraries and the proof traces of theorem provers on large problem sets (of the order of several gigabytes).

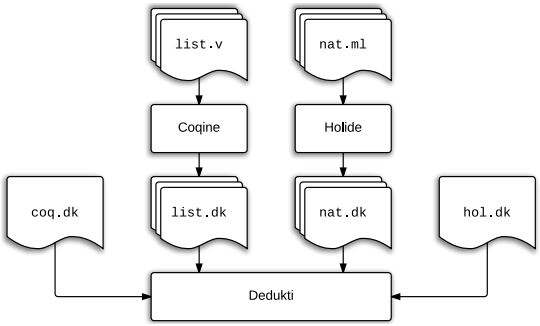
In this paper, we take one step further and show that we can combine the proofs coming from different systems in this same framework. A theorem can therefore be split into smaller blocks to be proved separately using different systems, and large libraries formalized in one system can be reused for the benefit of developments made in another one.

C.Kaliszyk and A. Paskevich (Eds.): 4th Workshop
on Proof eXchange for Theorem Proving (PxTP'15).
EPTCS 186, 2015, pp. 89–96, doi:10.4204/EPTCS.186.9

This approach has several advantages. First, we can use Dedukti as an independent proof checker. The $\lambda\Pi$-calculus modulo rewriting is fairly simple, and the kernel implementation is relatively small [23, 24] compared to systems like Coq. The soundness and completeness of the translations have been studied and proved [2, 10, 12], giving us further confidence. Compared to direct one-to-one translations [18, 19, 21, 20], we avoid the quadratic blowup of the number of translations needed to translate n systems. In that scenario, if a new proof system enters the market, we would need to design n new translations. Moreover, some systems such as Coq have complex foundations that are difficult to translate to other formalisms. Another possibility would be to compose existing translations, provided that they are scalable and composable. This avenue has not been investigated. In our approach, we instead translate the different systems to one common framework. We do not propose translations back into other systems, as we can use Dedukti as a low-level assembly language, akin to machine language when we compile and link programs coming from different programming languages.

Contributions

We used Holide and Coqine to translate proofs of HOL [15] and Coq [27], respectively, to Dedukti. We examined the logical theories behind those two systems to determine how we can combine them in a single unified theory while addressing the problems mentioned above. Finally, we used the resulting theory to certify the correctness of a sorting algorithm involving Coq lists of HOL natural numbers. Our code is available online at http://dedukti-interop.gforge.inria.fr/.



2 Tools used

Dedukti

Dedukti[1] is a functional language with dependent types based on the $\lambda\Pi$-calculus modulo rewriting [23, 24]. The type-checker/interpreter for Dedukti is called `dkcheck`. It accepts files written in the Dedukti format (`.dk`) containing declarations, definitions, and rewrite rules, and checks whether they are well-typed.

Following the LF tradition, Dedukti acts as a logical framework to define logics and express proofs in those logics. The approach consists in representing propositions as types and proofs as terms inhabiting those types, as in the Curry-Howard correspondence. Assuming the representation is correct, a proof is valid if and only if its corresponding proof term is well-typed. That way we can use Dedukti as an independent proof checker.

[1] Available at: http://dedukti.gforge.inria.fr/

Holide

Holide[2] translates HOL proofs to the Dedukti language. It accepts proofs in the OpenTheory format (`.art`) [17], and generates files in the Dedukti format (`.dk`). These files can then be verified by Dedukti to check that the proofs are indeed valid. The translation is described in detail in [3].

The generated files depend on a handwritten file called `hol.dk`. This file describes the theory of HOL, that is the types, the terms, and the derivation rules of HOL. The types of HOL are those of the simply-typed λ-calculus. We represent them as terms of type type (not to be confused with Type, the "type of types" of Dedukti). We represent the propositions as terms of type bool.

$$\begin{array}{lcl} \mathsf{type} & : & \mathsf{Type}. \\ \mathsf{bool} & : & \mathsf{type}. \\ \mathsf{arrow} & : & \mathsf{type} \to \mathsf{type} \to \mathsf{type}. \end{array} \qquad \begin{array}{lcl} \mathsf{term} & : & \mathsf{type} \to \mathsf{Type}. \\ \mathsf{proof} & : & \mathsf{term\ bool} \to \mathsf{type}. \\ \ldots & & \end{array}$$

Coqine

Coqine[3] translates Coq proofs to the Dedukti language. It takes the form of a Coq plugin that can be called to export loaded libraries (`.vo`) to generate files in the Dedukti format (`.dk`). These files can then be verified by Dedukti to check that the proofs are indeed valid.

A previous version of the translation is described in [6]. However, that translation is outdated, as it does not support the universe hierarchy and universe subtyping of Coq. A *universe* is just another name for a "'type of types". To avoid paradoxes, they are stratified into an infinite hierarchy [4], but that hierarchy is ignored by the first implementation of Coqine. The translation has since been updated to support both features following the ideas in [1], although some other features such as the module system are still missing.

The generated files depend on a handwritten file describing the theory of the calculus of inductive constructions (CIC) called `coq.dk`. There is a type prop that represents the universe of propositions and a type type i for every natural number i that represents the ith universe of types. We will write type_i and term_i for, respectively, type i and term i.

$$\begin{array}{lcl} \mathsf{type} & : & \mathsf{nat} \to \mathsf{Type}. \\ \mathsf{prop} & : & \mathsf{Type}. \\ \ldots & & \end{array} \qquad \begin{array}{lcl} \mathsf{term} & : & \Pi i : \mathsf{nat}.\ \mathsf{type}\ i \to \mathsf{Type}. \\ \mathsf{proof} & : & \mathsf{prop} \to \mathsf{Type}. \end{array}$$

3 Mixing HOL and Coq

HOL and Coq use very different logical theories. The first is based on Church's simple type theory, is implemented using the LCF approach, and its proofs are built by combining sequents in a bottom-up fashion. The second is based on the calculus of inductive constructions and checks proofs represented as λ-terms in a top-down fashion. Translating these two systems to Dedukti was a first step to bringing them closer together, but there are still important differences that set them apart. In this section, we examine these differences and show how we were able to bridge these gaps.

[2] Available at: https://www.rocq.inria.fr/deducteam/Holide/
[3] Available at: http://www.ensiie.fr/~guillaume.burel/blackandwhite_coqInE.html.en

Type inhabitation

The notion of types is different between HOL and Coq. In HOL, types are those of the simply-typed λ-calculus where every type is inhabited. In contrast, Coq allows the definition of empty types, which in fact play an important role as they are used to represent falsehood. A naïve reunion of the two theories would therefore be inconsistent: the formula $\exists x : \alpha, \top$, where α is a free type variable, is provable in HOL but its negation $\neg\forall\alpha : \mathsf{Type}, \exists x : \alpha, \top$ is provable in Coq.

Instead, we match the notion of HOL types with that of Coq's *inhabited* types, as done by Keller and Werner [19]. We define inhabited types in the Coq module `holtypes`:

```
Inductive type : Type := inhabited : forall (A : Type), A -> type.
```

It is then easy to prove in Coq that given inhabited types A and B, the arrow type $A \to B$ is also inhabited:

```
Definition carrier (A : type) : Type :=
  match A with inhabited B b => B end.
Definition witness (A : type) : carrier A :=
  match A with inhabited B b => b end.
Definition arrow (A : type) (B : type) : type :=
  inhabited (carrier A -> carrier B) (fun _ => witness B).
```

This is all that we need to interpret hol.type, hol.term, and hol.arrow using rewrite rules:

$$\begin{array}{lcl} \mathsf{hol.type} & \rightsquigarrow & \mathsf{coq.term}_1\ \mathsf{holtypes.type}. \\ \mathsf{hol.arrow}\ a\ b & \rightsquigarrow & \mathsf{holtypes.arrow}\ a\ b. \\ \mathsf{hol.term}\ a & \rightsquigarrow & \mathsf{coq.term}_1\ (\mathsf{holtypes.carrier}\ a). \end{array}$$

Booleans and propositions

In Coq, there is a clear distinction between booleans and propositions. Booleans are defined as an inductive type bool with two constructors true and false. The type bool lives in the universe Set (which is another name for the universe Type_0). In contrast, following the Curry-Howard correspondence, propositions are represented as types with proofs as their inhabitants. These types live in the universe Prop. Both Set and Prop live in the universe Type_1. As a consequence, Prop is not on the same level as other types such as bool or nat (the type of natural numbers), a notorious feature of the calculus of constructions. Moreover, since Coq is an intuitionistic system, there is no bijection between booleans and propositions. The excluded middle does not hold, though it can be assumed as an axiom.

In HOL, there is no distinction between booleans and propositions and they are both represented as a single type bool. Because the system is classical, it can be proved that there are only two inhabitants $\top$ and $\bot$, hence the name. Moreover, the type bool is just another simple type and lives on the same level as other types such as nat.

To combine the two theories, one must therefore reconcile the two pictures in Figure 1, which show how the types of HOL and Coq are organized.[4] One solution is to interpret the types of HOL as types in Set. To do this, we must rely on a reflection mechanism that interprets booleans as propositions, so that we can retrieve the theorems of HOL and interpret them as theorems in Coq. In our case, it consists of a function istrue of type hol.bool $\to$ coq.prop, which we use to define hol.proof:

$$\mathsf{hol.proof}\ b \rightsquigarrow \mathsf{coq.proof}\ (\mathsf{istrue}\ b).$$

[4] Since bool is the type of propositions, and propositions are the types of proofs in the Curry-Howard correspondence, bool can be viewed as a universe [4, 13].

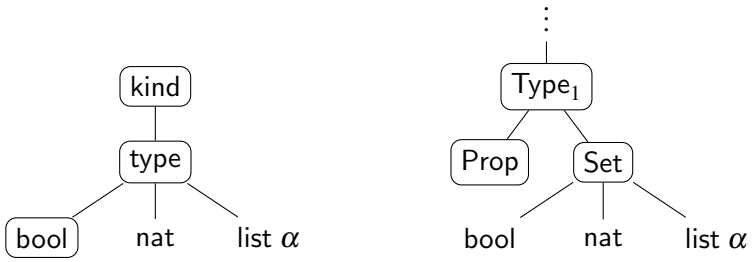


Figure 1: Booleans and propositions in HOL and Coq. Boxes represent universes.

Another solution is to translate hol.bool as coq.prop. To do this, we must therefore translate the types of HOL as types in Type_1 instead of Type_0. In particular, if we want to identify hol.nat and coq.nat, we must have coq.nat in Type_1. Fortunately, we have this for free with cumulativity since any element of Type_0 is also an element of Type_1.

We choose the first approach as it is more flexible and places less restrictions (e.g. regarding Prop elimination in Coq) on what we can do with booleans. In particular, it allows us to build lists by case analysis on booleans, which is needed in our case study.

4 Case study: sorting Coq lists of HOL numbers

We proved in Coq the correctness of the insertion sort algorithm on polymorphic lists and we instantiated it with the canonical order of natural numbers defined in HOL. More precisely, on the Coq side, we defined polymorphic lists, the insertion sort function, the sorted predicate, and the permutation relation. We then proved the following two theorems:

```
Theorem sorted_insertion_sort: forall l, sorted (insertion_sort l).
Theorem perm_insertion_sort: forall l, permutation l (insertion_sort l).
```

with respect to a given (partial) order:

```
Variable A : Set.
Variable compare : A -> A -> bool.
Variable leq : A -> A -> Prop.
Hypothesis leq_trans : forall a b c, leq a b -> leq b c -> leq a c.
Hypothesis leq_total : forall a b, if compare a b then leq a b else leq b a.
```

The order comes in two flavors: a relation leq used for proofs, and a decidable version compare which we can destruct for building lists. The totality assumptions relates leq and compare and can be seen as a specification of compare.

On the HOL side, we used booleans, natural numbers and the order relation on natural number as defined in the OpenTheory packages `bool.art` and `natural.art`. By composing the results, we obtain two Dedukti theorems:

$$\Pi l : \mathsf{coq.term}_1\ (\mathsf{coq\_list\ hol\_nat}).\ \mathsf{proof}\ (\mathsf{sorted}\ (\mathsf{insertion\_sort\ compare}\ l)).$$
$$\Pi l : \mathsf{coq.term}_1\ (\mathsf{coq\_list\ hol\_nat}).\ \mathsf{proof}\ (\mathsf{permutation}\ l\ (\mathsf{insertion\_sort\ compare}\ l)).$$

The composition takes place in a Dedukti file named `interop.dk`. This file takes care of matching the interfaces of the proofs coming from Coq with the proofs coming from HOL. Most of the work went into proving that HOL's comparison is indeed a total order in Coq:

$$\Pi m\ n : \mathsf{holtypes.carrier\ hol\_nat}.\ \mathsf{if}\ (\mathsf{compare}\ m\ n)\ \mathsf{then}\ m \leq n\ \mathsf{else}\ n \leq m.$$

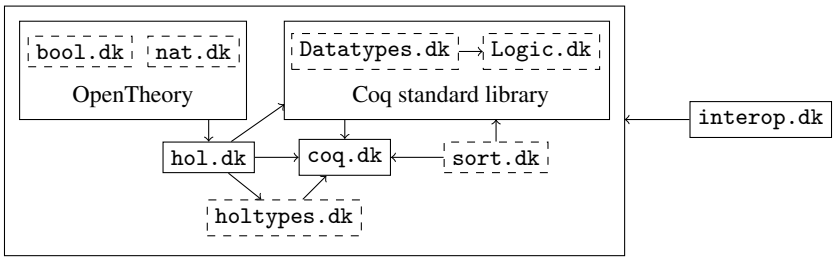


Figure 2: Components of the implementation. Solid frames represent source files. Dashed frames represent automatically generated files. Arrows represent dependencies.

We prove it using the following theorems from OpenTheory:

$$\forall m\ n : \mathsf{hol\_nat}.\ m < n \Rightarrow m \leq n$$
$$\forall m\ n : \mathsf{hol\_nat}.\ m \not\leq n \Leftrightarrow n < m$$

and some additional lemmas on if…then…else. Because of the verbosity of Dedukti and small style differences between HOL and Coq, this proof is long (several hundreds of lines) for such a simple fact. However, most of it is first-order reasoning and we believe that it could be automatically proved by the theorem prover Zenon [7] which can output proofs in the Dedukti format [9, 11].

We chose this example because the interaction between Coq and HOL types is very limited thanks to polymorphism: there is no need to reason about HOL natural numbers on the Coq side and no need to reason about lists on the HOL side so the only interaction takes place at the level of booleans which we wanted to study. We think it would have been harder for example to translate and link theorems about natural numbers in HOL and theorems about natural numbers in Coq. Our implementation is illustrated in Figure 2. All components were successfully verified by Dedukti.

5 Conclusion

We successfully translated a small Coq development to Dedukti and instanciated it with the HOL definition of natural numbers. The results have been validated by Dedukti. Mixing the underlying theories of Coq and HOL raised interesting questions but did not require a lot of human work: the file `hol.dk` is very close to the version included with Holide and the file `holtypes.v` is very small. In retrospect, the result looks a lot like an embedding of HOL in Coq but performed in Dedukti. This is not surprising, as the theory of HOL is fairly simple compared to Coq and is in fact a subset of the logic of Coq [4, 13, 19].

The interoperability layer `interop.dk` which is specific to our case study required a lot of work which should be automated before using this approach on larger scale; our next step on this front will be to integrate Zenon to solve the proof obligations when they happen to be in the first-order fragment. Interoperability raises more issues than mere proof rechecking and our translators to Dedukti need to be improved. The translations produce code intended for machines that is not very usable by humans. The linking of theories together should therefore either be more automated or benefit from a more readable output. We expect more complex examples of interoperability to require some form of parametrization in the translators: when the developer wants the translator to map a given symbol to a specific Dedukti definition, he should be able to alter the behaviour of the translator by annotations in some source file, as done by Keller and Werner [19] and by Hurd [17].

Another limitation of this example of interoperability is the lack of executability. Even though we have constructed a sorting "algorithm" on lists of HOL natural numbers and we have proved it correct, there is no way to actually execute this algorithm. Indeed, there is no notion of computation in HOL, so when the sorting algorithm asks compare for a comparison between two numbers, it will not return something which will unblock the computation. Therefore, insertion_sort $[4,1,3,2]$ is not *computationally* equal to $[1,2,3,4]$. However, the result is still *provably* equal to what is expected: we can show that insertion_sort $[4,1,3,2]$ is equal to $[1,2,3,4]$. A constructive and computational presentation of HOL will be necessary before we can obtain truly executable code. The pure type system presentation of HOL [4, 13] is a reasonable candidate for that but the proofs of OpenTheory will need to be adapted. Holide seems like a good starting point for such a transformation and is the subject of current ongoing work.

References

[1] Ali Assaf (2014): *A calculus of constructions with explicit subtyping*. Available at https://hal.inria.fr/hal-01097401. Accepted in Postproceedings of Types 2014.

[2] Ali Assaf (2015): *Conservativity of embeddings in the lambda-Pi calculus modulo rewriting*. In Thorsten Altenkirch, editor: *International Conference on Typed Lambda Calculi and Applications (TLCA)*, *LIPIcs* 38, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 31–44, doi:10.4230/LIPIcs.TLCA.2015.31.

[3] Ali Assaf & Guillaume Burel (2015): *Translating HOL to Dedukti*. Available at https://hal.inria.fr/hal-01097412. Accepted in PxTP 2015.

[4] Henk Barendregt (1992): *Lambda calculi with types*. In Samson Abramsky, Dov M. Gabbay & Thomas S. E. Maibaum, editors: *Handbook of Logic in Computer Science*, 2, Oxford University Press, pp. 117–309.

[5] M. Boespflug, Q. Carbonneaux & O. Hermant (2012): *The lambda-Pi-calculus modulo as a universal proof language*. In: *Proof Exchange for Theorem Proving - Second International Workshop, PxTP 2012*, pp. 28–43.

[6] Mathieu Boespflug & Guillaume Burel (2012): *CoqInE: Translating the calculus of inductive constructions into the λΠ-calculus modulo*. In: *Proof Exchange for Theorem Proving - Second International Workshop, PxTP 2012*, p. 44.

[7] Richard Bonichon, David Delahaye & Damien Doligez (2007): *Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs*. In: *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, *LNCS/LNAI* 4790, Springer, pp. 151–165, doi:10.1007/978-3-540-75560-9_13.

[8] Guillaume Burel (2013): *A Shallow Embedding of Resolution and Superposition Proofs into the λΠ-Calculus Modulo*. In Jasmin Christian Blanchette & Josef Urban, editors: *Proof Exchange for Theorem Proving - Third International Workshop, PxTP 2013*, *EPiC* 14, EasyChair, pp. 43–57.

[9] Raphaël Cauderlier & Pierre Halmagrand (2015): *Checking Zenon Modulo proofs in Dedukti*. Accepted in PxTP 2015.

[10] Denis Cousineau & Gilles Dowek (2007): *Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo*. In Simona Ronchi Della Rocca, editor: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, *LNCS* 4583, Springer, pp. 102–117, doi:10.1007/978-3-540-73228-0_9.

[11] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand & Olivier Hermant (2013): *Zenon Modulo: When Achilles Outruns the Tortoise Using Deduction Modulo*. In Ken McMillan, Aart Middeldorp & Andrei Voronkov, editors: *LPAR*, *LNCS* 8312, Springer Berlin Heidelberg, pp. 274–290, doi:10.1007/978-3-642-45221-5_20.

[12] Gilles Dowek (2014): *Models and termination of proof-reduction in the lambda-Pi-calculus modulo theory*. Technical report, Inria, Paris. Available at https://who.rocq.inria.fr/Gilles.Dowek/Publi/superpi.pdf.

[13] Herman Geuvers (1993): *Logics and type systems*. PhD thesis, University of Nijmegen.

[14] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A framework for defining logics*. *Journal of the ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[15] John Harrison (2009): *HOL Light: An Overview*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics*, *LNCS* 5674, Springer Berlin Heidelberg, pp. 60–66, doi:10.1007/978-3-642-03359-9_4.

[16] Fulya Horozal & Florian Rabe (2011): *Representing model theory in a type-theoretical logical framework*. *Theoretical Computer Science* 412, pp. 4919–4945, doi:10.1016/j.tcs.2011.03.022.

[17] Joe Hurd (2011): *The OpenTheory Standard Theory Library*. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *NFM*, *LNCS* 6617, Springer, pp. 177–191, doi:10.1007/978-3-642-20398-5_14.

[18] Cezary Kaliszyk & Alexander Krauss (2013): *Scalable LCF-style proof translation*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *Interactive Theorem Proving*, *LNCS* 7998, Springer Berlin Heidelberg, pp. 51–66, doi:10.1007/978-3-642-39634-2_7.

[19] Chantal Keller & Benjamin Werner (2010): *Importing HOL Light into Coq*. In Matt Kaufmann & Lawrence C. Paulson, editors: *ITP*, *LNCS* 6172, Springer Berlin Heidelberg, pp. 307–322, doi:10.1007/978-3-642-14052-5_22.

[20] Pavel Naumov, Mark-Oliver Stehr & José Meseguer (2001): *The HOL/NuPRL proof translator*. In Richard J. Boulton & Paul B. Jackson, editors: *Theorem Proving in Higher Order Logics*, *LNCS* 2152, Springer Berlin Heidelberg, pp. 329–345, doi:10.1007/3-540-44755-5_23.

[21] Steven Obua & Sebastian Skalberg (2006): *Importing HOL into Isabelle/HOL*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning*, *LNCS* 4130, Springer Berlin Heidelberg, pp. 298–302, doi:10.1007/11814771_27.

[22] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In: *Automated Deduction — CADE-16*, *LNCS* 1632, Springer Berlin Heidelberg, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[23] Ronan Saillard (2013): *Dedukti: a universal proof checker*. In: *Foundation of Mathematics for Computer-Aided Formalization Workshop*, Padova. Available at https://hal.inria.fr/hal-00833992.

[24] Ronan Saillard (2013): *Towards explicit rewrite rules in the $\lambda\Pi$-calculus modulo*. In: *IWIL - 10th International Workshop on the Implementation of Logics*. Available at https://hal.inria.fr/hal-00921340.

[25] Carsten Schürmann & Mark-Oliver Stehr (2006): *An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf*. In Miki Hermann & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning*, *LNCS* 4246, Springer Berlin Heidelberg, pp. 150–166, doi:10.1007/11916277_11.

[26] Geoff Sutcliffe, Stephan Schulz, Koen Claessen & Allen Van Gelder (2006): *Using the TPTP Language for Writing Derivations and Finite Interpretations*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning*, *LNCS* 4130, Springer Berlin Heidelberg, pp. 67–81, doi:10.1007/11814771_7.

[27] The Coq development team (2012): *The Coq Reference Manual, version 8.4*. Available at http://coq.inria.fr/doc.