



PxTP 2021 - Seventh Workshop on Proof eXchange for Theorem Proving

Chantal Keller, Mathias Fleury

► To cite this version:

Chantal Keller, Mathias Fleury. PxTP 2021 - Seventh Workshop on Proof eXchange for Theorem Proving. Electronic Proceedings in Theoretical Computer Science, 336, 61 p., 2021, 10.4204/eptcs.336 . hal-03443742

HAL Id: hal-03443742

<https://hal.science/hal-03443742>

Submitted on 26 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

EPTCS 336

Proceedings of the
**Seventh Workshop on
Proof eXchange for Theorem Proving**

Pittsburg, USA, 11th July 2021

Edited by: Chantal Keller and Mathias Fleury

Published: 7th July 2021
DOI: 10.4204/EPTCS.336
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents	i
Preface	ii
Proof Generation in CDSAT..... <i>Maria Paola Bonacina</i>	1
A Framework for Proof-carrying Logical Transformations..... <i>Quentin Garchery</i>	5
General Automation in Coq through Modular Transformations..... <i>Valentin Blot, Louise Dubois de Prisque, Chantal Keller and Pierre Vial</i>	24
Integrating an Automated Prover for Projective Geometry as a New Tactic in the Coq Proof Assistant..... <i>Nicolas Magaud</i>	40
Certifying CNF Encodings of Pseudo-Boolean Constraints (abstract)..... <i>Stephan Gocht, Jakob Nordström and Ruben Martins</i>	48
Alethe: Towards a Generic SMT Proof Format (extended abstract)..... <i>Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa and Pascal Fontaine</i>	49

Preface

This volume of EPTCS contains the proceedings of the Seventh Workshop on Proof Exchange for Theorem Proving (PxTP 2021), held on 11 July 2021 as part of the CADE-28 conference, originally planned in Pittsburg, USA, but finally hold in cyberspace.

The PxTP workshop series brings together researchers working on various aspects of communication, integration, and cooperation between reasoning systems and formalisms, with a special focus on proofs.

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 between such tools in larger systems has demonstrated the potential to reduce the amount of manual intervention.

Cooperation between reasoning systems relies on availability of theoretical formalisms and practical tools to exchange problems, proofs, and models. The PxTP workshop series strives to encourage such cooperation by inviting contributions on all aspects of cooperation between reasoning tools, whether automatic or interactive, including the following topics:

- applications that integrate reasoning tools (ideally with certification of the result);
- interoperability of reasoning systems;
- translations between logics, proof systems, models;
- distribution of proof obligations among heterogeneous reasoning tools;
- algorithms and tools for checking and importing (replaying, reconstructing) proofs;
- proposed formats for expressing problems and solutions for different classes of logic solvers (SAT, SMT, QBF, first-order logic, higher-order logic, typed logic, rewriting, etc.);
- metalanguages, logical frameworks, communication methods, standards, protocols, and APIs related to problems, proofs, and models;
- comparison, refactoring, transformation, migration, compression and optimization of proofs;
- data structures and algorithms for improved proof production in solvers (e.g. efficient proof representations);
- (universal) libraries, corpora and benchmarks of proofs and theories;
- alignment of diverse logics, concepts and theories across systems and libraries;
- engineering aspects of proofs (e.g. granularity, flexiformality, persistence over time);
- proof certificates;
- proof checking;
- mining of (mathematical) information from proofs (e.g. quantifier instantiations, unsat cores, interpolants, . . .);
- reverse engineering and understanding of formal proofs;
- universality of proofs (i.e. interoperability of proofs between different proof calculi);
- origins and kinds of proofs (e.g. (in)formal, automatically generated, interactive, . . .);
- Hilbert’s 24th Problem (i.e. what makes a proof better than another?);

- social aspects (e.g. community-wide initiatives related to proofs, cooperation between communities, the future of (formal) proofs);
- applications relying on importing proofs from automatic theorem provers, such as certified static analysis, proof-carrying code, or certified compilation;
- application-oriented proof theory;
- practical experiences, case studies, feasibility studies;

Previous editions of the workshop took place in Wrocław (2011), Manchester (2012), Lake Placid (2013), Berlin (2015), Brasília (2017), and Natal (2019).

This edition of the workshop received submissions of three regular papers and three extended abstracts. All submissions were evaluated by at least three anonymous reviewers. Two full papers and three extended abstracts were accepted in the post-proceedings.

The program committee had the following members: Chantal Keller (co-chair, LRI, Université Paris-Saclay), Mathias Fleury (co-chair, JKU Linz), Haniel Barbosa (Universidade Federal de Minas Gerais), Denis Cousineau (Mitsubishi Electric R&D Centre Europe), Stefania Dumbrava (ENSIIE, Télécom Sud-Paris), Katalin Fazekas (TU Wien), Predrag Janičić (Univerzitet u Beogradu), Jens Otten (University of Oslo), Aina Niemetz (Stanford University), Giselle Reis (Carnegie Mellon University), Geoff Sutcliffe (University of Miami), François Thiré (Nomadic Labs), Sophie Turret (Inria), and Josef Urban (Czech Technical University in Prague).

We would like to thank all authors for their submissions and all members of the program committee for the time and energy they spent to diligently ensure that accepted papers were of high quality. We also thank EasyChair for making it easy to chair the reviewing process. Furthermore, we are thankful to the CADE-28 organizers who organized the conference in these challenging times.

We had the honor to welcome two invited speakers: Maria Paola Bonacina from the Università degli Studi di Verona gave a talk entitled *Proof Generation in CDSAT* and Giles Reger from the University of Manchester, presented a talk entitled *Reasoning in many logics with Vampire: Everything's CNF in the end*. The presentations from the invited speakers and the presentations from the accepted papers raised numerous, interesting, and fruitful discussions. We thank all the participants of the workshop.

The organization of this edition of PxTP stood on the shoulders of previous editions, and we are grateful to the chairs of previous editions for all the resources and infrastructure that they made available to us.

July 11, 2021

Chantal Keller and Mathias Fleury

Proof Generation in CDSAT

Maria Paola Bonacina

Dipartimento di Informatica, Università degli Studi di Verona, Verona, Italy

`mariapaola.bonacina@univr.it`

Proofs of unsatisfiability of a negated conjecture, or, equivalently, proofs of validity of the original conjecture, are an essential output of automated reasoning methods. The transformation, exchange, and standardization of proofs is a key factor for the interoperability of different automated reasoning systems. In theorem proving *proof reconstruction* is the task of extracting a proof from the final state of a derivation after generating the empty clause. While for several theorem proving methods and theorem provers it is a standard task, it is never trivial. For example, in parallel theorem proving with distributed search (see [6] for a recent survey), multiple parallel processes perform inferences and search for a proof. A parallel theorem proving method has *distributed proof reconstruction*, if the process that generates the empty clause can reconstruct the proof from the final state of its database, even if all processes contributed to the proof [4].

In propositional satisfiability (SAT) solving, the *conflict-driven clause learning* (CDCL) procedure generates proofs by *resolution*, because it uses resolution to explain conflicts [28, 26]. SAT solvers apply pre-processing steps and simplification techniques that also need to be accounted for in proofs. Furthermore, proofs generated by SAT solvers are so huge that their definition, generation, and manipulation, involving various proof formats, is an important research topic (e.g., [14]).

Satisfiability modulo theories (SMT) solving represents a middle ground between first-order theorem proving and SAT solving. Initially, *model generation* was emphasized over proof generation in SMT, because the focus was on fragments of first-order theories where satisfiability is decidable, in contrast with first-order logic where satisfiability is not even semidecidable. Over time, SMT solvers have been applied more and more to unsatisfiable inputs, including inputs with quantifiers that may fall outside decidable fragments. SMT solvers have become more similar to theorem provers, and proof generation is crucial also in SMT. Since most SMT solvers are built on top of the CDCL procedure, their proofs are proofs by resolution (with the same caveat as above) with proofs of theory lemmas plugged in as leaves or *black-box sub-proofs* [17, 2, 12, 23, 1].

CDSAT (*Conflict-Driven SATisfiability*) is a paradigm for SMT that innovates SMT solving in several ways [8, 9, 10, 11]. To begin with, CDSAT solves *SATisfiability problems Modulo theories and Assignments* (SMA), which means that the input problem may contain assignments to first-order terms (e.g., $x \leftarrow 3$). The solver has to determine whether there exists a model that satisfies the input formula and also *endorses* the input first-order assignments. A model *endorses* an assignment if it interprets identically left hand side and right hand side of the assignment. For uniformity, CDSAT views also formulae as assignments to Boolean terms (e.g., $(\neg A \vee B) \leftarrow \text{true}$), and seeks a model that endorses all input assignments. There is a subtle technical difference between, say, $x \leftarrow 3$ and $(x \simeq 3) \leftarrow \text{true}$, since in the latter 3 is a constant symbol of the input language, whereas in the former 3 is a *value*, whose denotation requires a *theory extension*. The generalization of SMT to SMA is relevant to approaching *optimization* problems by solving iteratively SMA problems, where input first-order assignments are used to exclude sub-optimal solutions and induce a convergence towards an optimal one [16].

As the name says, CDSAT is a *conflict-driven* method. In general, a procedure is *conflict-driven* if it proposes a candidate model represented by a series of assignments, and performs non-trivial inferences

only to explain a conflict between the current candidate model and the formulæ to be satisfied. Since in CDSAT also formulæ are assignments, the separation between candidate model and formulæ disappears. The state of the computation is simply a sequence of assignments Γ , called a *trail*, which also contains the input assignments. A *conflict* is a subset of Γ that is unsatisfiable.

CDSAT is designed since the start for reasoning in a *union of theories*, with propositional logic as one of the theories. CDSAT lifts the conflict-driven style of CDCL from propositional logic to *conflict-driven reasoning in a union of theories*; and it reduces to CDCL if propositional logic is the sole theory. Prior to CDSAT, MCSAT (*Model-Constructing SATisfiability*) [15, 20, 27, 19, 3, 18] showed how to integrate CDCL with a conflict-driven theory satisfiability procedure (e.g., [22, 21, 13] and see [5] for a survey with more references). CDSAT generalizes MCSAT to generic unions of *disjoint* theories, meaning that their signatures do not share symbols other than equality on shared sorts. CDSAT resembles MCSAT, if there are only propositional logic and another theory with a conflict-driven satisfiability procedure.

For an input problem to be satisfiable in a union of theories, the theories need to agree on which shared terms are equal and on the cardinalities of shared sorts. Beginning with the pioneering work of Nelson and Oppen [25, 24], most approaches to reasoning in a union of theories are defined as *combination schemes* that combine theory satisfiability procedures (see [7] for a survey with more references). These schemes *separate* the original problem into sub-problems, one per theory in the union. The completeness of the combination scheme rests on a *combination lemma* that states which conditions the theories need to satisfy in order to agree on the cardinalities of shared sorts. The satisfiability of the original input in the union of theories is reduced to the satisfiability of every sub-problem in the respective theory, where every sub-problem is conjoined with an *arrangement*. An *arrangement* is a conjunction of equalities and disequalities between shared variables, or shared constants, depending on whether free variables or constants are used to represent shared terms. In a non-deterministic description the arrangement can be guessed. In practice, it is computed by the theory satisfiability procedures. The computation of the arrangement is the only activity where the theory satisfiability procedures cooperate, typically by exchanging equalities between shared variables.

In contrast with this traditional setting, CDSAT is defined as a *transition system* that orchestrates theory-specific *inference systems*, called *theory modules*. An inference system is a set of inference rules, and a theory module is an abstraction of a satisfiability procedure. Every module has its view of the trail, called *theory view*, which contains whatever the module can understand. A theory module can *expand* the trail with an assignment that is a *decision*, encapsulated in the `decide` transition rule of CDSAT, or the result of a *theory inference*, encapsulated in the `deduce` transition rule of CDSAT. Theory inferences are used for propagations, and conflict detection and explanation in the respective theory. The latter applies until the theory conflict surfaces on the trail as a Boolean conflict (e.g., $L \leftarrow \text{true}$ and $\neg L \leftarrow \text{true}$, or, equivalently, $L \leftarrow \text{true}$ and $L \leftarrow \text{false}$). Then the conflict-solving transition rules of CDSAT come into play. Since the Boolean conflict may descend from first-order assignments, the conflict-solving transition rules of CDSAT are designed to handle both Boolean and first-order assignments. It does not matter whether a theory satisfiability procedure is conflict-driven, because CDSAT is conflict-driven for all theories. At the very least, a theory satisfiability procedure can be abstracted into a *black-box theory module*, with an inference rule that detects unsatisfiability by invoking the procedure.

The completeness of CDSAT rests mainly on properties of the theory modules. Every theory module is required to be *complete*, meaning that it can expand its view of the trail if it is not satisfied by a model of its theory. One of the theories in the union needs to be the *leading theory*. A leading theory is aware of all the sorts in the union of theories, and its theory module is aware of all the constraints that the theories may have on the cardinalities of shared sorts. While for the leading theory module it suffices to be complete, any other module needs to be *leading-theory-complete*, meaning that it can expand its view

of the trail if it is not satisfied by a model of its theory that concurs with a model of the leading theory on cardinalities of shared sorts and equality of shared terms.

This description shows that while the traditional combination schemes combine decision procedures as *black-boxes*, CDSAT provides a tighter form of integration at the inference level. This has consequences on *proof generation*. Since the conflict-driven reasoning happens directly in the union of the theories and not only in propositional logic, resolution does not have a dominant role. CDSAT proofs can be rendered as resolution proofs, but this is not a necessary choice. Since the theory satisfiability procedures are not combined as black-boxes, theory sub-proofs are not necessarily black-boxes either. Since CDSAT solves SMA problems, also first-order assignments may appear in proofs. The theory inferences may introduce *new* (i.e., non-input) terms, in order to explain conflicts. Thus, such new terms may appear in proofs.

The powerful abstractions that characterize CDSAT leads to proof generation approaches also based on abstraction. The CDSAT transition system can be made *proof-carrying*, by equipping the transition rules with the capability to generate *abstract proof terms*. During proof reconstruction these proof terms can be translated into different proof formats, including resolution proofs. The resulting proofs can be dispatched to proof checkers or proof assistants, or otherwise manipulated and integrated. Alternatively, CDSAT can adopt the LCF style for proofs, which avoids building proof objects in memory altogether. In LCF style, the prover or solver (e.g., a CDSAT based solver) is built on top of a *trusted kernel* of primitive operations. When the reasoner detects unsatisfiability, the refutation is correct by construction, because otherwise a type error would arise.

Acknowledgements The author thanks Stéphane Graham-Lengrand for their discussions.

References

- [1] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury & Pascal Fontaine (2020): *Scalable Fine-Grained Proofs for Formula Processing*. *J. of Autom. Reason.* 64(3), pp. 485–550, doi:10.1007/s10817-018-09502-y.
- [2] Nikolaj Bjørner & Leonardo de Moura (2008): *Proofs and refutations, and Z3*. In: *IWIL-7, CEUR* 418, pp. 123–132.
- [3] François Bobot, Stéphane Graham-Lengrand, Bruno Marre & Guillaume Bury (2018): *Centralizing equality reasoning in MCSAT*. In: *SMT-16*.
- [4] Maria Paola Bonacina (1996): *On the reconstruction of proofs in distributed theorem proving: a modified Clause-Diffusion method*. *J. of Symb. Comput.* 21(4–6), pp. 507–522, doi:10.1006/jscs.1996.0028.
- [5] Maria Paola Bonacina (2018): *On conflict-driven reasoning*. In: *AFM-6, Kalpa Publications* 5, EasyChair, pp. 31–49, doi:10.29007/spwm.
- [6] Maria Paola Bonacina (2018): *Parallel theorem proving*. In: *Handbook of Parallel Constraint Reasoning*, chapter 6, Springer, pp. 179–235, doi:10.1007/978-3-319-63516-3_6.
- [7] Maria Paola Bonacina, Pascal Fontaine, Christophe Ringeissen & Cesare Tinelli (2019): *Theory combination: beyond equality sharing*. In: *Description Logic, Theory Combination, and All That: Essays Dedicated to Franz Baader, LNAI* 11560, Springer, pp. 57–89, doi:10.1007/978-3-030-22102-7_3.
- [8] Maria Paola Bonacina, Stéphane Graham-Lengrand & Natarajan Shankar (2017): *Satisfiability modulo theories and assignments*. In: *CADE-26, LNAI* 10395, Springer, pp. 42–59, doi:10.1007/978-3-319-63046-5_4.
- [9] Maria Paola Bonacina, Stéphane Graham-Lengrand & Natarajan Shankar (2018): *Proofs in conflict-driven theory combination*. In: *CPP-7, ACM*, pp. 186–200, doi:10.1145/3167096.

- [10] Maria Paola Bonacina, Stéphane Graham-Lengrand & Natarajan Shankar (2020): *Conflict-driven satisfiability for theory combination: transition system and completeness*. *J. of Autom. Reason.* 64(3), pp. 579–609, doi:10.1007/s10817-018-09510-y.
- [11] Maria Paola Bonacina, Stéphane Graham-Lengrand & Natarajan Shankar (2021): *Conflict-Driven Satisfiability for Theory Combination: Lemmas, Modules, and Proofs*. *J. of Autom. Reason.* Submitted, pp. 1–54. <http://profs.sci.univr.it/~bonacina/cdsat.html>.
- [12] Maria Paola Bonacina & Moa Johansson (2015): *Interpolation systems for ground proofs in automated deduction: a survey*. *J. of Autom. Reason.* 54(4), pp. 353–390, doi:10.1007/s10817-015-9325-5.
- [13] Franz Brauße, Konstantin Korovin, Margarita Korovina & Norbert Müller (2019): *A CDCL-style calculus for solving non-linear constraints*. In: *FroCoS-12, LNAI 11715*, Springer, pp. 131–148, doi:10.1007/978-3-030-29007-8_8.
- [14] Luís Cruz-Felipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann & Peter Schneider-Kamp (2017): *Efficient certified RAT verification*. In: *CADE-26, LNAI 10395*, Springer, pp. 220–236, doi:10.1007/978-3-319-63046-5_14.
- [15] Leonardo de Moura & Dejan Jovanović (2013): *A model-constructing satisfiability calculus*. In: *VMCAI-14, LNCS 7737*, Springer, pp. 1–12, doi:10.1007/978-3-642-35873-9_1.
- [16] Leonardo de Moura & Grant Olney Passmore (2013): *Exact global optimization on demand (Presentation only)*. In: *ADDCT-3*. Available at <https://userpages.uni-koblenz.de/~sofronie/addct-2013/>.
- [17] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto & Alwen Tiu (2006): *Expressiveness+automation+soundness: towards combining SMT solvers and interactive proof assistants*. In: *TACAS-12, LNCS 3920*, Springer, pp. 167–181, doi:10.1007/11691372_11.
- [18] Stéphane Graham-Lengrand, Dejan Jovanović & Bruno Dutertre (2020): *Solving bitvectors with MCSAT: explanations from bits and pieces*. In: *IJCAR-10, LNAI 12166*, Springer, pp. 103–121, doi:10.1007/978-3-030-51074-9_7.
- [19] Dejan Jovanović (2017): *Solving nonlinear integer arithmetic with MCSAT*. In: *VMCAI-18, LNCS 10145*, Springer, pp. 330–346, doi:10.1007/978-3-319-52234-0_18.
- [20] Dejan Jovanović, Clark Barrett & Leonardo de Moura (2013): *The design and implementation of the model-constructing satisfiability calculus*. In: *FMCAD-13*, ACM and IEEE.
- [21] Dejan Jovanović & Leonardo de Moura (2013): *Cutting to the chase: solving linear integer arithmetic*. *J. of Autom. Reason.* 51, pp. 79–108, doi:10.1007/s10817-013-9281-x.
- [22] Dejan Jovanović & Leonardo de Moura (2012): *Solving non-linear arithmetic*. In: *IJCAR-6, LNAI 7364*, Springer, pp. 339–354, doi:10.1007/978-3-642-31365-3_27.
- [23] Guy Katz, Clark W. Barrett, Cesare Tinelli, Andrew Reynolds & Liana Hadarean (2016): *Lazy proofs for DPLL(T)-based SMT solvers*. In: *FMCAD-16*, ACM and IEEE, pp. 93–100, doi:10.1109/FMCAD.2016.7886666.
- [24] Greg Nelson (1983): *Combining satisfiability procedures by equality sharing*. In: *Automatic Theorem Proving: After 25 Years*, AMS, pp. 201–211, doi:10.1090/conm/029/11.
- [25] Greg Nelson & Derek C. Oppen (1979): *Simplification by Cooperating Decision Procedures*. *ACM Trans. on Prog. Lang. and Syst.* 1(2), pp. 245–257, doi:10.1145/357073.357079.
- [26] Natarajan Shankar (2009): *Automated deduction for verification*. *ACM Comput. Surv.* 41(4), pp. 507–522, doi:10.1145/1592434.1592437.
- [27] Aleksandar Zeljić, Christoph M. Wintersteiger & Philipp Rümmer (2016): *Deciding bit-vector formulas with mcSAT*. In: *SAT-19, LNCS 9710*, Springer, pp. 249–266, doi:10.1007/978-3-319-40970-2_16.
- [28] Lintao Zhang & Sharad Malik (2003): *Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications*. In: *DATE 2003*, IEEE, pp. 10880–10885, doi:10.5555/789083.1022835.

A Framework for Proof-carrying Logical Transformations

Quentin Garchery

Université Paris-Saclay, CNRS, Inria, LMF, 91405, Orsay, France

In various provers and deductive verification tools, logical transformations are used extensively in order to reduce a proof task into a number of simpler tasks. Logical transformations are often part of the trusted base of such tools. In this paper, we develop a framework to improve confidence in their results. We follow a modular and skeptical approach: transformations are instrumented independently of each other and produce certificates that are checked by a third-party tool. Logical transformations are considered in a higher-order logic, with type polymorphism and built-in theories such as equality and integer arithmetic. We develop a language of proof certificates for them and use it to implement the full chain of certificate generation and certificate verification.

1 Introduction

General Context and Motivation. Verifying a program is meant to improve its soundness guarantees and relies on the trust towards the verification tool. Given how difficult it can be to verify relatively simple programs, most tools try to simplify this process and to make it as automatized as possible, which can drastically extend their trusted code base.

Consider deductive program verification, where the program to verify is annotated, and, in particular, given a specification. In this setting, the code is analyzed against its specification thus generating *proof tasks*, logical statements upon which depends the program correctness. To discharge a proof task, one can first apply a *logical transformation* which reduces it to a number of new proof tasks which are hopefully easier to discharge. Transformations are powerful tools that can, for example, be applied to translate a task into a prover’s logic before calling it.

The main objective of this article is to improve trust in those logical transformations. This work is decisive because logical transformations are general and can be used in many different settings. We apply our method to the deductive program verification tool Why3 [6], which makes extensive use of logical transformations. In fact, they are at the core of its interactive theorem proving feature and are necessary to be able to encode proof tasks into the logic of one of the dozens of third-party automatic theorem provers available inside Why3. The implementation of the transformations adds up to a total of more than 17000 lines of OCaml code. This code being in the trusted Why3 code base, it represents an interesting case study.

Example 1. Suppose that we have a proof task where we have to prove that $p(y * y)$ holds for any integer y of the form $y = 2 * x + 1$ and any integer predicate p that satisfies the hypothesis H stating that $\forall i : \text{int}. p(4 * i + 1)$. Finding how to instantiate the hypothesis H is difficult or even impossible for some automatic theorem provers so we cannot automatically discharge this task. The transformation *instantiate*, defined in Why3, simply instantiates an hypothesis and when called with arguments H and $x * x + x$ on the given task, produces the same task but with the added hypothesis that states that $p(4 * (x * x + x) + 1)$. Provers won’t have to instantiate the hypothesis H to discharge this new task. In fact, this task can now be discharged by theorem provers capable of handling arithmetic goals on condition of translating it into the logic of the prover in question. This translation is also being done

with the help of transformations. The entire process described in this example is to be trusted to ensure correctness of the initial program.

Contributions and Overview. In this article, we describe a practical framework to validate logical transformations. In order to define what it means for a transformation to be correct, the logical setting of proof tasks is detailed in Section 2. We follow a skeptical approach [4]: certificates, defined in Section 3, are generated every time a transformation is applied and are checked independently at a later time. Contrary to the autarkic approach, used for example for some automatic theorem provers [28], which would consist here in verifying directly the transformations, the skeptical approach has the benefit of not fixing the implementation of the transformations. Our work is based on certificates with holes, a notion that is, to our knowledge, new in the setting of the skeptical approach. This allows for modular development, where certificates can be built incrementally and transformations can be composed and defined independently. We extend our framework with some key interpreted theories in Section 4 and show how to do so for any other interpreted theory along the way. The checkers for our certificates can also be defined independently, as it is done in Section 5. In fact, we designed two checkers and one of them is based on Lambdapi/Dedukti [3], an off-the-shelf proof assistant. This has also led us to develop a translation procedure for proof tasks into the $\lambda\Pi$ -Calculus modulo rewriting. This approach, while applicable to logical transformations in general, has been applied to the program verification tool Why3 for a number of its transformations including transformations dealing specifically with higher-order logic. We conclude this article by evaluating this application to Why3 in Section 6. The source code for the whole work described in this article is available in the Why3 repository [21].

2 Logical Setting

We present the logical setting used throughout this article. The goal here is to define logical transformations and the proof tasks they are applied to.

2.1 Types and Terms

Proof tasks are formed from typed terms and those terms are meant to designate both the terms from the program and the formulas stating properties about them. We use the Hindley-Milner type system [29] except that our terms are explicitly quantified over types. Names are taken from an infinite set of available identifiers which is designated by *ident*.

Types are described by a *type signature* I , a set of pairs of the form ' $\iota : n$ ' composed of an *ident* called type symbol and an integer representing its arity. Sets are denoted by separating their elements with commas. Note that according to the following grammar, type symbols are always completely applied.

$type ::= \alpha$	type variable
$prop$	type of formulas
$type \rightsquigarrow type$	arrow type
$\iota(type, \dots, type)$	type symbol application

Terms have polymorphic types and new terms can be built by quantifying over terms of any type. Quantification over type variables is explicit and restricted to only be in the prenex form. Note that the application uses the Curry notation, i.e., a function term is applied to a single argument term at a time. The application is left-associative and the type arrow \rightsquigarrow is right-associative.

$term_{poly}$	$::=$	$term_{mono}$	
	$ $	$\Pi\alpha. term_{poly}$	type quantifier
$term_{mono}$	$::=$	x	variable
	$ $	\top	true formula
	$ $	\perp	false formula
	$ $	$\neg term_{mono}$	negation
	$ $	$term_{mono} op term_{mono}$	logical binary operator
	$ $	$term_{mono} term_{mono}$	application
	$ $	$\lambda x : type. term_{mono}$	anonymous function
	$ $	$\exists x : type. term_{mono}$	existential quantifier
	$ $	$\forall x : type. term_{mono}$	universal quantifier
op	$::=$	\wedge	conjunction
	$ $	\vee	disjunction
	$ $	\Rightarrow	implication
	$ $	\Leftrightarrow	equivalence

The (*term*) *substitution* of variable x by term u in term t is written $t[x \mapsto u]$ and $t[\alpha \mapsto \tau]$ is the (*type*) *substitution* of type variable α by type τ in term t . A *signature* Σ is a set of pairs of the form ' $x : \tau$ ' composed of a variable and its type; this type should be understood as quantified over all of its type variables.

Definition 2 (Typing). We write $I \mid \Sigma \Vdash t : \tau$ when the term t has no free type variables and has type τ in type signature I and signature Σ . We omit I when it is clear from the context.

The complete set of rules defining this predicate is given in Appendix A. Remark that, in the case where t is an element of $term_{mono}$ then the predicate $\Sigma \Vdash t : \tau$ implies that t has no type variables: t is monomorphic.

2.2 Proof Tasks

Proof tasks represent sequents in higher-order logic, they are formed from two sets of premises: a set of hypotheses and a set of goals. A *premise* is a pair of the form ' $P : t$ ' composed of an *ident* and a $term_{poly}$ representing a formula.

Definition 3 (Proof Task). Let I be a type signature, Σ be a signature, Γ and Δ be sets of premises. Proof tasks are denoted by $I \mid \Sigma \mid \Gamma \vdash \Delta$ which represents the sequent where goals, given by Δ , and hypotheses, given by Γ , are written in the signature Σ with types in I . We allow ourselves to omit I and, possibly, Σ , when they are clear from the context.

A task $T := I \mid \Sigma \mid \Gamma \vdash \Delta$ is said to be *well-typed* when every premise $P : t$ from Γ or Δ is such that $I \mid \Sigma \Vdash t : prop$. The *validity* of a task is only defined when it is well-typed. In this case, the task T is said to be valid when every model of I, Σ and every formula in Γ is also a model of some formula in Δ .

Example 4. Consider the task $I \mid \Sigma \mid \Gamma \vdash \Delta$ with

$$\begin{aligned}
I &:= \text{color} : 0, \text{set} : 1 \\
\Sigma &:= \text{red} : \text{color}(), \text{green} : \text{color}(), \text{blue} : \text{color}(), \\
&\quad \text{empty} : \text{set}(\alpha), \text{add} : \alpha \rightsquigarrow \text{set}(\alpha) \rightsquigarrow \text{set}(\alpha), \\
&\quad \text{mem} : \alpha \rightsquigarrow \text{set}(\alpha) \rightsquigarrow \text{prop} \\
\Gamma &:= H_1 : \Pi\alpha. \forall x : \alpha. \forall y : \alpha. \forall s : \text{set}(\alpha). \text{mem } x \ s \Rightarrow \text{mem } x \ (\text{add } y \ s), \\
&\quad H_2 : \Pi\alpha. \forall x : \alpha. \forall s : \text{set}(\alpha). \text{mem } x \ (\text{add } x \ s) \\
\Delta &:= G : \text{mem } \text{green} \ (\text{add } \text{red} \ (\text{add } \text{green} \ \text{empty}))
\end{aligned}$$

This task defines the types *color* and *set* with associated symbols *red*, *green*, *blue*, *empty*, *add* and *mem*. The type symbol declaration $\text{set} : 1$ defines a type symbol *set* of arity 1 for polymorphic sets. The signature declaration $\text{add} : \alpha \rightsquigarrow \text{set}(\alpha) \rightsquigarrow \text{set}(\alpha)$ allows us to declare a function that can be used to add an element of any type to a set containing elements of the same type. This task also defines hypotheses such that the predicate *mem* holds if its first argument is contained in the second argument. For instance, the hypothesis H_2 is applicable to sets of any type, and states that every set contains the element that has just been added to it. With the given goal, this task is valid.

2.3 Logical Transformations

A *logical transformation* is a function that takes a task as input and returns a list of tasks. Lists are denoted by separating their elements with semicolons. We say that a transformation is applied on an *initial task* and returns *resulting tasks*. A transformation can fail, in this case the whole process is terminated and we do not have to prove the correctness of the application. If a transformation succeeds, we want the verification to be based on the validity of the resulting tasks and to be able to forget about the initial task. This is why we say that a *transformation application is correct* when the validity of each resulting task implies the validity of the initial task. In Example 1, the transformation *instantiate* returns the initial task modified by adding the instantiated hypothesis to it. When a transformation application is correct, it only remains to prove that this resulting task is valid in order to make sure that the initial task is also valid. This is our approach: we certify transformation applications, thus relating initial and resulting tasks.

3 Certificates

To verify a transformation, we instrument it to produce a certificate and check each application of the transformation thanks to the corresponding certificate. We first define our own certificate format with the goal of making the verification of those certificates as easy as possible. We show how to improve modularity and ease of use in a second time in Section 5.1.

3.1 Syntax

An excerpt of the recursive definition of certificates is given in Figure 1. More certificates will be detailed on their own in Section 4 and the others won't be presented in this article for brevity. We call these certificates the *kernel certificates*. To make certificates easier to check, we design them in such a way that they are very precise. For example, the Boolean values indicate whether the premise is an hypothesis

```

cert ::= KHole(task)
      | KTrivial(bool, ident)
      | KAssert(ident, termpoly, cert, cert)
      | KSplit(bool, termmono, termmono, ident, cert, cert)
      | KDestruct(bool, termmono, termmono, ident, ident, ident, cert)
      | KIntroQuant(bool, type, termmono, ident, ident, cert)
      | KInstQuant(bool, type, termmono, ident, ident, termmono, cert)
      | KIntroType(termpoly, ident, ident, cert)
      | KInstType(termpoly, ident, ident, type, cert)
      | ...

```

Figure 1: Definition of Kernel Certificates (excerpt)

or a goal. Moreover, the kernel certificates have voluntarily been kept as elementary as possible and this makes it easier to trust them. In particular, this approach makes it easier to check every case (about 20 of them) when proving by induction a property of correctness of kernel certificates, as it is done in paragraph 5.3.3.

The certificates can contain tasks and each KHole node carries one of those tasks. When c is a certificate, *the leaves of c* designate the list of all tasks obtained by collecting them (in the KHole nodes) when doing an in-order traversal of the certificate tree. The leaves of a certificate are meant to be, in the end, the resulting tasks of the transformation it is certifying. A certificate with holes associated to a transformation application can be checked without needing to wait for the proof of the returned tasks to fill its holes, and this is what makes our certificates original. This design choice has been guided by our will for modularity: we want to progressively certify logical transformations.

3.2 Semantics

The semantics of certificates is defined by a binary predicate $T \downarrow c$, linking the initial task T to a certificate c . Informally, the predicate $T \downarrow c$ holds if c represents a proof of the fact that the validity of the leaves of c implies the validity of T . In Figure 2, we give the rules that cover the certificates from Figure 1. In this sense, the rules are only an excerpt of the complete set of rules (given in Appendix B) defining the predicate $T \downarrow c$. Notice that some of the certificates have dual rules for the hypotheses and the goals.

The certificate KHole is used to validate transformations that have resulting tasks and can be used directly for the identity transformation. The certificate KTrivial is used to validate a transformation application that has no resulting task when the initial task contains a trivial premise. The KAssert certificate allows to introduce a cut on a polymorphic formula. Remember that the side condition implies that this formula cannot have free type variables. The certificates KSplit and KDestruct are used to validate a transformation application that first splits a premise H . Certificate KIntroQuant is used to introduce the variable of a quantified premise. The certificate KInstQuant is used to instantiate a quantified premise with a term and the side condition ensures that this term is monomorphic. The certificates KIntroType and KInstType are used to deal with type-quantified premises.

Example 5. Let T and T_{inst} denote, respectively, the initial and the resulting task from Example 1, and let H_{inst} be the name of the new instantiated hypothesis. Suppose that symbol *plus*, symbol *mult* and type

$$\begin{array}{c}
\overline{\Gamma \vdash \Delta \downarrow \text{KHole}(\Gamma \vdash \Delta)} \quad \overline{\Gamma, H : \perp \vdash \Delta \downarrow \text{KTrivial}(\text{false}, H)} \quad \overline{\Gamma \vdash \Delta, G : \top \downarrow \text{KTrivial}(\text{true}, G)} \\
\\
\frac{\Sigma \mid \Gamma \vdash \Delta, P : t \downarrow c_1 \quad \Sigma \mid \Gamma, P : t \vdash \Delta \downarrow c_2 \quad \Sigma \Vdash t : \text{prop}}{\Sigma \mid \Gamma \vdash \Delta \downarrow \text{KAssert}(P, t, c_1, c_2)} \\
\\
\frac{\Gamma, H : t_1 \vdash \Delta \downarrow c_1 \quad \Gamma, H : t_2 \vdash \Delta \downarrow c_2}{\Gamma, H : t_1 \vee t_2 \vdash \Delta \downarrow \text{KSplit}(\text{false}, t_1, t_2, H, c_1, c_2)} \quad \frac{\Gamma \vdash \Delta, G : t_1 \downarrow c_1 \quad \Gamma \vdash \Delta, G : t_2 \downarrow c_2}{\Gamma \vdash \Delta, G : t_1 \wedge t_2 \downarrow \text{KSplit}(\text{true}, t_1, t_2, G, c_1, c_2)} \\
\\
\frac{\Gamma, H_1 : t_1, H_2 : t_2 \vdash \Delta \downarrow c}{\Gamma, H : t_1 \wedge t_2 \vdash \Delta \downarrow \text{KDestruct}(\text{false}, t_1, t_2, H, H_1, H_2, c)} \\
\\
\frac{\Gamma \vdash \Delta, G_1 : t_1, G_2 : t_2 \downarrow c}{\Gamma \vdash \Delta, G : t_1 \vee t_2 \downarrow \text{KDestruct}(\text{true}, t_1, t_2, G, G_1, G_2, c)} \\
\\
\frac{\Sigma, y : \tau \mid \Gamma, H : t[x \mapsto y] \vdash \Delta \downarrow c \quad y \text{ is fresh w.r.t. } \Sigma, \Gamma, \Delta, t}{\Sigma \mid \Gamma, H : \exists x : \tau. t \vdash \Delta \downarrow \text{KIntroQuant}(\text{false}, \tau, \lambda x : \tau. t, H, y, c)} \\
\\
\frac{\Sigma, y : \tau \mid \Gamma \vdash \Delta, G : t[x \mapsto y] \downarrow c \quad y \text{ is fresh w.r.t. } \Sigma, \Gamma, \Delta, t}{\Sigma \mid \Gamma \vdash \Delta, G : \forall x : \tau. t \downarrow \text{KIntroQuant}(\text{true}, \tau, \lambda x : \tau. t, G, y, c)} \\
\\
\frac{\Sigma \mid \Gamma, H_1 : \forall x : \tau. t, H_2 : t[x \mapsto u] \vdash \Delta \downarrow c \quad \Sigma \Vdash u : \tau}{\Sigma \mid \Gamma, H_1 : \forall x : \tau. t \vdash \Delta \downarrow \text{KInstQuant}(\text{false}, \tau, \lambda x : \tau. t, H_1, H_2, u, c)} \\
\\
\frac{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau. t, G_2 : t[x \mapsto u] \downarrow c \quad \Sigma \Vdash u : \tau}{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau. t \downarrow \text{KInstQuant}(\text{true}, \tau, \lambda x : \tau. t, G_1, G_2, u, c)} \\
\\
\frac{I, \iota : 0 \mid \Sigma \mid \Gamma \vdash \Delta, G : t[\alpha \mapsto \iota] \downarrow c \quad \iota \notin I}{I \mid \Sigma \mid \Gamma \vdash \Delta, G : \Pi \alpha. t \downarrow \text{KIntroType}(\Pi \alpha. t, G, \iota, c)} \\
\\
\frac{\Gamma, H_1 : \Pi \alpha. t, H_2 : t[\alpha \mapsto \tau] \vdash \Delta \downarrow c \quad \tau \text{ has no type variables}}{\Gamma, H_1 : \Pi \alpha. t \vdash \Delta \downarrow \text{KInstType}(\Pi \alpha. t, H_1, H_2, \tau, c)}
\end{array}$$

Figure 2: Certificate Rules (excerpt)

symbol *int* are defined in the signature and type signature, then we have $T \downarrow c$ with

$$\begin{aligned}
c := & \text{KInstQuant}(\text{false}, \text{int}, \lambda i : \text{int}. p \text{ (plus (mult 4 } i) 1), \\
& H, H_{\text{inst}}, \text{int}, \text{plus (mult } x \text{ } x), \text{KHole}(T_{\text{inst}}))
\end{aligned}$$

Proposition 6. *If T is well-typed then every task in a derivation $T \downarrow c$ is also well-typed.*

We also assume that transformations are always applied to well-typed initial tasks and produce well-typed resulting tasks (or they fail), so that every task we consider from now on is implicitly assumed to be well-typed.

Theorem 7 (Certificate Correctness). *If $T \downarrow c$ then the validity of each leaf of c implies the validity of T .*

Proof. By induction on $T \downarrow c$. □

3.3 Design Choices

The certificate rules are taken from the sequent calculus LK rules with modifications for two reasons. First, we want the production of certificates to be more natural. This is why the name `KSpl` is well-suited for a transformation application that, from the initial task $\Gamma, H : t_1 \vee t_2 \vdash \Delta$, returns $\Gamma, H : t_1 \vdash \Delta$ and $\Gamma, H : t_2 \vdash \Delta$. Indeed, it would be confusing to say that, from this initial task, the transformation does the left introduction of the disjunction. Second, we want to be able to implement a checker of certificates following these rules. To this end, instead of asking the checkers to find the names that were chosen by the transformation, we register these names in the certificates. For example, the `KIntroQuant` certificate mentions the name y of the new fresh variable that is being introduced and this is reflected in the corresponding rules.

3.4 Certifying Transformations and Composition

Definition 8 (Certifying transformation). *A certifying transformation is a transformation that, applied on an initial task, produces, on top of a list L of resulting tasks, a certificate c such that L is the leaves of c . We say that we instrumented the transformation to produce a certificate.*

Composing transformations is useful to define a transformation from simpler ones. To compose certifying transformations, one also needs to be able to substitute certificates, that is, to replace a `KHole` in one certificate with another certificate. This composition allows for a modular development of certifying transformations.

4 Adding Support for Interpreted Theories

For now, our formalism implicitly makes the assumption that every symbol is uninterpreted: they are taken as fresh new symbols for every task. Still, we want some symbols (such as equality or arithmetic operations) to have a fixed interpretation. Moreover, some transformations, like induction, use specific theories and we need to add certificate steps to be able to certify them.

To make sure that the interpretation is unique, we should not quantify over the interpreted symbols at the level of the tasks. Interpreted symbols are not part of the signature or type signature of tasks. This ensures that the interpretation stays the same for the initial task and for the resulting tasks and this is enough to handle transformations on tasks that contain interpreted symbols. To handle transformations that deal with the interpreted symbols and use their properties, we extend our certificate format and add rules corresponding to their properties.

4.1 Polymorphic Equality

The polymorphic equality is interpreted. To obtain the usual properties of the equality, we add the certificates:

$$\begin{aligned} & \text{KEqRefl}(\text{term}_{\text{mono}}, \text{ident}) \\ & \text{KRewrite}(\text{bool}, \text{term}_{\text{mono}}, \text{term}_{\text{mono}}, \text{term}_{\text{mono}}, \text{ident}, \text{ident}, \text{cert}) \end{aligned}$$

and three kernel rules:

$$\begin{array}{c} \frac{}{\Sigma \mid \Gamma \vdash \Delta, G : x = x \downarrow \text{KEqRefl}(x, G)} \quad \frac{\Gamma, H : a = b, P : t[b] \vdash \Delta \downarrow c}{\Gamma, H : a = b, P : t[a] \vdash \Delta \downarrow \text{KRewrite}(\text{false}, a, b, t, P, H, c)} \\[10pt] \frac{\Gamma, H : a = b \vdash \Delta, P : t[b] \downarrow c}{\Gamma, H : a = b \vdash \Delta, P : t[a] \downarrow \text{KRewrite}(\text{true}, a, b, t, P, H, c)} \end{array}$$

When t is a function of the form $\lambda x. u$, we write $t[u']$ for the substitution $u[x \mapsto u']$. These rules deal with the reflexivity of equality and the rewriting under context. They are sufficient to obtain the standard properties of equality: symmetry, transitivity, and congruence.

Application to the `rewrite` Transformation. The Why3 `rewrite` transformation is a powerful transformation that can rewrite terms modulo an equality that is under implications and universal quantifiers. It looks for a substitution to match the left-hand side of the given equality to rewrite it as the right-hand side following this substitution. Moreover, it allows rewriting from right to left instead. We instrument this transformation in the general case: using the found substitution, we define certificates to introduce in turns implications and universal quantifiers in a temporary hypothesis, to then apply symmetry of equality if needed and to use this equality to rewrite the target premise and finally remove the temporary hypothesis.

4.2 Integers

The type symbol `int`, integer literals and the operator symbols $+$, $*$, $-$, $>$, $<$, \geq and \leq are interpreted. To be able to certify a transformation that performs an induction on integers, we add a certificate $\text{KInduction}(\text{ident}, \text{term}_{\text{mono}}, \text{term}_{\text{mono}}, \text{ident}, \text{ident}, \text{ident}, \text{cert}, \text{cert})$ to the kernel certificates with one rule for strong induction:

$$\frac{\begin{array}{c} i \text{ is fresh w.r.t. } \Gamma, \Delta, t \quad \Sigma \Vdash i : \text{int} \quad \Sigma \Vdash a : \text{int} \\ \Gamma, H_i : i \leq a \vdash \Delta, G : t[i] \downarrow c_{\text{base}} \quad \Gamma, H_i : i > a, H_{\text{rec}} : \forall n : \text{int}, n < i \Rightarrow t[n] \vdash \Delta, G : t[i] \downarrow c_{\text{rec}} \end{array}}{\Gamma \vdash \Delta, G : t[i] \downarrow \text{KInduction}(i, a, t, G, H_i, H_{\text{rec}}, c_{\text{base}}, c_{\text{rec}})}$$

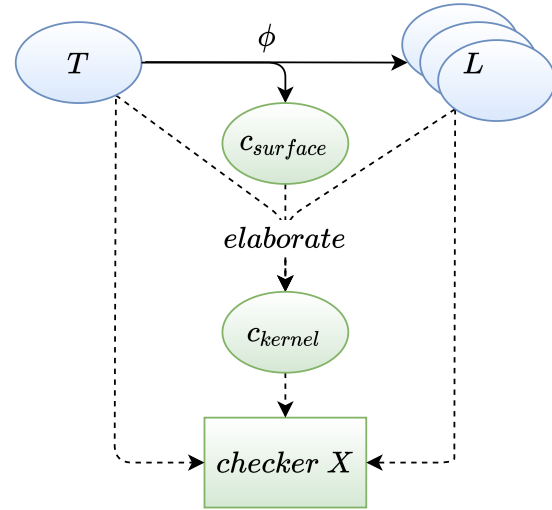
Application to the `induction` Transformation. The Why3 `induction` transformation can be called even if the context depends on the integer on which the induction is done and, in this case, the induction hypothesis takes into account this context. This transformation is instrumented to produce the certificate that first puts in the goal the premises that depend on the integer on which the induction is done, applies the `KInduction` certificate, and then, for the two resulting tasks, introduces those premises.

5 Certificate Checker

Let us consider an application of a certifying transformation ϕ on an initial task T that produces a certificate c and a list of resulting tasks L . To verify that this application is correct, c , T and L are provided as input to some checker. If the checker validates this application then the transformation returns L , otherwise it fails. In this section, we present the elaboration of certificates, a preprocessing step realized before calling any checker. We then show how to define such checkers and be confident in their answers.

5.1 Surface Certificates and Elaboration

Making a transformation certifying can be difficult, especially if the transformation has to produce a low level certificate. To facilitate this process, we define *surface certificates* that are easier to use than kernel certificates. We instrument the transformations to produce surface certificates instead of kernel certificates and implement an *elaboration* procedure to translate them into kernel certificates. In order to obtain the needed data to produce the kernel certificate, before calling a checker, the elaboration procedure is called with the initial task and the resulting tasks as input. Because the proof of correctness of certificates is done on the kernel certificates, we can define more complex surface certificates, as long as we are able to define the elaboration for them. Another advantage of surface certificates over kernel certificates is that they are less verbose, making them easier to produce.



Example 9. The surface certificate $\text{SSplit}(\text{ident}, \text{cert}, \text{cert})$ is elaborated into the kernel certificate KSplit . Suppose that a certifying transformation applied on initial task $T := H : x_1 \vee x_2 \vdash G : x$ returns the list $T_1; T_2$ with $T_1 := H : x_1 \vdash G : x$ and $T_2 := H : x_2 \vdash G : x$ and the surface certificate

$$\text{SSplit}(H, \text{KHole}(T_1), \text{KHole}(T_2))$$

The elaboration produces a kernel certificate indicating which formulas it is applied to (x_1 and x_2) and that H is not a goal (Boolean false):

$$\text{KSplit}(\text{false}, x_1, x_2, H, \text{EHole}(T_1), \text{EHole}(T_2))$$

We can define every surface certificate that we find convenient. For now there are about 10 more surface certificates than kernel certificates. Among them, there are SEqSym and SEqTrans for symmetry and transitivity of equality and SConstruct described in the following example.

Example 10. We define the surface certificate $\text{SConstruct}(\text{ident}, \text{ident}, \text{ident}, \text{cert})$ to validate a transformation application that first merges two premises into one. More precisely, by writing c' a certificate c

that has been elaborated, we should be able to derive the following rules:

$$\frac{\Gamma, P : t_1 \wedge t_2 \vdash \Delta \downarrow c'}{\Gamma, P_1 : t_1, P_2 : t_2 \vdash \Delta \downarrow \text{SConstruct}(P_1, P_2, P, c)'} \quad \frac{\Gamma \vdash \Delta, P : t_1 \vee t_2 \downarrow c'}{\Gamma \vdash \Delta, P_1 : t_1, P_2 : t_2 \downarrow \text{SConstruct}(P_1, P_2, P, c)'}$$

The *SConstruct* certificate does not have a corresponding kernel certificate. Instead, it is replaced during elaboration by a combination of the *KAssert* certificate on formula $t_1 \wedge t_2$ and other propositional certificates, notably *KDestruct*. Notice that we need to have access to the formula $t_1 \wedge t_2$, which is precisely the point of the elaboration and why we could not define directly *SConstruct* as a combination of surface certificates.

5.2 OCaml Checker

We implemented two checkers, the first one is written in OCaml and follows a computational approach: it is based on a function *ccheck* that is called with the certificate *c* and initial task *T* and interprets the certificate as instructions to derive tasks such that their validity implies the validity of *T*, verifying in the end that the derived tasks are the leaves of *c*. The checker validates the application when this function returns *true*.

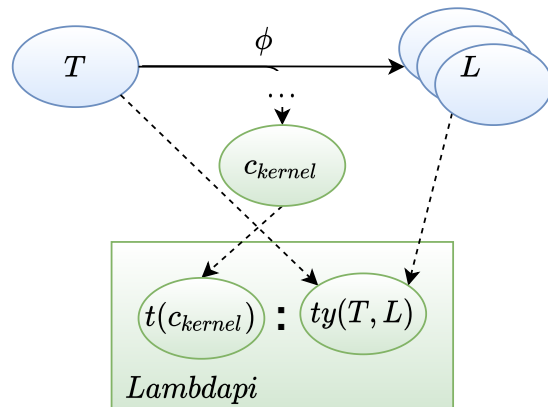
Theorem 11. *Let c be the certificate produced by applying a certifying transformation on task T . If $ccheck\ c\ T = true$ then $T \downarrow c$.*

The OCaml-checker definition follows closely the semantics of the certificates. For this reason, the proof of the previous theorem is relatively straightforward to do on paper. Together with Theorem 7, this guarantees that each application of a transformation that is checked by the OCaml checker is correct.

5.3 Lambdapi Checker

The second checker uses Lambdapi/Dedukti [3], a proof assistant based on a type checker for the $\lambda\Pi$ -Calculus modulo rewriting which extends the $\lambda\Pi$ -Calculus formalism with custom rewriting rules. This checker has two main advantages over the OCaml checker: (1) this checker uses an off-the-shelf proof assistant, benefiting from its reliability and its features, such as the ability to define custom rewriting rules; (2) this checker is proven to be correct, and this proof is machine-checked.

Every time a transformation is called, a Lambdapi proof is generated, and this proof is then checked by the type checker. More precisely, we define a shallow embedding of proof tasks in Lambdapi: a proof task *T* is encoded as a Lambdapi formula \hat{T} . In the diagram, a certifying transformation is applied to the initial task *T* and produces the resulting tasks $L := T_1; \dots; T_n$ and a certificate *c*, elaborated as c_{kernel} . Our tool then generates the type $ty(T, L)$ which is the Lambdapi formula stating that $\hat{T}_1, \hat{T}_1, \dots$, and \hat{T}_n implies \hat{T} , type that we call the *application correctness type*. Finally, we check that the application is correct by generating a proof term $t(c_{kernel})$ and asking Lambdapi to check that $t(c_{kernel})$ has type $ty(T, L)$.



This approach assumes that we trust `Lambdapi`, its type checker and the embedding of proof tasks (paragraphs 5.3.1 and 5.3.2). However the proof term generation is not contained in the trust base: the way the term is obtained does not matter as long as it has the requested type. Additionally, we have defined terms in `Lambdapi` for each certificate (paragraph 5.3.3), including certificates from interpreted theories (paragraph 5.3.4). These terms have been checked by `Lambdapi` to have the expected type so this gives us a machine-checked proof of Theorem 7.

Theorem 12. *Consider a transformation application that from task T returns the list of tasks L . If the application correctness type $\text{ty}(T, L)$ is inhabited, then this application is correct.*

5.3.1 Shallow Embedding

In `Lambdapi`, we define the translation of a task validity by quantifying over type symbols and function symbols, thus making these declarations explicit. We are able to quantify in this way, both at the level of types and at the level of terms, by using an encoding of the Calculus of Constructions [26] (written CoC) inside `Lambdapi`. We obtain a formal description of the whole task which allows us to state and prove the correctness of a transformation application. For the system to stay coherent we should be careful when adding rewriting rules and axioms (symbols in `Lambdapi`). We make use of an existing CoC encoding inside `Lambdapi` [12] to which we add the axiom of excluded middle. This encoding is also a shallow embedding inside `Lambdapi`, so we also get a shallow embedding of our language inside `Lambdapi`. In this way, we do not need to explicitly mention the context of proof and to handle it through inversion and weakening lemmas, which would make the method impracticable.

We are able to translate our whole formalism using this embedding. The translation of a term, a type or a task t inside `Lambdapi` is denoted \widehat{t} . We use exclusively the CoC syntax to describe this translation: we write $\forall x : A, B$ for the dependent product, $A \rightarrow B$ when B does not depend on x , $\lambda x : A, B$ for the abstraction and omit A when it can easily be inferred. The sorts are *Type* and *Kind*, with *Type* being of type *Kind*. To translate the terms, we use an impredicative encoding [30]. Here is an excerpt of this encoding:

$$\begin{aligned} \widehat{\text{prop}} &:= \text{Type} & \widehat{\perp} &:= \forall C : \text{Type}, C \\ \widehat{t_1 \wedge t_2} &:= \forall C : \text{Type}, (\widehat{t_1} \rightarrow \widehat{t_2} \rightarrow C) \rightarrow C & \widehat{\top} &:= \widehat{\perp} \rightarrow \widehat{\perp} \\ \widehat{t_1 \vee t_2} &:= \forall C : \text{Type}, (\widehat{t_1} \rightarrow C) \rightarrow (\widehat{t_2} \rightarrow C) \rightarrow C & \widehat{\neg t} &:= \widehat{t} \rightarrow \widehat{\perp} \end{aligned}$$

We note $\widehat{\neg} u$ for $u \rightarrow \widehat{\perp}$ such that $\widehat{\neg} t = \widehat{\neg} \widehat{t}$ and we extend this notation to the conjunction and the disjunction. Note that $\widehat{\top}$ is inhabited by $\lambda c, c$.

5.3.2 Translating Tasks

Let us give the translation of a task, where Type^n denotes the n -ary function over *Type* (for example, Type^2 is $\text{Type} \rightarrow \text{Type} \rightarrow \text{Type}$).

Task $I \mid \Sigma \mid \Gamma \vdash \Delta$ with :	Corresponding <code>Lambdapi</code> term:
$I := t_1 : i_1, \dots, t_m : i_m$	$\forall t_1 : \text{Type}^{i_1}, \dots, \forall t_m : \text{Type}^{i_m},$
$\Sigma := f_1 : \tau_1, \dots, f_n : \tau_n$	$\forall f_1 : \widehat{\tau_1}, \dots, \forall f_n : \widehat{\tau_n},$
$\Gamma := H_1 : t_1, \dots, H_k : t_k$	$\widehat{t_1} \rightarrow \dots \rightarrow \widehat{t_k} \rightarrow$
$\Delta := G_1 : u_1, \dots, G_l : u_l$	$\widehat{\neg} u_1 \rightarrow \dots \rightarrow \widehat{\neg} u_l \rightarrow \widehat{\perp}$

Note that for polymorphic symbols, we need to declare them with extra type parameters and to apply them to the appropriate type in the translation.

5.3.3 Proof Term

For each kernel rule, we associate a *Lambdapi* type and define a term that has this type. When building a proof term, we first introduce the identifiers of the resulting tasks and the identifiers of the type symbols, function symbols and name of the premises of the initial task. Then, we translate the whole certificate using the terms having the associated types. The *KHole* certificate is special and does not have a fixed associated type. Instead, it is translated as the identifier of the task it contains applied to its symbols and premises following the same order that they have been introduced in. Assuming that our encoding is correct, the fact that we define a term for every rule of the certificate semantics gives us a machine-checked proof of the certificate correctness, Theorem 7.

To produce the proof term, we benefit from the elaboration of certificate in two ways. First, the fact that the kernel certificates are elementary also facilitates the definition of the terms corresponding to a kernel rule. Second, each kernel certificate comes with additional data that we are able to use to define such terms.

Example 13. For the *KSplit* rule presented in Figure 2, we define a *Lambdapi* term *split* that has the associated type $\forall t_1 : \text{Type}, \forall t_2 : \text{Type}, (t_1 \rightarrow \hat{\perp}) \rightarrow (t_2 \rightarrow \hat{\perp}) \rightarrow t_1 \hat{\vee} t_2 \rightarrow \hat{\perp}$. We check the application of Example 9 by verifying that the type

$$\begin{aligned} & (\forall x_1, \forall x, x_1 \rightarrow \hat{\wedge} x \rightarrow \hat{\perp}) \rightarrow \\ & (\forall x_2, \forall x, x_2 \rightarrow \hat{\wedge} x \rightarrow \hat{\perp}) \rightarrow \\ & \forall x_1, \forall x_2, \forall x, x_1 \hat{\vee} x_2 \rightarrow \hat{\wedge} x \rightarrow \hat{\perp} \end{aligned}$$

is inhabited by the term

$$\begin{aligned} & \lambda s_1, \lambda s_2, \lambda x_1, \lambda x_2, \lambda x, \lambda H, \lambda G, \\ & \text{split } x_1 \ x_2 \ (\lambda H, s_1 \ x_1 \ x \ H \ G) \ (\lambda H, s_2 \ x_2 \ x \ H \ G) \end{aligned}$$

Notice that *split* takes the formulas it is applied to as arguments (x_1 and x_2) and that those formulas have been found by elaborating the certificate.

5.3.4 Encoding of Interpreted Theories

In *Lambdapi*, interpreted symbols are first declared in the preamble. When interpreted symbols have corresponding certificate rules, we need to use the properties of those symbols to prove that the types associated to these rules are inhabited. Instead of declaring such symbols, we define them, which allows us to prove the needed properties. Since our *Lambdapi* development is included in the trusted code base of the *Lambdapi* checker, we make sure to only add axioms and rewrite rules when necessary.

Polymorphic Equality. We define the equality in *Lambdapi* using the Leibniz definition of equality: two terms t_1 and t_2 of type τ are equal when $\forall Q : \tau \rightarrow \text{Type}, Q \ t_1 \rightarrow Q \ t_2$. Note that the context of rewriting in the *KRewrite* rules is explicitly given as a function. We use this function, translated as a *Lambdapi* function, to apply it to the Leibniz equality when writing a proof term for a *KRewrite* certificate.

number of variables	5	10	15	20	25	50	100	200	400	800
transformation time (sec)	~ 0	~ 0	0.008	0.016	0.020	0.080	0.29	1.21	5.5	25
kernel certificate size (kB)	2.1	5.8	12	19	28	85	270	950	3500	13000
OCaml checker time (sec)	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0	0.020	0.084	0.35
Lambdapi checker time (sec)	0.072	0.25	0.80	2.0	4.2	48	660	-	-	-

Figure 3: Tests on Propositional Tasks

Integers. We define the integer type and usual integer operators in Lambdapi. First, we define binary positive integers and binary negative integers. Integers are then either 0, positive or negative. We use rewrite rules to define those data types which should be understood as algebraic data types [5]. From these definitions, we get a simple induction principle that we use to define a Lambdapi term corresponding to the stronger induction principle described by the rule of the certificate KInduction.

6 Experimental Evaluation

The goal of this article is to provide a practical framework to render logical transformations certifying. We apply the framework to Why3 and show that the approach does not have an inherent problem of efficiency. More importantly, we show that it is expressive enough to allow us to render complex transformations certifying.

6.1 Tests and Benchmarks

We defined simple certifying transformations (about 15 of them) to test every certificate. We also defined a more complex transformation called `blast` meant to discharge tautological propositional tasks. This transformation decomposes every logical connector appearing at the head of formulas before calling itself recursively. Rendering this transformation certifying required using the composition of certifying transformations. We evaluated the efficiency of our checkers by applying this transformation on problems of increasing size in Figure 3. The problem with n propositional variables is to verify that:

$$p_1 \Rightarrow (p_1 \Rightarrow p_2) \Rightarrow \dots (p_{n-1} \Rightarrow p_n) \Rightarrow p_n$$

We notice that the size of the kernel certificates is not linear with respect to the number of variables. This is due to the fact that, contrary to the surface certificates, the kernel certificates contain the formulas they are applied to. Looking at the OCaml checker, our approach does not seem to have an inherent problem of efficiency as the overhead it adds to the transformation is negligible. On the other hand, the Lambdapi checker seems to be much slower. Performances of Lambdapi have already been improved for our purposes [16–20], and we believe they could be further improved in future versions. We could also modify our checker to help Lambdapi, for example by abstracting away big formulas. We leave this for future work.

6.2 Applications

We evaluated our method by applying it at different levels. When rendering the existing transformations `rewrite` and `induction` certifying, we found that it is well-suited to add interpreted theories. When transformations do not specifically deal with the theory in question, we do not need to extend our certificate format, while, in general, the duo surface/kernel certificates allows us to only define a minimal set of kernel rules, even if it means deriving more surface certificates. By defining the `Lambdapi` checker, we gave a machine-checked proof of the rules of our certificates which gives us confidence in our certificates and their semantics.

6.3 Better Understanding of Transformations

This work has led to a better understanding of transformations. On one hand, instrumenting a transformation to produce an appropriate certificate requires to understand why each application of this transformation is correct. Additionally, once this is done, reading the certificate gives us another way to understand why a particular transformation application is correct. On the other hand, this work had led to the definition of the semantics of tasks inside `Lambdapi` and the definition of the correction of a transformation in this setting.

In particular, type quantification is explicit in Section 2.1. For example, the formula $\Pi\alpha. (\forall x : \alpha. \forall y : \alpha. x = y) \vee \neg(\forall x : \alpha. \forall y : \alpha. x = y)$ means that every type α either has at most one element or it has more than one. This formula is provable but we cannot apply the certificate `KSplit` on such an hypothesis. By contrast, in `Why3`, the type quantification is implicit, and it is possible apply the `destruct` transformation on the hypothesis. This gives us two resulting tasks: one with an hypothesis which states that every type has at most one element, and the other with an hypothesis which states that every type has more than one element, both being contradictory. This bug [15] has been found in the transformation `destruct` when encoding proof tasks in `Lambdapi`; a similar bug was also found in the transformation `case`.

7 Related Work

To aid deductive program verification, a number of tools have been developed, based on proof assistants or independently from them. In the first case, the programming language on which the verification is done is built from dedicated libraries and definition of both the programming language and its logic inside the proof assistant. This is the case for example for the library `Iris` [27] built on top of `Coq` and that allows reasoning about concurrent, imperative programs or the library `AutoCorres` [23, 24] built on top of `Isabelle` and allowing to verify C programs. In such context, the correctness of the approach is based on the formal semantics of programs and on deduction rules established once and for all, which requires a large proof effort, thus limiting the flexibility of the language. In the second case, the tools developed are verifying annotated programs, and generate proof obligations that are discharged by automatic theorem provers such as SMT solvers. Examples of such tools are `Why3`, `Dafny`, `Viper`, `Frama-C` and `SPARK`. Even though they rely on strong fundamental bases, their particular implementations of such tools and some practical aspects such as their use of automatic theorem provers have not been machine-checked and can contain bugs. An exception is given by `F*` [31], whose encoding's correctness to SMT logic has been partially proved in `Coq` [1].

Our work lies in between these two approaches. On one hand logical transformations are similar to tactics used in proof assistants such as `Coq` [13], except that our transformations are considered part of the

trusted code base. On the other hand, logical transformations can be used automatically or interactively to help discharging proof obligations. We followed a skeptical approach extended with a preprocessing step (namely the elaboration of certificates) similarly to [10], except that our framework allows to check higher order proofs, and that the focus is put on the ease of production of certificates. Indeed, we aim at making it as easy as possible to render transformations certifying. We designed two checkers: one based on the reflexive approach, known to be very efficient [2, 25] and the other one based on a shallow embedding into the *Lambdapi* proof assistant. When using a shallow embedding, the correctness of the verification relies on the considered proof tool’s correctness which makes its proof much easier [9, 11].

8 Conclusion

We presented a framework to validate logical transformations based on a skeptical approach. When defining certificates, we put an emphasis on *modularity* by having certificates with holes and, with the notions of surface and kernel certificates, *ease of use* without compromising the checker’s verification. We combined all of these notions and applied them to *Why3* by implementing the certificate generation for various transformations and the certificate verification with two checkers. The first checker was written in OCaml and uses a computational approach which makes it very efficient while the second checker is based on *Lambdapi* and gives us formal guarantees to its correctness. We extended our work by adding the interpreted theories of the integers and of the polymorphic equality. This allowed us to instrument more complex and existing transformations to produce certificates, such as *induction* and *rewrite*. Finally, we validated our method during development and through tests and benchmarks.

Future Work. The current application of our method to *Why3* could be improved at different levels. The first idea is to instrument more transformations to produce certificates, with polymorphism elimination [7] and algebraic data type elimination being important challenges. As the number of certifying transformations increases, we also want to improve the efficiency of the verification. To do so, we consider two factors: first, we want to compress certificates on the fly when combining them; second, we want to improve the efficiency of the *Lambdapi* checker by allowing to reuse the context of proof that does not change. Additionally, we consider adding support for more interpreted theories, while keeping the number of axioms and rewrite rules added to *Lambdapi* to a minimum.

A long term goal is to increase trust in other parts of *Why3*. For example, we could improve trust when calling automatic theorem provers [2, 8] or improve trust in the proof task generation which would require to formalize the semantics of the *Why3* programming language [14].

Finally, our method is not specific to *Why3* and can be applied, in general, to certified logical encodings. In particular, existing (certifying) transformations could be used for encoding a proof assistant’s logic into an automatic theorem prover’s logic in order to benefit from both systems.

Acknowledgments. We are grateful to Alexandrina Korneva for the English proofreading and to Claude Marché, Chantal Keller and Andrei Paskevich for their constant support and their helpful suggestions.

References

- [1] Alejandro Aguirre (2016): *Towards a provably correct encoding from F^* to SMT*. Master’s thesis. Available at <https://prosecco.gforge.inria.fr/personal/hritcu/students/alejandro/report.pdf>.
- [2] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In: *First International Conference on Certified Programs and Proofs*, doi:10.1007/978-3-642-25379-9_12.
- [3] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard (2016): *Expressing theories in the $\lambda\Pi$ -calculus modulo theory and in the Dedukti system*. In: *22nd International Conference on Types for Proofs and Programs*. Available at <https://hal-mines-paristech.archives-ouvertes.fr/hal-01441751>.
- [4] Henk Barendregt & Erik Barendsen (2002): *Autarkic computations in formal proofs*. *Journal of Automated Reasoning*, doi:10.1023/A:1015761529444.
- [5] Frédéric Blanqui (2003): *Inductive Types in the Calculus of Algebraic Constructions*. Lecture Notes in Computer Science, doi:10.1007/3-540-44904-3_4.
- [6] François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich (2015): *Let’s Verify This with Why3*. *International Journal on Software Tools for Technology Transfer (STTT)*, doi:10.1007/s10009-014-0314-5.
- [7] François Bobot & Andrei Paskevich (2011): *Expressing Polymorphic Types in a Many-Sorted Language*, doi:10.1007/978-3-642-24364-6_7.
- [8] Sascha Böhme & Tjark Weber (2010): *Fast LCF-Style Proof Reconstruction for Z3*. In: *Interactive Theorem Proving*, doi:10.1007/978-3-642-14052-5_14.
- [9] Raphaël Cauderlier & Pierre Halmagrand (2015): *Checking Zenon Modulo Proofs in Dedukti*. In: *Proof eXchange for Theorem Proving*, doi:10.4204/EPTCS.186.7.
- [10] Zakaria Chihani, Dale Miller & Fabien Renaud (2013): *Checking Foundational Proof Certificates for First-Order Logic (Extended Abstract)*. In: *PxTP 2013. Third International Workshop on Proof Exchange for Theorem Proving*, doi:10.29007/7gnr.
- [11] Évelyne Contejean (2008): *Coccinelle, a Coq library for rewriting*. In: *Types*.
- [12] Denis Cousineau & Gilles Dowek (2007): *Embedding Pure Type Systems in the lambda-Pi-calculus modulo*. In: *Typed lambda calculi and applications*, doi:10.1007/978-3-540-73228-0_9.
- [13] David Delahaye (2000): *A tactic language for the system Coq*. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, doi:10.1007/3-540-44404-1_7.
- [14] Jean-Christophe Filliâtre (1999): *Preuve de programmes impératifs en théorie des types*. Ph.D. thesis. Available at <http://www.lri.fr/~filliatr/ftp/publis/these.ps.gz>.
- [15] Quentin Garchery (2021): *destruct/case transformations incorrectly handle polymorphic formulas*. <https://gitlab.inria.fr/why3/why3/-/issues/525/>.
- [16] Quentin Garchery (2021): *Performance slowdown with variable shadowing*. <https://github.com/Deducteam/lambdaapi/issues/565>.
- [17] Quentin Garchery (2021): *Performance with growing context*. <https://github.com/Deducteam/lambdaapi/issues/579>.
- [18] Quentin Garchery (2021): *Performances with both new variables and hypotheses in context*. <https://github.com/Deducteam/lambdaapi/issues/595>.
- [19] Quentin Garchery (2021): *Performances with linear propositional problem*. <https://github.com/Deducteam/lambdaapi/issues/649>.
- [20] Quentin Garchery (2021): *Performances with nested applications in context*. <https://github.com/Deducteam/lambdaapi/issues/584>.

- [21] Quentin Garchery (2021): *Why3 cert_pxtp branch*. Available at https://gitlab.inria.fr/why3/why3/-/blob/cert_pxtp/README_PXTP.md.
- [22] Quentin Garchery, Chantal Keller, Claude Marché & Andrei Paskevich (2020): *Des transformations logiques passent leur certificat*. In: *JFLA 2020 - Journées Francophones des Langages Applicatifs*, Gruissan, France. Available at <https://hal.inria.fr/hal-02384946>.
- [23] David Greenaway (2015): *Automated proof-producing abstraction of C code*. Ph.D. thesis, CSE, UNSW. Available at <http://unsworks.unsw.edu.au/fapi/datastream/unsworks:13743/SOURCE02?view=true>.
- [24] David Greenaway, Japheth Lim, June Andronick & Gerwin Klein (2014): *Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, doi:10.1145/2594291.2594296.
- [25] Benjamin Grégoire, Laurent Théry & Benjamin Werner (2006): *A Computational Approach to Pocklington Certificates in Type Theory*. In: *Functional and Logic Programming*, doi:10.1007/11737414_8.
- [26] Gérard P. Huet (1987): *The Calculus of Constructions: State of the Art*. In: *Foundations of Software Technology and Theoretical Computer Science*, doi:10.1007/3-540-18625-5_61.
- [27] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer & Lars Birkedal (2017): *The Essence of Higher-Order Concurrent Separation Logic*. In: *26th European Symposium on Programming Languages and Systems*, doi:10.1007/978-3-662-54434-1_26.
- [28] Stéphane Lescuyer (2011): *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et développement d'une tactique reflexive pour la demonstration automatique en coq)*. Ph.D. thesis, University of Paris-Sud, Orsay, France. Available at <https://tel.archives-ouvertes.fr/tel-00713668>.
- [29] Robin Milner (1978): *A theory of type polymorphism in programming*. *Journal of computer and system sciences*, doi:10.1016/0022-0000(78)90014-4.
- [30] Frank Pfenning & Christine Paulin-Mohring (1989): *Inductively Defined Types in the Calculus of Constructions*. *Lecture Notes in Computer Science*, doi:10.1007/BFb0040259.
- [31] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue & Santiago Zanella Béguelin (2016): *Dependent types and multi-monadic effects in F**. In: *Principles of Programming Languages*, doi:10.1145/2837614.2837655.

A Typing

The predicate $\Sigma \Vdash t : \tau$ holds when t has no free type variables and is of type τ in signature Σ and is formally defined by the following rules:

$$\begin{array}{c}
\frac{I, \iota : 0 \mid \Sigma \Vdash t[\alpha \mapsto \iota] : prop \quad \iota \notin I}{I \mid \Sigma \Vdash (\Pi \alpha. t) : prop} \quad \frac{\tau \text{ is a subtype of } \Sigma(x) \quad \tau \text{ has no type variables}}{\Sigma \Vdash x : \tau} \\
\\
\frac{}{\Sigma \Vdash \top : prop} \quad \frac{}{\Sigma \Vdash \perp : prop} \quad \frac{\Sigma \Vdash t : prop}{\Sigma \Vdash \neg t : prop} \quad \frac{\Sigma \Vdash t_1 : prop \quad \Sigma \Vdash t_2 : prop}{\Sigma \Vdash t_1 \text{ op } t_2 : prop} \\
\\
\frac{\Sigma \Vdash t_1 : \tau' \rightsquigarrow \tau \quad \Sigma \Vdash t_2 : \tau'}{\Sigma \Vdash t_1 t_2 : \tau} \quad \frac{\Sigma, x : \tau \Vdash t : prop \quad \tau \text{ has no type variables} \quad x \notin \Sigma}{\Sigma \Vdash (\forall x : \tau. t) : prop} \\
\\
\frac{\Sigma, x : \tau \Vdash t : prop \quad \tau \text{ has no type variables} \quad x \notin \Sigma}{\Sigma \Vdash (\exists x : \tau. t) : prop} \\
\\
\frac{\Sigma, x : \tau' \Vdash t : \tau \quad \tau' \text{ has no type variables} \quad x \notin \Sigma}{\Sigma \Vdash (\lambda x : \tau'. t) : \tau' \rightsquigarrow \tau}
\end{array}$$

B Certificate rules

For each kernel certificate appearing in this article, we give its corresponding rules. These rules are taken from the set of rules defining the predicate $T \downarrow c$.

$$\begin{array}{c}
\frac{}{\Gamma \vdash \Delta \downarrow \text{KHole}(\Gamma \vdash \Delta)} \quad \frac{}{\Gamma, H : \perp \vdash \Delta \downarrow \text{KTrivial}(false, H)} \quad \frac{}{\Gamma \vdash \Delta, G : \top \downarrow \text{KTrivial}(true, G)} \\
\\
\frac{\Sigma \mid \Gamma \vdash \Delta, P : t \downarrow c_1 \quad \Sigma \mid \Gamma, P : t \vdash \Delta \downarrow c_2 \quad \Sigma \Vdash t : prop}{\Sigma \mid \Gamma \vdash \Delta \downarrow \text{KAssert}(P, t, c_1, c_2)} \\
\\
\frac{\Gamma, H : t_1 \vdash \Delta \downarrow c_1 \quad \Gamma, H : t_2 \vdash \Delta \downarrow c_2}{\Gamma, H : t_1 \vee t_2 \vdash \Delta \downarrow \text{KSplit}(false, t_1, t_2, H, c_1, c_2)} \quad \frac{\Gamma \vdash \Delta, G : t_1 \downarrow c_1 \quad \Gamma \vdash \Delta, G : t_2 \downarrow c_2}{\Gamma \vdash \Delta, G : t_1 \wedge t_2 \downarrow \text{KSplit}(true, t_1, t_2, G, c_1, c_2)} \\
\\
\frac{\Gamma, H_1 : t_1, H_2 : t_2 \vdash \Delta \downarrow c}{\Gamma, H : t_1 \wedge t_2 \vdash \Delta \downarrow \text{KDestruct}(false, t_1, t_2, H, H_1, H_2, c)} \\
\\
\frac{\Gamma \vdash \Delta, G_1 : t_1, G_2 : t_2 \downarrow c}{\Gamma \vdash \Delta, G : t_1 \vee t_2 \downarrow \text{KDestruct}(true, t_1, t_2, G, G_1, G_2, c)}
\end{array}$$

$$\begin{array}{c}
 \frac{\Sigma, y : \tau \mid \Gamma, H : t[x \mapsto y] \vdash \Delta \downarrow c \quad y \text{ is fresh w.r.t. } \Sigma, \Gamma, \Delta, t}{\Sigma \mid \Gamma, H : \exists x : \tau. t \vdash \Delta \downarrow \text{KIntroQuant}(\text{false}, \tau, \lambda x : \tau. t, H, y, c)} \\
 \\
 \frac{\Sigma, y : \tau \mid \Gamma \vdash \Delta, G : t[x \mapsto y] \downarrow c \quad y \text{ is fresh w.r.t. } \Sigma, \Gamma, \Delta, t}{\Sigma \mid \Gamma \vdash \Delta, G : \forall x : \tau. t \downarrow \text{KIntroQuant}(\text{true}, \tau, \lambda x : \tau. t, G, y, c)} \\
 \\
 \frac{\Sigma \mid \Gamma, H_1 : \forall x : \tau. t, H_2 : t[x \mapsto u] \vdash \Delta \downarrow c \quad \Sigma \Vdash u : \tau}{\Sigma \mid \Gamma, H_1 : \forall x : \tau. t \vdash \Delta \downarrow \text{KInstQuant}(\text{false}, \tau, \lambda x : \tau. t, H_1, H_2, u, c)} \\
 \\
 \frac{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau. t, G_2 : t[x \mapsto u] \downarrow c \quad \Sigma \Vdash u : \tau}{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau. t \downarrow \text{KInstQuant}(\text{true}, \tau, \lambda x : \tau. t, G_1, G_2, u, c)} \\
 \\
 \frac{I, \iota : 0 \mid \Sigma \mid \Gamma \vdash \Delta, G : t[\alpha \mapsto \iota] \downarrow c \quad \iota \notin I}{I \mid \Sigma \mid \Gamma \vdash \Delta, G : \Pi \alpha. t \downarrow \text{KIntroType}(\Pi \alpha. t, G, \iota, c)} \\
 \\
 \frac{\Gamma, H_1 : \Pi \alpha. t, H_2 : t[\alpha \mapsto \tau] \vdash \Delta \downarrow c \quad \tau \text{ has no type variables}}{\Gamma, H_1 : \Pi \alpha. t \vdash \Delta \downarrow \text{KInstType}(\Pi \alpha. t, H_1, H_2, \tau, c)} \\
 \\
 \frac{}{\Sigma \mid \Gamma \vdash \Delta, G : x = x \downarrow \text{KEqRefl}(x, G)} \quad \frac{\Gamma, H : a = b, P : t[b] \vdash \Delta \downarrow c}{\Gamma, H : a = b, P : t[a] \vdash \Delta \downarrow \text{KRewrite}(\text{false}, a, b, t, P, H, c)} \\
 \\
 \frac{\Gamma, H : a = b \vdash \Delta, P : t[b] \downarrow c}{\Gamma, H : a = b \vdash \Delta, P : t[a] \downarrow \text{KRewrite}(\text{true}, a, b, t, P, H, c)} \\
 \\
 \frac{\begin{array}{c} i \text{ is fresh w.r.t. } \Gamma, \Delta, t \quad \Sigma \Vdash i : \text{int} \quad \Sigma \Vdash a : \text{int} \\ \Gamma, H_i : i \leq a \vdash \Delta, G : t[i] \downarrow c_{\text{base}} \quad \Gamma, H_i : i > a, H_{\text{rec}} : \forall n : \text{int}, n < i \Rightarrow t[n] \vdash \Delta, G : t[i] \downarrow c_{\text{rec}} \end{array}}{\Gamma \vdash \Delta, G : t[i] \downarrow \text{KInduction}(i, a, t, G, H_i, H_{\text{rec}}, c_{\text{base}}, c_{\text{rec}})}
 \end{array}$$

General Automation in Coq through Modular Transformations

Valentin Blot

LMF, Inria, Université Paris-Saclay*

Valentin.Blot@inria.fr

Louise Dubois de Prisque

LMF, Inria, Université Paris-Saclay*

Louise.Dubois-de-Prisque@inria.fr

Chantal Keller

LMF, Université Paris-Saclay*

Chantal.Keller@lri.fr

Pierre Vial

LMF, Inria, Université Paris-Saclay*

Pierre.Vial@inria.fr

Whereas proof assistants based on Higher-Order Logic benefit from external solvers' automation, those based on Type Theory resist automation and thus require more expertise. Indeed, the latter use a more expressive logic which is further away from first-order logic, the logic of most automatic theorem provers. In this article, we develop a methodology to transform a subset of Coq goals into first-order statements that can be automatically discharged by automatic provers. The general idea is to write modular, pairwise independent transformations and combine them. Each of these eliminates a specific aspect of Coq logic towards first-order logic. As a proof of concept, we apply this methodology to a set of simple but crucial transformations which extend the local context with proven first-order assertions that make Coq definitions and algebraic types explicit. They allow users of Coq to solve non-trivial goals automatically. This methodology paves the way towards the definition and combination of more complex transformations, making Coq more accessible.

1 Introduction

Interaction vs. automation. The Coq proof assistant allows us to prove theorems *interactively*: that is, given a Coq goal G (*i.e.*, a statement to prove), the user has to write a *proof* P consisting of inference steps with *ad hoc* premises and conclusion. Then, they can check whether the given proof is correct by writing `Qed`: the proof is then certified (or rejected) by a **logical kernel** implemented in OCaml which type-checks the proof-term constructed by the user. In practice, proof assistants support a limited form of *automation*: in Coq, the user can use keywords called *tactics*, which operate logical transformations on a goal and its hypotheses until solving it, *e.g.*, a tactic may try to apply some inference/typing rules as many times as possible and relieve the user of some bureaucratic work. An important idea to keep in mind is that tactics may use elaborate tools (including plugins, auxiliary softwares, *etc.*...) to produce whole proof terms, *including flawed ones*. However, eventually, **any proof term**, whether it is written by the user or produced by a tactic or an external tool, **will be type-checked by the kernel of Coq**, and this is how the trust in Coq lies in the one we have in its kernel.

Anyway, automation in Coq is limited: in most cases (including many trivial proofs), the user has to provide the bigger part of the structure and the steps of the proof: we will give a simple example just below.

This situation is strikingly different from *Automated Theorem Provers*, such as first-order solvers or **SMT solvers**, which find a proof without the user having to find the proof steps if the problem is expressed in a suited way. However, these automated provers have two main limitations:

*This work is funded by a Nomadic Labs-Inria collaboration.

	Automatic provers	Coq
Expressivity	First-order logic	CIC
Safety	Trust the whole software	Kernel for proof checking
Automation	Automatic proof	User-guided proof

Table 1: Pros (bold) and Cons of automatic provers and Coq

- One needs to trust their whole code, and not a small kernel.
- Most of them handle only **first-order logic (FOL)** whereas Coq is based on the **Calculus of Inductive Constructions (CIC)**, which is far richer: (1) CIC is polymorphic, and thus allows quantifying not only over objects (such as integers) but also on types, *e.g.*, in CIC, one may define polymorphic list concatenation `++` whose type is `forall (A : Type), list A → list A → list A` (2) CIC features dependent typing, so that types may depend on terms and other types, *e.g.*, `Vec A n` specifies the type of vectors of length `n` on the carrier `A`, and thus depends on the number `n` and the type `A` (3) CIC enables higher-order computation.

The logical expressiveness of CIC makes it possible to handle complex aspects of programming languages and to prove properties about programs in Coq, in particular fine-grained specifications that an automatic prover based on first-order logic could not understand. For instance, dependent typing allows the specification of equalities beyond decidable datatypes, *e.g.*, in Coq one may directly specify that two *functions* are equal with an equality `f = g`, which is not possible in general in automatic solvers. Yet, it is precisely because most automatic provers only tackle FOL that they admit a lot of powerful automation.

Improving the automation of Coq within first-order logic. To sum up the situation, on the one hand, we have Coq, which is based on CIC and is a rich specification language, but for that reason, it is difficult to automatize its proof search. On the other hand, automatic provers are based on a more limited logic (FOL) but feature extremely efficient proof-search heuristics. Moreover, the trust we have in the former is based on the implementation of a small, isolated kernel. This can be summarized in Table 1.

Such differences are to be expected, since proof assistants and automatic provers do not have the same purposes and uses. However, as it turns out, in Coq, even a *simple proof of first-order logic*, *e.g.*, on decidable datatypes, can be tedious and force the user to provide a lot of input, whereas an automatic prover would automatically find a proof. Introducing more automation in Coq would thus be very useful to spare the user some trivial parts of a proof.

Let us illustrate this with two examples.

Example 1 (Dealing with datatypes). We use the function `hd_error` from the `List` module of Coq Standard Library, of type `forall {A:Type}, list A → option A` (the curly brackets mean that `A` is an implicit argument, that is to say it can be omitted) defined by `hd_error l = Some x` when `l = x :: l0` (*i.e.*, `l` not empty, `x : A`, `l0` a list of elements of type `A`) and `hd_error [] = None`. We then prove:

```
Goal forall l (a:A), hd_error l = Some a → l <> nil.
```

in a context where `A` is a type variable¹. A typical Coq proof is:

¹This is handled in Coq thanks to the section mechanism. It allows the user to introduce section-local variables that can be used in other declarations in the section.

Proof.

```
intros l a H. intro H'. rewrite H' in H. simpl in H. discriminate H.
Qed.
```

The statement is straightforward, its proof relies on the fact that two different constructors always output different values, *e.g.*, `Some x` and `None` cannot be equal, neither `x :: l` and `[]`. Yet, in Coq, the user (especially a non-expert one) has to be highly precise in the way they compose their keywords, even though they would not even bother writing the proof on paper. It may even be more frustrating that this is a statement of first-order logic and as such, it would automatically be dealt with by a first-order prover, provided it knows about (1) the definition of the function `hd_error` and (2) the datatypes `list` and `option`.

Example 2 (Calling lemmas). In this example, we recall some of the annoyances met while using lemmas in Coq. We consider a Boolean search function:

```
Fixpoint search {A : Type} {H: CompDec A} (x : A) l :=
  match l with
  | [] => false
  | x0 :: l0 => eqb_of_compdec H x x0 || search x l0
end.
```

Not going into details, the implicit argument `H: CompDec A` specifies that `A` is a *decidable type*, *i.e.*, that equality is decidable on `A`. Boolean equality can then be computed with the function `eqb_of_compdec H`².

By induction, we prove `search_app : forall {A: Type} {H : CompDec A} (x: A)(l1 l2: list A), search x (l1 ++ l2) = (search x l1) || (search x l2)`. Now, let us consider a typical Coq proof of the following simple statement:

```
Lemma search_lemma : forall (x: Z) (l1 l2 l3: list Z),
  search x (l1 ++ l2 ++ l3) = search x (l3 ++ l2 ++ l1).
Proof.
  intros x l1 l2 l3. rewrite !search_app.
  rewrite orb_comm with (b1 := search x l3).
  rewrite orb_comm with (b1 := search x l2) (b2 := search x l1).
  rewrite orb_assoc. reflexivity.
Qed.
```

As expected, the proof uses the commutativity and the associativity of Boolean disjunction `||`, which can be found in the module `Coq.Bool.Bool` of the Standard Library. We start by using the lemma `search_app` 4 times to eliminate `++` from the statement, which can be done automatically with the `!` operator that rewrites as much as needed. However, the order of commutativity/associativity uses must be carefully chosen. Moreover, the instance of the bound variables of the commutativity lemma `forall b1 b2 : bool, b1 || b2 = b2 || b1` must be specified: the same proof without specifying them fails, since Coq would try to rewrite only the leftmost-outermost `||`, which would not work. Actually, although the user recognizes the proof as trivial, they must keep a keen eye at each step on the local proof context to determine which lemma must be used with which instances, and sometimes, they have to print lemmas to identify the names of bound variables they need to instantiate. All this may appear as a nuisance compared to writing a formal proof with a pencil and a paper and deter new users of Coq.

²There are multiple ways of representing a decidable equality in Coq; we use the representation from the `SMTCoq` plugin since we will use it as a back-end (see later).

Our contribution: linking first-order Coq goals with automated provers An interesting observation about Examples 1 and 2 is that their statements and proofs pertain to first-order logic (with decidable equalities). As such, they *should* be automatized.

In this paper, we provide a methodology to reconcile Coq goals with the logic of first-order provers. This methodology consists in

1. implementing pairwise independent logical transformations: each transformation encodes one aspect of Coq logic as formulas in a less expressive logic (until reaching first-order formulas in Coq) and establishes a soundness proof of this encoding;
2. providing strategies to combine these transformations in order to automatically translate Coq goals into fully explicit first-order logic goals.

We also implement this methodology as a new Coq tactic called `snipe` (for the bird known in french as *bécassine des marais*) so that the proofs of the two lemmas presented above become only one single call to this tactic, passing the required lemma `search_app` in the second case.

The tactic `snipe` is two-fold.

1. As presented before, we implemented five small logical transformations, and a strategy that combines them. The transformations presented in this paper deal with definitions, datatypes and polymorphism, and thus the strategy transforms a Coq goal containing these features into a fully first-order goal.
2. Then, we use the SMT solver `veriT` as a back-end to discharge the obtained first-order goal, available through the `SMTCoq`³ plugin[9], which enables safe communication between Coq and SMT solvers.

In step 2, we benefit from the automation provided by the SMT solver `veriT`, which is able in particular to perform Boolean computation (as in Example 1), Linear Integer Arithmetic (LIA) or relieve the user of the burden of finding the right instantiations of the lemmas. An important observation is that in step 2, the use of `SMTCoq` could be replaced by any tactic solving first-order logic.

The remainder of the paper is dedicated to explaining the concept of `snipe` in details. In the next section, we explain our methodology. In § 3, we present examples of useful logical transformations together with their implementations. In § 4, we explain the full `snipe` tactic, as well as more examples of its power. We finally present the state of the art before concluding.

The source code can be found at <https://github.com/smtcoq/sniper/releases/tag/pxtp21>.

2 From the logic of Coq to first-order logic: modular transformations

From the Calculus of Inductive Constructions to first-order logic As we saw in the introductory examples, Coq, which is based on CIC, is mostly not automatized even for simple proofs: while it now enjoys very efficient decision procedures for dedicated theories [2, 11], attempts for general automation has not truly succeeded yet (see § 5 for a detailed comparison).

Our contribution is to propose a new approach for general automation, based on (1) small transformations of a CIC goal towards a goal of first-order logic (2) stating and proving first-order properties in the local context of a Coq proof. Then, the first-order goal and the associated first-order properties can be sent to any external automated prover based on first-order logic. As we will see in the next paragraph, the user may choose which transformation they apply.

³SMTCoq is available at <https://smtcoq.github.io>.

We choose FOL as our target logic because a lot of work has been done to automatize reasoning in this logic. So, after calling our transformations, the user can choose any way to automatically prove a first-order goal: *e.g.*, an automatic prover certified in Coq, a tool calling external solvers, tactics like `firstorder` or `crush`[6]. In § 4, we provide a fully-automated tactic that applies the transformations presented in this paper and solves the resulting first-order goal using the SMTCoq plugin.

Modular and independent transformations Formally, a logical transformation from the language of Coq to itself is a function f from the terms and formulas of Coq to themselves. Leaving aside the details on the nature of the function f for now, we are interested in **sound transformations**, *i.e.*, transformations f such that, given any Coq statement G in the domain of f , we have $f(G) \Rightarrow G$. This means that it is enough to prove $f(G)$ for G to be valid. More precisely, we are interested with sound transformations f such that $f(G)$ is a first-order formula: indeed, if a Coq goal G is left to prove, we may transform G into $f(G)$ and then send $f(G)$ to an automated theorem prover dealing with first-order. If this succeeds, it means that G is valid.

We are actually also interested in functions g from a subset of Coq formulas such that, given a Coq statement G , $g(G)$ outputs (a list of) *valid* first-order logic statements in Coq that may help proving G . We also call such a function g (which produce auxiliary first-order statements) a sound transformation.

Our approach is to develop modular and independent transformations: each of them encodes one important aspect of CIC. The aim is to tackle only one aspect at a time: it facilitates the proof of soundness whenever we need to write one in Coq. Indeed, as the transformations are simple, they preserve as much as possible the structure of the source formula. We expect that they are easier to implement than a bigger encoding. In addition, this methodology facilitates the debugging and allows us to know precisely which fragment of Coq we can handle. The transformations can be composed in different ways: we can use them separately, combine them all, or only some of them, either by using a default tactic provided for a user who does not want to think about which method they should choose, or by writing them one by one. Moreover, they are independent from the technology used for first-order proving in the end. This is illustrated by the left part of Figure 2.

Certifying and certified transformations There are two ways for writing logical transformations f from Coq to itself. As we will see, both need to resort to the **meta-language** of Coq at some point: they feature functions which cannot be defined in the core language of Coq, but only in extra-layers outside its kernel. However, they do not work the same way from the logical point of view.

1. **Certified transformations.** The function f may be a Coq function. This relies on an internal representation of Coq terms inside Coq, that we call `term` (see Example 3). It comes with two meta-language transformations: the **reification** takes a Coq term as a parameter and outputs its reification (in the type `term`) and the **dereification** is the converse. In this situation, f is a Coq function (*not* a meta-function) of type `term → term` and we may prove that, for all t of type `term`, $f\ t$ implies t (up to some implicit reification). We call this kind of transformation a *certified* one, because the soundness is proved *once and for all, in a Coq statement*.
2. **Certifying transformations.** The function f may be a meta-language function, *i.e.*, f is not a Coq function. In that case, it is *not possible* to write a Coq statement specifying that, for instance, for all G of type `set`, $f(G)$ implies G . However, we may write another meta-language function g such that, for all G , $g(G)$ generates on the fly a proof of $f(G) \rightarrow G$. Such a transformation is said to be *certifying*, because its soundness is not previously established. It takes a local context and

a *specific* goal, and it operates the transformation, which will be type-checked at the end by the kernel (when Qed is written) *every time we use it*.

The differences between certified and certifying transformations are summarized in Figure 1. In the work presented in this article, we follow the paradigm of certifying transformations. Indeed, the former require we work only with the reified syntax of the terms of CIC and even for a simple transformation, the proof of soundness is hard (thousands lines of code). But in some situations, it could be useful to use this solution, because it covers all cases.

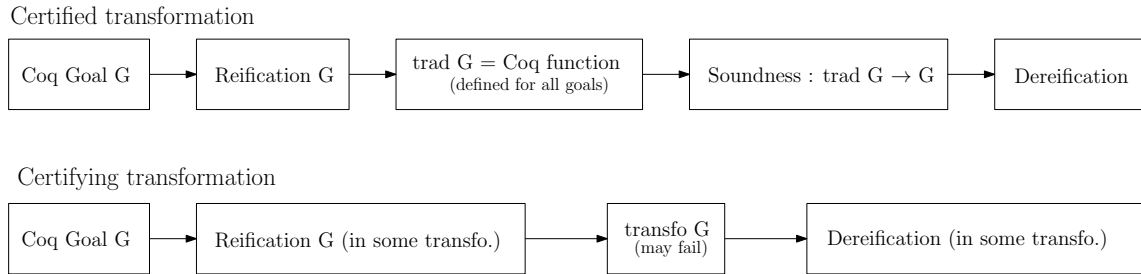


Figure 1: Difference between certified and certifying transformations

Metalinguage tools Most logical transformations analyze the syntax of terms in the source language and produce new statements. But in Coq, we do not directly have access to the syntax of terms. This is the reason why we need meta-programming tools as we saw in the previous paragraph. Many different tools are available now (see § 5), and we used two of them: Ltac [8] and MetaCoq [17], which offer different advantages.

- Ltac is a tactic language in Coq which is based on pattern matching, recursion and backtracking. It allows its users to match on the syntax of Coq terms and also on the local context, for building new goals and terms. It is also possible to execute basic tactics which operate on kernel terms, called tacticals, in a Ltac function. Ltac cannot be seen as a proper meta-programming language as there is no data type which contains the reified syntax of Coq terms. In our transformations, we use Ltac whenever we need to use a particular function on every hypotheses of the context. Once this function is applied, its result is certified in Ltac by a combination of simple Coq tacticals. Sometimes, it is also useful when a superficial access to the syntax of a Coq term is required, but Ltac is less convenient for analyzing the syntax completely.
- MetaCoq enables a more fine-grained analysis on Coq terms. Indeed, this plugin includes an inductive type `term` which corresponds exactly to the Coq counterpart of OCaml kernel terms, and comes with the reification and dereification transformations described above. As it offers an easy access to the syntax of Coq terms, our transformations often use MetaCoq. The problem with MetaCoq terms is that the reified syntax is not very readable: the variables are represented by de Bruijn indexes, and there is no notation to reduce the size of the terms. This is the reason why we prefer Ltac to get information about the initial goal, and MetaCoq when we need to build a new term. Such a flexible approach is possible, since we build certifying (not certified) transformations.

Example 3 (Reification). The MetaCoq representation of the Coq term `forall (A:Type), A → A` is `tProd (name "A")(tSort type_reif)(tProd unnamed (tRel 0)(tRel 1))`. The constructor `tProd`

corresponds to the Coq `forall` dependent product binder (and the \rightarrow is simply a notation for a non-dependent product). The type of the variable in this product is `type_reif` (not going into details, this is the MetaCoq reification of `Type`). `tRe1 0` and `tRe1 1` are the variables, represented by their De Bruijn indexes.

3 Examples of transformations

We describe now some of the transformations we have implemented. Most of them are motivated by the following facts.

1. When Coq and an external automated theorem prover communicate, **a lot of symbols defined in Coq will not be interpreted**. For instance, the function `hd_error` used in Example 1 may be uninterpreted in the external prover, which does not know anything about it except its type. In general, pattern-matching may be uninterpreted. Moreover, algebraic data types and their constructors may also be left uninterpreted. For instance, an external prover may not know that lists have basic properties, *e.g.*, `x1 :: l1 = x2 :: l2` implies `x1 = x2` and `l1 = l2`, or that `[] <> x :: l`, whereas these two *first-order statements* just come from the definition of the type `list`.
2. **Coq handles higher-order objects (in particular, equalities about such objects), whereas first-order provers do not**. For instance, `fun x =>(x + 1)** 2 = fun x =>x ** 2 + 2 * x + 1` is an equality between functions (of type `Z → Z`) and cannot be directly interpreted in first-order logic. Yet, it may be sufficient to consider its first-order consequence `forall x, (x + 1)** 2 = x ** 2 + 2 * x + 1`, which is a quantified equality on the type `Z`.

Such transformations, while simple at first glance, are already mandatory to give a bridge between a Coq goal and a first-order prover: the first kind of transformation makes the global context explicit, whereas the second kind encodes higher-order aspects.

As explained in the previous section, we implemented these encodings as certifying transformations (cf. Figure 1), using meta-programming. Transformations 3.1 and 3.5 do not need reification while others do. The approach of each transformation is the following: by scanning the goal, it states and proves *in Coq* various auxiliary lemmas of first-order logic about the terms encountered in the goal. The proofs are done easily by applying Coq basic tactics. These lemmas are stored in the local context. It may also perform some transformations on the goal, so that it becomes first-order.

3.1 Definitions

The first transformation⁴ makes user-defined terms explicit. For instance, on Example 1, it will add the following assertion to the local context:

```
hd_error_def : hd_error =
  (fun (A : Type) (l : list A) =>
    match l with | [] => None | x :: _ => Some x end)
```

This assertion is not first-order yet (it will be transformed again by the next two transformations) but it allows one to have access to the definition of the constant.

For its implementation, it mainly uses the Coq tactic `unfold`, which takes an identifier (the name of a previously defined term) as a parameter and replaces it by its definition. This tactic, `get_def`, takes the

⁴See file `definitions.v`.

name of a constant and adds its definition as a proven hypothesis. The example right above is obtained by applying this tactic on the identifier `hd_error`. While `unfold` may cause the goal or the hypotheses to become verbose and not particularly understandable for the user, `get_def` will keep the definitions in separated statements. In the tactic which combines all of our transformations, we apply `get_def` recursively to all the definitions that occur in the hypotheses and in the goal.

3.2 Expansion

Note that in the example for the tactic `get_def`, the added hypothesis pertains to a higher-order function (because of `fun (A : Type) ...`). A good way to make the equality deal with first-order objects is to apply the functional definition to an arbitrary argument of its domain⁵. That is, instead of writing $f = \lambda x. t(x)$, we write: $\forall x, f(x) = t(x)$ where x is a fresh variable, and t the unfolded definition of f . We did not use a transformation which encodes partial application with an applicative symbol because it may lead to bigger terms and it is not necessary here. We have left it for future work and more complex cases. The tactic takes a hypothesis H of the form $t = u$ where $t : A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$, and asserts and proves automatically the hypothesis

$$\forall x_1 : A_1, \dots, \forall x_n : A_n, t(x_1, \dots, x_n) = u(x_1, \dots, x_n).$$

If we come back to our running example, once we get the axiom `hd_error_def` in our context, we can apply `expand hd_error_def` to obtain a new assertion in the local context:

```
H0 : forall (A : Type) (l : list A), hd_error l =
  match l with | [] => None | x :: _ => Some x end
```

The proof of this statement is really simple in Coq. It suffices to use the tactic `rewrite H`, followed by `reflexivity`, which checks that both members of the equality are convertible in CIC. But the construction of the statement is harder. First, the tactic takes a hypothesis H (`hd_error_def` in our case) and finds, thanks to the Ltac value-function `type_of`, its type T , which must be an equality $t = u$. In order to have an easy access to the domain and codomain of t and u , we reify T . That is, we call a tactic `quote_term` from the MetaCoq plugin, which takes a Coq term and returns its reified syntax. Then we can perform syntactic operations on this term by implementing auxiliary functions directly in Coq. Our tactic is thus divided in four parts:

- The main tactic `expand` reifies the hypothesis H and calls the auxiliary functions.
- The first auxiliary function `list_of_args_and_codomain` finds the common type of t and u : $A_0 \rightarrow \dots \rightarrow A_n \rightarrow B$ and returns the pair $([A_0; \dots; A_n], B)$. All the terms involved here are MetaCoq terms.
- The second auxiliary function `gen_eq` constructs the reified equality that we want. It is defined recursively on the list $[A_0; \dots; A_n]$ by the following equations.

$$\text{gen_eq}([], B, t, u) \triangleq t =_B u$$

$$\text{gen_eq}([A_0; \dots; A_n], B, t, u) \triangleq \forall x_0 : A_0, \text{gen_eq}([A_1; \dots; A_n], B, t\ x_0, u\ x_0)$$
 The `gen_eq` function deals with De Bruijn indices by lifting these at every recursive call.
- The generated equality is then unquoted and proved by using Coq tactics.

3.3 Elimination of fixpoints

For the sake of simplicity, we have so far taken the example of a non-recursive function (`hd_error`). However, it is very often the case that functions are defined recursively on datatypes. In this section,

⁵See file `expand.v`.

we take the example of the recursive function `length` computing the number of elements in a list. If we expand its definition as presented in the previous sections, we get:

```
H : forall (A : Type) (l : list A), length l =
  (fix length_anon (l : list A) : nat :=
    match l with | [] => 0 | _ :: l' => S (length_anon l') end) l
```

As `length` is a fixpoint, its definition is hidden in an anonymous function denoted by `fix`. Again, we transform this expression to make it more intelligible by automatic provers. This is the role of the tactic `eliminate_fix`⁶ that replaces the anonymous function with the constant it defines. Here, `eliminate_fix H` asserts and proves the new hypothesis `H0`:

```
H0 : forall (A : Type) (l : list A), length l =
  match l with
  | [] => 0
  | _ :: l' => S (length l')
  end
```

Thanks to this tactic, we now have access to the body of the definition of `length` as for non-recursive functions.

3.4 Elimination of pattern matching

Definitions by pattern matching, such as the examples of the previous two subsections, are not understandable for automated provers. More generally, they are not part of the syntax of first-order terms. Thus, instead of getting a function defined by pattern matching, we would like to have one statement for each pattern. We implemented a tactic⁷ which does precisely this. In order to present it, let us continue our example from 3.2. The tactic `eliminate_pattern_matching H0` (where `H0` is the hypothesis generated in 3.2), produces two hypotheses:

```
H1 : forall (A : Type), hd_error [] = None
H2 : forall (A : Type) (x : A) (l : list A), hd_error (x::l) = Some x
```

The tactic works in five steps. Note that the formula to which it is applied must be of the form `forall (x0: A0) ... (xi: Ai), E[match xi with ...]` where `E` is an environnement with (possibly) free variables.

- The index i of the matched variable is computed with a combination of a dummy subgoal and a metavariable allowing to pass on the result to the main goal.
- The formula is reified and the reified types A_0, \dots, A_i are retrieved.
- An independent tactic scans the global environment in which the inductive definition of A_i can be found. It returns the list of its reified constructors $[C_0; \dots; C_j]$ and their reified types $[T_{0,0} \rightarrow \dots \rightarrow T_{0,n_0} \rightarrow A_i; \dots; T_{j,0} \rightarrow \dots \rightarrow T_{j,n_j} \rightarrow A_i]$.
- For each constructor C_k , we construct the following statement:

```
forall (x0: A0) ... (xi-1: Ai-1),
  forall (ak,0: Tk,0) ... (ak,nk: Tk,nk),
  E(match Ck ak,0 ... ak,nk with ...)
```

- Each statement is unquoted, asserted and proved by `intros`; `rewrite H0`; `reflexivity`.

⁶See file `elimination_fixpoints.v`

⁷See file `elimination_pattern_matching.v`.

3.5 Monomorphization

Most automatic provers do not support polymorphism. In other words, they cannot prove lemmas about functions that can be defined on any type of data. Typically, as we saw in Example 2, the lemma `search_app` is polymorphic, and thus cannot be sent directly to most provers. However, only its instance on `Z` is useful for proving `search_lemma`. This transformation will add the following statement to the local context:

```
search_app_Z : forall (x: Z) (l1 l2: list Z), search x (l1 ++ l2) =
  (search x l1) || (search x l2)
```

There are various ways to handle polymorphism[3][4]. Among them we chose a monomorphization based on instantiating the polymorphic types with chosen ground types from the context.

To write the monomorphization tactic⁸, the meta-programming language Ltac was our main tool. Indeed, we did not need a detailed access to the syntax of the terms as MetaCoq provides, but we wanted to apply the same tactic to all the hypotheses in a given context (or to a list of polymorphic lemmas). Ltac allows matching on the local context in a rather simple way, thanks to the tactic `match goal`. In details, the instantiation tactic tries to match all hypotheses in the local context whose type P is a quantified hypothesis: $\forall A : \text{Type}, P'$. MetaCoq is useful here, to check that A has type Type .

Then, the monomorphization tactic scans the goal and instantiates the variable of type Type with all the subterms of type Type in the goal. All the generated hypotheses are automatically proven by the tactic `specialize`.

In order to avoid infinite loops, the instantiated hypothesis is not added in the context if it is already present. The tactic can also take parameters (polymorphic lemmas), and they are monomorphized in the same way.

In the future, we will run benchmarks to measure the performance of our tactic. This may help us to develop heuristics for choosing the instances of lemmas efficiently.

3.6 Interpreting Algebraic types

Algebraic datatypes are a special case of inductive types which do not use non-prenex polymorphism or type dependencies. The epitomy of such a type is perhaps `list` that we used in our examples:

```
Inductive list (A : Type) : Type :=
| [] : list A
| cons : A → list A → list A.
```

where the constructor `cons` has the infix notation `::`. When one is familiar with inductive types, one knows that this declaration specifies how equality works on the type `list`. For instance, `x1 :: l1 = x2 :: l2` implies `x1 = x2` and `l1 = l2`. Moreover, `[] <> x :: l1`. In general, in an algebraic datatype I as defined in Coq:

- Each constructor C of I is injective, that is:

```
forall (x1 y1: A1) ... (xn yn: An),
  C x1 ... xn = C y1 ... yn → x1 = y1 ∧ ... ∧ xn = yn
```

- If C and C' are two distinct constructors of I , then their direct images are disjoint, that is:

⁸See file `elimination_polymorphism.v`.


```
forall (x1 : A1) ... (xn : An) (x1' : A1') ... (xp' : Ap'),
  C x1 ... xn <> C' x1' ... xp'
```

Any inhabitant of I is obtained from one of the C_i of I , that is:

```
forall (x : I),
  ((exists x1,l : A1,l) ... (exists xkl,l : Akl,l), x = C1 x1,l ... xkl,l )
  ∨ ... ∨ ((exists xn,l : An,l) ... (exists xkn,kn : Akn,kn), x = Cn xn,l ... xkn,kn )
```

Note that the above propositions are statements of first-order logic and as such, may be communicated to a first-order automatic theorem prover. However, the third property is written with existential quantifiers which are not treated by the back-end we use in our proof of concept (see Sec 4), so our tactic does not generate this property for the moment.

We define the tactic `interp_alg_types`⁹ that finds the algebraic datatypes of Coq (e.g., `list Z`) which occur in the goal and automatically proves that (1) their constructors are injective (2) the direct images of their constructors are disjoint.

If we come back to Example 1, the datatypes `list` and `option` are both made explicit so that the automatic prover is able to conclude.

4 Proof of concept

As explained in § 2 and the left part of Figure 2, the methodology is to combine such transformations, possibly in different ways, then call an automatic solver.

In this section, we provide a proof of concept: all the transformations in the previous section are combined in a fully automatized tactic called `snipe` which applies them and sends the resulting goal and context to the SMT solver `veriT` through the `SMTCoq` plugin. This is illustrated by the right part of Figure 2. We now detail this combination and come back to examples illustrating the `snipe` tactic.

4.1 Proof strategy

As presented in Figure 2 we proceed as follows. We first apply a combination of our transformations in a unique tactic called `scope`. Thanks to the `SMTCoq` plugin, we send the goal and the local context with additional lemmas obtained by the `scope` tactic to the external solver `veriT`. Let us describe this two-part process more precisely:

- `scope`: This tactic consists of applying first `interp_alg_types` to all algebraic types in the goal and in the context, except types already interpreted by the SMT solver like \mathbb{Z} or the Booleans. Then it calls the `get_definitions` tactic: it adds new definitional hypotheses in the local context, except for the symbols which are part of the built-in theories of `veriT`. For instance, the definition of the addition in \mathbb{Z} is not needed. Then, the tactics `expand`, `eliminate_fix` and `elimination_pattern_matching` are applied to the new generated hypotheses. Finally, the monomorphization tactic will assert and prove a new proposition for every polymorphic hypothesis applied to a subterm of type `Type` in the goal. A tuple of Coq lemmas chosen by the user can be added as parameters to `snipe`: the tactic will also try to instantiate them if they are polymorphic.

⁹See file `interpretation_algebraic_types.v`.

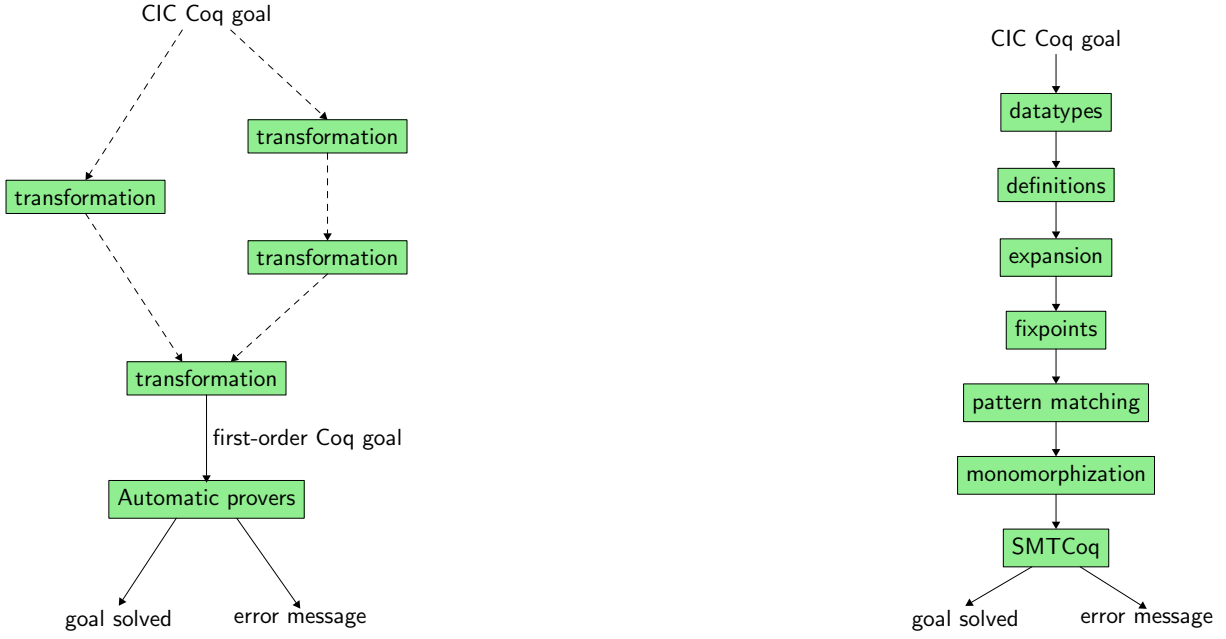


Figure 2: General methodology (left) and proof of concept (right)

- Once scope is applied, the tactic `veriT` is called. It is an `SMTCoq` tactic which solves first-order goals in a combination of built-in theories (such as linear arithmetic or congruence) by calling the external SMT solver `veriT` and reconstructing a Coq proof. Note that:
 - this tactic requires equalities to be decidable (this is the proposition `CompDec A` mentioned in the introduction)
 - its implementation relies on the Coq standard library for machine integers and arrays, which axiomatize these data-structures; that is why the tactic `snipe` also relies on these axioms.

4.2 Examples

Let us go back to our examples¹⁰. Example 1 is automatically solved with `snipe`. Note however that we need an additional hypothesis `CompDec A`: as previously mentioned the type `A` has to have a decidable equality for `SMTCoq` to reason on it. The `scope` part of the tactic adds all the hypotheses about the constructors of types `option` and `list`, and they are instantiated by the variable `A`. The definition of `hd_error` is also added in the local context. Since it contains a pattern matching, a proposition for every pattern is created and proved. The tactic `veriT` transforms the first-order hypotheses into assertions for `veriT` and solves the goal.

Example 2, about `search`, requires the instantiation of the previous lemma `search_app`. Again, it is solved automatically by the tactic `snipe`, taking this time the lemma `search_app` as a parameter. The proof becomes:

```
Goal forall (A : Type) (H : CompDec A) (x: A) (l1 l2 l3: list A),
  search x (l1 ++ l2 ++ l3) = search x (l3 ++ l2 ++ l1).
Proof. intros A H. snipe @search_app. Qed.
```

¹⁰See file `examples.v`.

The instantiation of the polymorphic lemma is done by the monomorphization tactic within `scope`. The part of the proof which required a lemma about Booleans and an adequate instantiation of it is automatically handled by the capabilities of `veriT` to perform propositional reasoning with quantifier instantiation.

More interestingly, we can also prove the intermediate lemma in a very satisfactory way. This lemma is proved by induction on the first list, and induction is currently out of the scope of most automated provers. However, once this induction is done, the user should not worry about proving the sub-cases. This is made possible by our tactic:

```
Lemma search_app : forall {A:Type} {H:CompDec A} x (l1 l2:list A),
  search x (l1 ++ l2) = (search x l1) || (search x l2).
Proof. intros A H x l1 l2. induction l1 as [ | x0 l0 IH]; simpl;
  snipe. Qed.
```

5 State of the art

General automation in proof assistants Improving automation in proof assistants based on Type Theory is a long-standing research topic.

Decision procedures for dedicated theories have met a large success, witnesses the daily used `ring` [11] and `lia` [2] tactics in Coq, deciding equalities respectively in (semi-)ring structures and propositions in the linear integer arithmetic theory. Automatic tactics also exist for propositional and first-order logic, such as `intuition`, `firstorder` or `crush`[6] in Coq. The limitations of these tactics are that they are useless as soon as the reasoning requires handling multiple aspects at one time (such as propositional logic, arithmetic, equalities...), which is very often the case when using an interactive theorem prover.

This is why research in this area moved towards making use of more complex automatic solvers, mainly SMT solvers, which can combine theories with tableau or first-order provers that usually do not natively handle theories but better deal with quantifiers. In this direction, two approaches are usually considered: the *autarkic* approach, which consists in implementing and proving correct the automatic prover in the proof assistant (*e.g.*, the SMT solver `ergo` [13] in Coq, the tableau prover `blast` [14] in Isabelle/HOL, or the first-order solver `metis` in the HOL family [12]), and the *skeptical* approach, which consists in using external provers that output explanations and only checking these explanations at each execution (*e.g.*, `SMTCoq` [9] in Coq or `smt` [5] in Isabelle/HOL).

However, as we explained, goals in proof assistant usually do not belong to first-order logic, which is the logic handled by these provers. This is why encodings need to be performed. In this direction, the most successful tool is `sledgehammer` [15] for Isabelle/HOL, which mixes both an autarkic and skeptical approaches: it encodes a higher-order goal, calls many external solvers in parallel (using lemmas from the global context selected by machine learning), and uses their answers to reconstruct a proof of the original goal, by mixing standard tactics with `smt` or `metis`. This approach was ported to Coq in the `CoqHammer` [7] tool, but with less success.

We identified one main limitation of `CoqHammer` to be proof reconstruction: the tool tries to build a proof of the original goal, but the external solvers proved the encoded goal. This encoded goal is far from the original goal because `CoqHammer` uses a one pass, very complex encoding of CIC into FOL. This was not a problem for Isabelle/HOL whose logic is simpler. One cannot rely on the reconstruction of a proof of the encoded goal, because then it should be proved correct with respect to the original goal, which is very difficult again because of the complexity of the encoding.

This is why we proposed this new approach, where the encoding is a combination of small transformations that can either be certified or output Coq proofs (certifying). Then the resulting goal can

be checked by any approach for first-order proving since we do not need to reconstruct a proof of the original goal. This approach with small, independent transformations was inspired by other tools that use external automatic solvers, in particular Why3 [10].

As we explained, we are independent of the back-end used to discharge the first-order goal produced by the transformations. We chose SMTCoq in our proof of concept. It restricts us to hypotheses and goals with only universal and prenex quantification, but offers built-in theories, which seems a good trade-off for the kind of goals that are commonly present in Coq. Targeting first-order provers could also be done by providing other strategies that would do less work on quantifiers but encode theories such as linear arithmetic.

We leave for future work a detailed comparison with CoqHammer, both in terms of performance and expressivity. For this latter, we are currently theoretically less expressive than CoqHammer (since we only handle a small part of Coq logic beyond FOL), but

- we can detail the fragment of Coq logic that we handle, whereas the success of CoqHammer is more unpredictable,
- we believe that the approach will scale when implementing more involved transformations.

Recently, automatic provers have pushed towards more expressivity than FOL [1]. Once they can be used in proof assistants such as Coq (currently they do not output certificates for higher-order aspects), it will be very easy to integrate them in our setting: it simply requires unplugging the transformations that deal with the aspects newly handled by these provers. We could also consider using the theory of algebraic datatypes supported by some SMT solvers to handle a sub-part of Coq inductive types.

Meta-programming in Coq The approach by certified/certifying transformations relies on meta-programming. As explained in § 2, we use two meta-programming tools available in Coq: Ltac and MetaCoq. These tools are complementary: Ltac allows us to handle surface meta-programming very easily, whereas MetaCoq allows us to go deeper into the structure of terms, at the cost of difficult aspects such as De Bruijn indices. We plan for future work to look at other tools that could enjoy both worlds, in particular Ltac2 [16] and Coq-Elpi [18].

6 Conclusion and perspectives

We have developed a methodology to encode some aspects of CIC into FOL by writing independent and modular transformations. This modularity is useful to delimit the features of CIC we can translate, and to combine the transformations in a chosen order. As a proof of concept, we created a Coq tactic *snipe* which performs the transformations described above and calls an external SMT solver. Some Coq proofs are now totally automatized thanks to our tactic.

Now that we have the crucial and basic transformations to extend the local context with first-order hypotheses, we plan to tackle more complex transformations. This future work will help to automatize other aspects of Coq logic. Here is a non exhaustive list of transformations or features we would like to treat:

- **Encoding of higher-order terms.** We may treat them as usual first-order terms and add an applicative symbol $@$ to the language signature. Thus, $f(x)$ becomes $@(f,x)$.
- **Encoding of inductive predicates.** They are a particular case of dependent typing, and we would like to obtain first-order statements from them as they are very common whenever program specifications are written in Coq.

- **Skolemization.** As we may use external solvers which do not deal with existential quantifiers after applying our tactic `scope`, we would like to be able to encode them and thus perform a skolemization on the goal and the hypotheses.

Another important part of our future work is to improve performance and do a benchmark analysis. This benchmark has to be *qualitative* and *quantitative*, that is, we need to evaluate the efficiency of our tactic and to know how many goals it can solve. In particular, we will compare `snipe` with `CoqHammer`. As previously said, we hope that a more elaborated version of our tactic will do better than `CoqHammer` in the fragment of CIC we chose to deal with.

References

- [1] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli & Clark W. Barrett (2019): *Extending SMT Solvers to Higher-Order Logic*. In Pascal Fontaine, editor: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings, Lecture Notes in Computer Science* 11716, Springer, pp. 35–54, doi:10.1007/978-3-030-29436-6_3.
- [2] Frédéric Besson (2006): *Fast Reflexive Arithmetic Tactics the Linear Case and Beyond*. In Thorsten Altenkirch & Conor McBride, editors: *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers, Lecture Notes in Computer Science* 4502, Springer, pp. 48–62, doi:10.1007/978-3-540-74464-1_4.
- [3] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu & Nicholas Smallbone (2016): *Encoding Monomorphic and Polymorphic Types*. *Log. Methods Comput. Sci.* 12(4), doi:10.2168/LMCS-12(4:13)2016.
- [4] François Bobot & Andrey Paskevich (2011): *Expressing Polymorphic Types in a Many-Sorted Language*. In Cesare Tinelli & Viorica Sofronie-Stokkermans, editors: *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings, Lecture Notes in Computer Science* 6989, Springer, pp. 87–102, doi:10.1007/978-3-642-24364-6_7.
- [5] Sascha Böhme & Tjark Weber (2010): *Fast LCF-Style Proof Reconstruction for Z3*. In Matt Kaufmann & Lawrence C. Paulson, editors: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings, Lecture Notes in Computer Science* 6172, Springer, pp. 179–194, doi:10.1007/978-3-642-14052-5_14.
- [6] Adam Chlipala (2013): *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. Available at <http://mitpress.mit.edu/books/certified-programming-dependent-types>.
- [7] Lukasz Czajka & Cezary Kaliszyk (2018): *Hammer for Coq: Automation for Dependent Type Theory*. *J. Autom. Reason.* 61(1-4), pp. 423–453, doi:10.1007/s10817-018-9458-4.
- [8] David Delahaye (2000): *A Tactic Language for the System Coq*. In Michel Parigot & Andrei Voronkov, editors: *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings, Lecture Notes in Computer Science* 1955, Springer, pp. 85–95, doi:10.1007/3-540-44404-1_7.
- [9] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds & Clark W. Barrett (2017): *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*. In Rupak Majumdar & Viktor Kuncak, editors: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II, Lecture Notes in Computer Science* 10427, Springer, pp. 126–133, doi:10.1007/978-3-319-63390-9_7.
- [10] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 - Where Programs Meet Provers*. In Matthias Felleisen & Philippa Gardner, editors: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of*

- Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. *Proceedings, Lecture Notes in Computer Science* 7792, Springer, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.
- [11] Benjamin Grégoire & Assia Mahboubi (2005): *Proving Equalities in a Commutative Ring Done Right in Coq*. In Joe Hurd & Thomas F. Melham, editors: *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings, Lecture Notes in Computer Science* 3603, Springer, pp. 98–113, doi:10.1007/11541868_7.
 - [12] J. Hurd (2005): *System Description: The Metis Proof Tactic*. *Empirically Successful Automated Reasoning in Higher-Order Logic (ESHOL)*, pp. 103–104.
 - [13] Stéphane Lescuyer (2011): *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et développement d’une tactique reflexive pour la demonstration automatique en coq)*. Ph.D. thesis, University of Paris-Sud, Orsay, France. Available at <https://tel.archives-ouvertes.fr/tel-00713668>.
 - [14] Lawrence C. Paulson (1999): *A Generic Tableau Prover and its Integration with Isabelle*. *J. Univers. Comput. Sci.* 5(3), pp. 73–87, doi:10.3217/jucs-005-03-0073.
 - [15] Lawrence C. Paulson & Jasmin Christian Blanchette (2010): *Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers*. In Geoff Sutcliffe, Stephan Schulz & Eugenia Ternovska, editors: *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011, EPIc Series in Computing* 2, EasyChair, pp. 1–11. Available at <https://easychair.org/publications/paper/wV>.
 - [16] Pierre-Marie Pédro (2019): *Ltac2: tactical warfare*. *CoqPL 2019*.
 - [17] Matthieu Sozeau, Abhishek Anand, Simon Boulrier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau & Théo Winterhalter (2020): *The MetaCoq Project*. *J. Autom. Reason.* 64(5), pp. 947–999, doi:10.1007/s10817-019-09540-0.
 - [18] Enrico Tassi (2019): *Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq*. In John Harrison, John O’Leary & Andrew Tolmach, editors: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA, LIPIcs* 141, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 29:1–29:18, doi:10.4230/LIPIcs.ITP.2019.29.

Integrating an Automated Prover for Projective Geometry as a New Tactic in the Coq Proof Assistant

Nicolas Magaud

ICube UMR 7357 CNRS - Université de Strasbourg

magaud@unistra.fr

Recently, we developed an automated theorem prover for projective incidence geometry. This prover, based on a combinatorial approach using matroids, proceeds by saturation using the matroid rules. It is designed as an independent tool, implemented in C, which takes a geometric configuration as input and produces as output some Coq proof scripts: the statement of the expected theorem, a proof script proving the theorem and possibly some auxiliary lemmas. In this document, we show how to embed such an external tool as a plugin in Coq so that it can be used as a simple tactic.

1 Introduction

The Coq proof assistant [7, 3] is a generic theorem prover, with huge capabilities. One of its main strengths is that it allows carrying out proofs either interactively or under some assumptions automatically. Its standard tactics allow to solve goals in some well-understood fragments of its underlying logic, e.g. first order logic using the tactic `firstorder` or linear arithmetic using the tactic `lia`. When the proofs get more technical or are outside of the scope of these automatic tactics, the user can take control and thus we are not limited any more by the power of automation. Contrary to what happens with SMT provers, like Z3 [12] or Vampire [9], which either succeed or fail on the goal, in Coq, the user can perform some proof steps interactively and then rely again on automated tools to solve the goal.

However, it requires a lot of expertise to be able to develop some new tactics in Coq to help the user proving theorems more easily. Most tactics are written using the Ltac tactic language [8], but writing tactics sometimes also requires to implement some features in OCaml in a Coq plugin.

In the context of geometry, we developed an independent theorem prover, implemented in C, which handles first-order statements. It takes as input a geometric configuration, written as plain text, and returns a Coq proof script, ready to be verified (type-checked) by Coq. We choose to proceed that way in order to make the development as easy as possible for a non-expert in Coq internals. The only requirements are to know the specification and the tactic languages of Coq, and thus to be able to produce some Coq files (actually a simple `.v` file).

In this article, we show how to integrate such an automated prover as a tactic in Coq, without modifying the interface of the automated prover.

In Sect. 2, we briefly present the automated prover, which we shall see as a blackbox in the rest of this document. In Sect. 3, we show how to translate a Coq goal into the input language of our prover. In Sect. 4, we investigate how to use the Coq script produced by our prover to solve the initial Coq goal. In Sect. 5, we discuss our implementation choices and explain how it could be extended to other external provers.

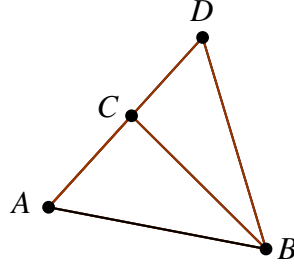


Figure 1: A simple geometric configuration illustrating the statement of Fig. 2

2 The automated prover

We work in the framework of projective incidence geometry, which is one of the simplest theory that can be used to capture most aspects of geometry. It is based on an incidence relation between points and lines. In its projective version, we assume that two coplanar lines always intersect. One of its main advantages is that it can be described using only a few axioms; thus it is a well-suited framework to try and automate proofs of theorems. This geometry can be described either in a synthetic way, writing statements using the incidence relation \in , or in a more combinatorial way, using the rank of a set of points [11]. For instance, the property that a line has at least three distinct points can be expressed in a synthetic way as follows :

$$\forall l : \text{Line}, \exists ABC : \text{Point}, A \neq B \wedge B \neq C \wedge A \neq C \wedge A \in l \wedge B \in l \wedge C \in l.$$

It can also be implemented as the following statement using the matroid based description using ranks :

$$\forall AB : \text{Point}, rk\{A, B\} = 2 \Rightarrow \exists C : \text{Point}, rk\{A, B, C\} = rk\{B, C\} = rk\{A, C\} = 2.$$

In the combinatorial approach, we exclusively deal with points and geometric reasoning is replaced by some computations relying on the combinatorial properties of the underlying matroid on which the rank properties are based [11]. Concretely, the rank of a set of two distinct points A and B is equal to 2. The rank of a set of three collinear points A, B, and C is also equal to 2. The rank of the whole plane in a two dimensional setting is 3. In a space of dimension $n \geq 3$, the maximum rank of a set of points is $n + 1$. In the case $n = 3$, a set of points whose rank is 4 is not a plane and actually captures the whole space.

The above two approaches are shown to be equivalent [5] and the combinatorial one can be successfully used to automatically prove some emblematic theorems of 3D projective incidence geometry [4]. Among them we can cite Desargues' theorem and Dandelin-Gallucci's theorem [6]. The automated prover, named Bip for *matroid Based Incidence Prover*, is designed to prove equality between ranks of various sets of points. It is based on rank interval computations. For each subset of the powerset of the geometric configuration, we define the minimum and the maximum rank (in the worst case, when no information is known, the rank of each non-empty subset is between 1 and 4). We then use the matroid axioms, which specify the rank function, and we reformulate them as rewrite rules to incrementally reduce the size of the interval for each subset. This is achieved using a saturation algorithm, which is run on a valuated graph implementing the inclusion lattice of the point powerset, labeled by the minimum and maximum rank. Once the saturation graph is built, it is traversed to build a Coq proof script which

Lemma ex2 :

```
forall A B C D:Point,
rk(A :: D :: B :: nil) = 3 ->
rk(A :: C :: D :: nil) = 2 ->
rk(C :: A :: nil) = 2 ->
rk(C :: D :: nil) = 2 ->
rk(A :: C :: B :: nil) = 3.
```

Proof.

```
context
dimension 3
layers 1
endofcontext
layer 0
points
A B C D
hypotheses
C D : 2
C A : 2
A C D : 2
A D B : 3
conclusion
A C B : 3
endoflayer
conclusion
A C B : 3
end
```

Figure 2: The Coq statement of a simple projective geometry theorem (left) and the corresponding input for the automated prover (right)

actually proves the statement at stake. Technical details about the implementation can be found in [4] and the current implementation of the prover as well as several examples of applications are available in the git repository: <https://github.com/pascalschreck/MatroidIncidenceProver>.

In the next sections, we show how to integrate such a tool so that it can be simply used as a tactic in Coq.

3 Translating a Coq statement into the input language of the prover

Provided that we define the theory of ranks in Coq (see Appendix A), as we first did in [10], we can state geometric properties as Coq lemmas using equalities on the ranks of some sets of points.

A Coq statement featuring some rank properties can easily be translated into an input file for the automated prover. It simply consists in traversing the context and the goal, harvesting all variables of type `Point` and all statements of the form $\text{rk}(e)=k$. All other statements are simply dropped, as they could not be used by the automatic prover yet. Fig. 2 shows the correspondence between the Coq statement of a simple theorem in projective geometry, depicted in Fig. 1, and its translation into the input language of our automated prover.

This translation can be easily implemented inside a Coq plugin, following the examples of the plugin tutorial available in the reference manual of Coq [7]. It produces a simple text file, which can be then used as input by the Bip prover. In the generated input of the Bip prover shown in the right part of Fig.2, layers are a legacy tool which was used to decompose the context at stake. This allows to build smaller proof scripts which can be easily checked by Coq. In the current version of the prover, each deduced statement is stated as a lemma and thus layers are not needed anymore. Once the input file is generated, we simply execute the external prover from Coq using the `Unix.system` primitive of the OCaml library. It produces a Coq file (`.v`), with the appropriate prelude, which is ready to be verified

by Coq.

An implementation of this translation as well as the mechanisms to use the produced proof inside Coq is available in this git repository: <https://github.com/magaud/projective-prover>. It works with Coq 8.12.2 (December 2020) and we are currently updating it to the most recent release of Coq.

4 Running the (generated) Coq script and proving the goal

Assuming the automated prover manages to produce a Coq proof script which proves the goal at stake, we still need to load this Coq file inside Coq and use the externally proven lemma to actually prove the initial goal. Let us check what happens with our simple example. As the automated prover reorders the points in the sets at stake (they are ordered according to the order in which they were introduced, during the construction of the input file), the automatically generated statement and the initial one slightly differ. In our case, the prover produces the following statement in a file named `pprove_ex2.v`¹:

```
Lemma LABC : forall A B C D ,
  rk(A :: C :: nil) = 2 ->
  rk(A :: B :: D :: nil) = 3 ->
  rk(C :: D :: nil) = 2 ->
  rk(A :: C :: D :: nil) = 2 ->
  rk(A :: B :: C :: nil) = 3.
```

Proof.

Proving the initial goal `ex2` from Sect.3 requires compiling all the automatically generated code leading to the proof of lemma `LABC` and loading it in Coq. Then the statement `LABC` can be almost directly applied to prove the statement at stake `ex2`. The slight differences between these two statements lie in the order of the points in each hypothesis or conclusion of the form $rk(e) = n$. Points need to be reordered according to the order in which they are introduced in the context. This is achieved by running the Ltac code `solve_using`² which transforms the initial goal until it exactly matches the automatically-proven statement.

```
Require Import pprove_ex2.
solve_using LABC.
```

At this point, even if we can be fairly confident that the whole machinery works properly, we do not have a strong warranty that everything went as planned. The last step of the proof is, as usual in Coq, to type-check the built proof using the concluding command `Qed`. Once this step is completed, we are sure that our prover actually proved the initial statement provided by the user.

5 Discussion

We choose a simple but efficient approach to run the prover inside Coq. One of the main advantages of this approach is that it does not require the designer of the external prover to have any understandings of the internals of Coq. It simply requires the developer to be knowledgeable of the way Coq scripts

¹The name of the Lemma is automatically generated, starts from the `L` of Lemma, followed by all the names of the points used in the conclusion.

²The code of `solve_using` is in the file `Ltac_utils.v`

are written. This is a skill every Coq user has. Such an approach thus makes every Coq user a potential tactic writer, provided we can build an interface linking Coq statements to the expected inputs of such an external prover. Indeed, the critical part of our tool is the translation of the statement from Coq to the input language of the prover. The proof generation does not need to be formally verified. Indeed, Coq will reject the proof script if it does not actually prove the statement at stake (either because statements do not coincide, or because the proof script is incorrect).

The connection from Coq to the automated prover is fairly straightforward. However the embedding of the automated prover into Coq is a bit more technical. It is not fully automated yet because calling a command (to load a compiled `.vo` file) from a tactic requires subtle interactions with the State Transaction Machine (which allows for faster and parallel executions of some parts of the proofs [2]) to maintain consistency of the proof document. For the time being, we make the `pprove` tactic display the command and tactic to be applied to complete solving the goal. In the near future, we plan to integrate these two extra steps directly into the code of the `pprove` tactic.

We think this simple way of embedding the prover could be generalized to other external provers. So far, our automated prover is running from scratch every time we compute the proof again. We plan to add some memoization techniques in order to avoid recomputing the saturation every time. We could aim at a more incremental tool where the saturation mechanism is carried out when a new point is added to the context.

More integrated tools to make automated proofs exist [1], designed to embed SAT and SMT solvers inside Coq and ensure that they provide correct decision procedures. Their approach is safer than ours, but we argue that simply generating a Coq file is much easier to carry out by an average programmer and can be used as a first step towards integrating an automated tool into Coq.

6 Conclusions and Perspectives

We successfully embed the external prover Bip as a simple tactic to be used in Coq using Ltac. We proceed in two steps: we first develop a prover, which is completely independent from the internals of Coq. The second step consists in connecting this external prover to Coq. We call the prover from Coq and retrieve the external proof script it generates. We then simply feed this script to Coq to type-check it. The API of Coq is not intended to make it easy to embed some Coq proof scripts generated externally to prove a goal. We believe it would be a nice feature to ease this process by allowing a tactic to simply run a snippet of Coq code loaded from a file.

In the longer run, improving our prover requires to make it able to take synthetic geometry statements as input rather than sets of points and ranks. We carry on developing a way to automatically translate synthetic geometry statements into ranks equalities in Coq, so that the user deals with synthetic geometry, whereas the prover deals with sets of points and their ranks in the back-end. A draft implementation of this translation is available in the file `translate.v` of the git repository.

Finally, we think our example describes an efficient and straight-to-the-point approach to embed an automated prover inside Coq and is useful as a first step towards a more integrated embedding.

References

- [1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In Jean-Pierre Jouanaud & Zhong Shao, editors: *Certified Programs and Proofs - First International Conference, CPP 2011*,

- Kenting, Taiwan, December 7-9, 2011. Proceedings, Lecture Notes in Computer Science 7086*, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.
- [2] Bruno Barras, Carst Tankink & Enrico Tassi (2015): *Asynchronous Processing of Coq Documents: From the Kernel up to the User Interface*. In Christian Urban & Xingyuan Zhang, editors: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings, Lecture Notes in Computer Science 9236*, Springer, pp. 51–66, doi:10.1007/978-3-319-22102-1_4.
 - [3] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, An EATCS Series, Springer-Verlag, Berlin/Heidelberg, doi:10.1007/978-3-662-07964-5. 469 pages.
 - [4] David Braun (2019): *Approche combinatoire pour l'automatisation en Coq des preuves formelles en géométrie d'incidence projective*. Ph.D. thesis, Université de Strasbourg. Available at <https://tel.archives-ouvertes.fr/tel-02512327>.
 - [5] David Braun, Nicolas Magaud & Pascal Schreck (2019): *Two Cryptomorphic Formalizations of Projective Incidence Geometry*. *Annals of Mathematics and Artificial Intelligence* 85(2–4), pp. 193–212, doi:10.1007/s10472-018-9604-z. Available at <https://hal.inria.fr/hal-01887000>.
 - [6] David Braun, Nicolas Magaud & Pascal Schreck (2021): *Two New Ways to Formally Prove Dandelin-Gallucci's Theorem*. In: *Proceedings of the 2021 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2021, Saint Petersburg (Virtual), Russia, July 18-23, 2021*, ACM, pp. –, doi:10.1145/3452143.3465550.
 - [7] Coq development team (2021): *The Coq Proof Assistant Reference Manual, Version 8.13.2*. INRIA. Available at <http://coq.inria.fr>.
 - [8] David Delahaye (2000): *A Tactic Language for the System Coq*. In Michel Parigot & Andrei Voronkov, editors: *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings, Lecture Notes in Computer Science 1955*, Springer, pp. 85–95, doi:10.1007/3-540-44404-1_7.
 - [9] Laura Kovács & Andrei Voronkov (2013): *First-Order Theorem Proving and Vampire*. In Natasha Sharygina & Helmut Veith, editors: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, Lecture Notes in Computer Science 8044*, Springer, pp. 1–35, doi:10.1007/978-3-642-39799-8_1.
 - [10] Nicolas Magaud, Julien Narboux & Pascal Schreck (2012): *A Case Study in Formalizing Projective Geometry in Coq: Desargues Theorem*. *Computational Geometry: Theory and Applications* 45(8), pp. 406–424, doi:10.1016/j.comgeo.2010.06.004. Available at <http://hal.inria.fr/inria-00432810/en>.
 - [11] Dominique Michelucci & Pascal Schreck (2006): *Incidence Constraints: a Combinatorial Approach*. *Int. Journal of Computational Geometry and Applications* 16(5-6), pp. 443–460, doi:10.1142/S0218195906002130.
 - [12] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Proceedings of TACAS 2008, LNCS 4963*, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

A A few lines of code specifying the rank function and its properties

We outline the context describing the rank function which is loaded before starting the proof in Coq. The axioms describe the properties of the rank function and are used in the automatically built Coq proof script. See files `basic_matroid_list.v` and `basic_rank_space_list.v` for details.

```
Parameter Point : Set.
```

```
Parameter eq_dec : forall A B : Point, {A = B} + {~ A = B}.
```

```

Definition equivlist (l l':list Point) :=
  forall x, List.In x l <-> List.In x l'.

Parameter rk : list Point -> nat.
Parameter rk_compat :
  forall x x', equivlist x x' -> rk x = rk x'.

Global Instance rk_morph : Proper (equivlist ==> (@Logic.eq nat)) rk.

Parameter matroid1_a : forall X, rk X >= 0.
Parameter matroid1_b : forall X, rk X <= length X.
Parameter matroid2 :
  forall X Y, incl X Y -> rk X <= rk Y.
Parameter matroid3 :
  forall X Y, rk(X ++ Y) + rk(list_inter X Y) <= rk X + rk Y.

Parameter rk_singleton_ge :
  forall A, rk (A :: nil) >= 1.
Parameter rk_couple_ge :
  forall A B, ~ A = B -> rk(A :: B :: nil) >= 2.

Parameter rk_three_points_on_lines :
  forall A B, exists C, rk (A :: B :: C :: nil) = 2 /\
    rk (B :: C :: nil) = 2 /\
    rk (A :: C :: nil) = 2.

Parameter rk_inter :
  forall A B C D, rk(A :: B :: C :: D :: nil) <= 3 ->
    exists J : Point, rk(A :: B :: J :: nil) = 2 /\
      rk (C :: D :: J :: nil) = 2.

Parameter rk_lower_dim :
  exists A0 A1 A2 A3, rk( A0 :: A1 :: A2 :: A3 :: nil) = 4.
Parameter rk_upper_dim :
  forall e, rk(e) <= 4.

```

B An example of an automatically generated file

The file `pprove_ex2.v`, which is 163-line long including comments, is automatically generated in the process of proving the statement `ex2`. It is available (or can be regenerated) from https://github.com/magaud/projective-prover/blob/master/theories/pprove_ex2.v. All statements, including intermediary statements have exactly the same hypotheses. In addition, there is exactly one statement for each deduction achieved during the saturation phase of the algorithm. The last statement `LABC` corresponds to the actual statement proved automatically. This lemma is then reused to prove the initial Coq goal: `ex2` in our example.

Require Import lemmas_automation_g.

```
Lemma LB : forall A B C D ,
rk(A :: C :: nil) = 2 -> rk(A :: B :: D :: nil) = 3 ->
rk(C :: D :: nil) = 2 -> rk(A :: C :: D :: nil) = 2 ->
rk(B :: nil) = 1.
Proof. [...] Qed.
```

```
Lemma LAC : forall A B C D ,
rk(A :: C :: nil) = 2 -> rk(A :: B :: D :: nil) = 3 ->
rk(C :: D :: nil) = 2 -> rk(A :: C :: D :: nil) = 2 ->
rk(A :: C :: nil) = 2.
Proof. [...] Qed.
```

[...]

```
Lemma LABC : forall A B C D ,
rk(A :: C :: nil) = 2 -> rk(A :: B :: D :: nil) = 3 ->
rk(C :: D :: nil) = 2 -> rk(A :: C :: D :: nil) = 2 ->
rk(A :: B :: C :: nil) = 3.
Proof.
intros A B C D
HACeq HABDeq HCDeq HACDeq .
assert(HABCM2 : rk(A :: B :: C :: nil) >= 2).
{
try assert(HACeq : rk(A :: C :: nil) = 2)
by (apply LAC with (A := A) (B := B) (C := C) (D := D) ;
try assumption).
assert(HACmtmp : rk(A :: C :: nil) >= 2)
by (solve_hyps_min HACeq HACM2).
assert(Hcomp : 2 <= 2) by (repeat constructor).
assert
  (Hincl : incl (A :: C :: nil) (A :: B :: C :: nil))
  by (repeat clear_all_rk;my_inO).
assert
  (HT := rule_5 (A :: C :: nil) (A :: B :: C :: nil)
    2 2 HACmtmp Hcomp Hinc1);apply HT.
}
assert(HABCM3 : rk(A :: B :: C :: nil) >= 3).
[...]
assert(HABCM : rk(A :: B :: C :: nil) <= 3)
by (solve_hyps_max HACeq HABCM3).
assert(HABCM : rk(A :: B :: C :: nil) >= 1)
by (solve_hyps_min HACeq HABCM1).
intuition.
Qed.
```

Certifying CNF Encodings of Pseudo-Boolean Constraints

(abstract)

Stephan Gocht

Lund University
Lund, Sweden

University of Copenhagen
Copenhagen, Denmark

`stephan.gocht@cs.lth.se`

Jakob Nordström

University of Copenhagen
Copenhagen, Denmark

Lund University
Lund, Sweden

`jn@di.ku.dk`

Ruben Martins


Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

`rubenm@andrew.cmu.edu`

The dramatic improvements in Boolean satisfiability (SAT) solving since the turn of the millennium have made it possible to leverage state-of-the-art conflict-driven clause learning (CDCL) solvers for many combinatorial problems in academia and industry, and the use of proof logging has played a crucial role in increasing the confidence that the results these solvers produce are correct. However, the conjunctive normal form (CNF) format used for SAT proof logging means that it is hard to extend the method to other, stronger, solving paradigms for combinatorial problems.

We show how to instead leverage the cutting planes proof system to provide proof logging for pseudo-Boolean solvers that translate pseudo-Boolean problems (a.k.a 0-1 integer linear programs) to CNF and run CDCL. We are hopeful that this is just a first step towards providing a unified proof logging approach that will also extend to maximum satisfiability (MaxSAT) solving and pseudo-Boolean optimization in general.

Alethe: Towards a Generic SMT Proof Format (extended abstract)

Hans-Jörg Schurr 

University of Lorraine, CNRS, Inria, and LORIA, Nancy, France
hans-jorg.schurr@inria.fr

Haniel Barbosa 

Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
hbarbosa@dcc.ufmg.br

Mathias Fleury 

Johannes Kepler University Linz, Austria
mathias.fleury@jku.at

Pascal Fontaine 

University of Liège, Belgium
pascal.fontaine@uliege.be

The first iteration of the proof format used by the SMT solver veriT was presented ten years ago at the first PxTP workshop. Since then the format has matured. veriT proofs are used within multiple applications, and other solvers generate proofs in the same format. We would now like to gather feedback from the community to guide future developments. Towards this, we review the history of the format, present our pragmatic approach to develop the format, and also discuss problems that might arise when other solvers use the format.

Over the years the production of machine-consumable formal proofs of unsatisfiability from SMT solvers [6] has attracted significant attention [4]. Such proofs enable users to certify unsatisfiability results similarly to how satisfiable results may be certified via models. However, a major difficulty that SMT proof formats must address is the complex and heterogeneous nature of SMT solvers: a SAT solver drives multiple, often very different, theory solvers; instantiation procedures generate ground instances; and heavily theory-dependent simplification techniques ensure that the solvers are fast in practice. Moreover, how each of these components works internally can also differ from solver to solver. As a testament to these challenges proof formats for SMT solvers have been mostly restricted to individual solvers and no standard has emerged. To be adopted by several solvers, an SMT proof format must be carefully designed to accommodate needs of specific solvers. This will require repeated refinement and generalization.

The basis for our efforts in this field is the proof format implemented by the SMT solver veriT [8] that is now mature and used by multiple systems. To further improve the format, as well as to accommodate not only the reasoning of the SMT solver veriT but also of other solvers, we are currently extending the format and developing better tooling, such as an independent proof checker. To facilitate this effort and overall usage, we are also writing a full specification. To emphasize the independence of the format we are baptizing it *Alethe*.¹ We do not presume to propose a standard format a priori. Instead we believe that Alethe, together with its tooling, can provide a basis for further discussions on how to achieve a format to be used by multiple solvers.

1 The State of Alethe

Alethe combines two major ideas whose roots reach back ten years to the first PxTP workshop in 2011 [7, 9]. It was proposed as an easy-to-produce format with a term language very close to SMT-LIB [5], the standard input language of SMT solvers, and rules with a varying level of granularity, allowing implicit

¹Alethe is a genus of small birds and the name resembles *aletheia*, the Greek word for truth.

proof steps in the proof and thus relying on powerful proof checkers capable of filling the gaps. Since then the format has been refined and extended [2]. It is now mature, supports coarse- and fine-grained proof steps capturing SMT solving for the SMT-LIB logic UFLIRA² and can be reconstructed by the proof assistants Coq [1, 11] and Isabelle [12, 14]. In particular, the integration with Coq was also used as a bridge for the reconstruction of proofs from the SMT solver CVC4 [3] in Coq, where its proofs in the LFSC format [15] were first translated into the veriT format before reconstruction. Finally, the format will also be natively supported in the upcoming `cvc5. solver`³

On the one hand, Alethe uses a natural-deduction style calculus driven mostly by resolution [7]. To handle first-order reasoning, dedicated quantifier instantiation rules are used [9]. On the other hand, it implements novel ideas to express reasoning typically used for processing, such as Skolemization, renaming of variables, and other manipulations of bound variables [2]. While the format was always inspired by the SMT-LIB language, we recently [12] changed the syntax of Alethe to closely resemble the command structure used within SMT-LIB. When possible Alethe uses existing SMT-LIB features, such as the `define-fun` command to define constants and the `:named` annotation to implement term sharing.

The following proof fragment gives a taste of the format. The fragment first renames the bound variable in the term $\exists x. f(x)$ from x to vr and then skolemizes the quantifier. A proof is a list of commands. The *assume* command introduces an assumption, *anchor* starts a subproof, and *step* denotes an ordinary proof step. Steps are annotated with an identifier, a rule, and premises. The SMT-LIB command *define-fun* defines a function. The rule *bind* used by step `t1` performs the renaming of the bound variable. It uses a subproof (Steps `t1.t1` and `t1.t2`). The subproof uses a context to denote that x is equal to vr within the subproof. The *anchor* command starts the subproof and introduces the context. The *bind* rule does not only make it possible to rename bound variables, but within the subproof it is possible to simplify the formula as done during preprocessing. The steps `t2` and `t3` use resolution to finish the renaming. In step `t4` the bound variable is skolemized. Skolemization uses the choice binder ε and derives $f(\varepsilon vr. f(vr))$ from $\exists vr. f(vr)$. To simplify the reconstruction the choice term is introduced as a defined constant (by *define-fun*). Finally, resolution is used again to finish the proof.

```
(assume a0 (exists ((x A)) (f x)))
(anchor :step t1 :args (:= x vr))
(step t1.t1 (cl (= x vr)) :rule cong)
(step t1.t2 (cl (= (f x) (f vr))) :rule cong)
(step t1 (cl (= (exists ((x A)) (f x))
               (exists ((vr A)) (f vr)))) :rule bind)
(step t2 (cl (not (= (exists ((vr A)) (f x))
                    (exists ((vr A)) (f vr))))
         (not (exists ((vr A)) (f x)))
         (exists ((vr A)) (f vr))) :rule equiv_pos1)
(step t3 (cl (exists ((vr A)) (f vr))) :premises (a0 t1 t2) :rule resolution)
(define-fun X () A (choice ((vr A)) (f vr)))
(step t4 (cl (= (exists ((vr A)) (f vr)) (f X))) :rule sko_ex)
(step t5 (cl (not (= (exists ((vr A)) (f vr)) (f X)))
         (not (exists ((vr A)) (f vr)))
         (f X)) :rule equiv_pos1)
(step t6 (cl (f X)) :premises (t3 t4 t5) :rule resolution)
```

²That is the logic for problems containing a mix of any of quantifiers, uninterpreted functions, and linear arithmetic.

³<https://cvc4.github.io/2021/04/02/cvc5-announcement.html>

The output of Alethe proofs from veriT has now reached a certain level of maturity. The 2021 version of the Isabelle theorem prover was released earlier this year and supports the reconstruction of Alethe proofs generated by veriT. Users of Isabelle/HOL can invoke the `smt` tactic. This tactic encodes the current proof goal as an SMT-LIB problem and calls an SMT solver. Previously only the SMT solver Z3 was supported. Now veriT is supported too. If the solver produces a proof, the proof is reconstructed within the Isabelle kernel. In practice, users will seldom choose the `smt` tactic themselves. Instead, they call the Sledgehammer tool that calls external tools to find relevant facts. Sometimes, the external tool finds a proof, but the proof cannot be imported into Isabelle, requiring the user to write a proof manually. The addition of the veriT-powered `smt` tactic halves [14] the rate of this kind of failures. The improvement is especially pronounced for proofs found by CVC4. A key reason for this improvement is the support for the conflicting-instance instantiation technique within veriT. Z3, the singular SMT solver supported previously, does not implement this technique. Nevertheless, it is Alethe that allowed us to connect veriT to Isabelle, and we hope that the support for Alethe in other solvers will ease this connection between powerful SMT solvers and other tools in the future.

The process of implementing proof reconstruction in Isabelle also helped us to improve the proof format. We found both, possible improvements in the format (like providing the Farkas' coefficient for lemmas of linear arithmetic) and in the implementation (by identifying concrete errors). One major shortcoming of the proofs were rules that combined several simplification steps into one. We replaced these steps by multiple simple and well-defined rules. In particular every simplification rule addresses a specific theory instead of combining them. An interesting observation of the reconstruction in Isabelle is that some steps can be skipped to improve performance. For example, the proofs for the renaming of variables are irrelevant for Isabelle since this uses De Bruijn indices. This shows that reconstruction specific optimizations can counterbalance the proof length which is increased by fine-grained rules. We will take this prospect into account as we further refine the format.

2 A Glance Into the Future

The development of the Alethe proof format so far was not a monolithic process. Both practical considerations and research progress — such as supporting fine-grained preprocessing rules — influenced the development process. Due to this, the format is not fully homogeneous, but this approach allowed us to quickly adapt the format when necessary. We will continue this pragmatic approach.

Speculative Specification. We are writing a speculative specification.⁴ During the development of the Isabelle reconstruction it became necessary to document the proof rules in a coherent and complete manner. When we started to develop the reconstruction there was only an automatically generated list of rules with a short comment for each rule. While this is enough for simple tautological rules, it does not provide a clear definition of the more complex rules such as the linear arithmetic rules. To rectify this, we studied veriT's source code and wrote an independent document with a list of all rules and a clear mathematical definition of each rule. We chose a level of precision for these descriptions that serves the implementer: precise enough to clarify the edge case, but without the details that would make it a fully formal specification. We are now extending this document to a full specification of the format. This specification is speculative in the sense that it will not be cast in stone. It will describe the format as it is in use at any point in time and will develop in parallel with practical support for the format within SMT solvers, proof checkers, and other tools.

⁴The current version is available at <http://www.verit-solver.org/documentation/alethe-spec.pdf>.

Flexible Rules. The next solver that will gain support for the Alethe format is the upcoming *cvc5* solver. Implementing a proof format into another solver reveals where the proof format is too tied to the current implementation of *veriT*. On the one hand, new proof rules must be added to the format — e.g., *veriT* does not support the theory of bitvectors, while *cvc5* does. When CVC4 was integrated into Coq via a translation of its LFSC proofs into Alethe proofs [11], an ad-hoc extension with bitvector rules was made. A revised version of this extension will now be incorporated into the upcoming specification of the format so that *cvc5* bitvector proofs can be represented in Alethe. Further extensions to other theories supported by *cvc5*, like the theory of strings, will eventually be made as well.

Besides new theories, *cvc5* can also be stricter than *veriT* in the usage of some rules. This strictness can simplify the reconstruction, since less search is required. A good example of this is the *trans* rule that expresses transitivity. This rule has a list of equalities as premises and the conclusion is an equality derived by transitivity. In principle, this rule can have three levels of “strictness”:

1. The premises are ordered and the equalities are correctly oriented (like in *cvc5*), e.g., $a = b$, $b = c$, and $c = d$ implies $a = d$.
2. The premises are ordered but the equalities might not be correctly oriented (like in *veriT*), e.g., $b = a$, $c = b$, and $d = c$ implies $d = a$.
3. Neither are the assumptions ordered, nor are the equalities oriented, e.g., $c = b$, $b = a$, and $d = c$ implies $d = a$.

The most strict variant is the easiest to reconstruct: a straightforward linear traversal of the premises suffices for checking. From the point of view of producing it from the solver, however, this version is the hardest to implement. This is due to implementations of the congruence closure decision procedure [13, 10] in SMT solvers being generally agnostic to the order of equalities, which can lead to implicit reorientations that can be difficult to track. Anecdotally, for *cvc5* to achieve this level of detail several months of work were necessary, within the overall effort of redesigning from scratch CVC4’s proof infrastructure. Since we cannot assume every solver developer will, or even should, undertake such an effort, all the different levels of granularity must be allowed by the format, each requiring different complexity levels of checking.

To keep the proof format flexible and proofs easy to produce, we will provide different versions of proof rules, with varying levels of granularity as in the transitivity example case above, by *annotating* them. This leverages the rule *arguments*, which are already used by some rules. For example, the Farkas’ coefficient of the linear arithmetic rule are provided as arguments. This puts pressure on proof checkers and reconstruction in proof assistants to support all the variants or at least the most general one (at the cost of efficiency). Hence, our design principle here is that the annotation is optional: the absence of an annotation denotes the least strict version of the rule.

Powerful Tooling. We believe that powerful software tools may greatly increase the utility of a proof format. Towards this end we have started implementing an independent proof checker for Alethe. In contrast to a proof-assistant-based reconstruction, this checker will not be structured around a small, trusted kernel, and correct-by-construction extensions. Instead, the user would need to trust the implementation does not lead to wrong checking results. Instead, its focus is on performance, support for multiple features and greater flexibility for integrating extensions and refinements to the format. The Isabelle checker is currently not suited to this task — one major issue is that it does not support SMT-LIB input files.⁵

⁵A version capable of doing so was developed for Z3 but it was unfortunately lost.

This independent checker will also serve as a proof “elaborator”. Rather than checking, it will also allow converting a coarse-grained proof, containing implicit steps, to a fine-grained one, with more detailed steps. The resulting proof can then be more efficiently checked by the tool itself or via proof-assistant reconstructions. An example of such elaboration is the transitivity rule. If the rule is not in its most detailed version, with premises in the correct order and none implicitly reordered, it can be elaborated by greedily reordering the premises and adding proof steps using the symmetry of equality. Note however that in the limit detailing coarse-grained steps can be as hard as solving an SMT problem. Should such cases arise, the checker will rely on internal proof-producing procedures capable of producing a detailed proof for the given step. At first the veriT and cvc5 solvers, which can produce fine-grained proofs for several aspects of SMT solving, could be used in such cases.

A nice side effect of the use of an external checker is that it could prune useless steps. Currently SMT solvers keep a full proof trace in memory and print a pruned proof after solving finishes. This is in contrast to SAT solvers that dump proofs on-the-fly. For SAT proofs, the pruned proof can be obtained from a full trace by using a tool like DRAT-TRIM. There is some ongoing work by Nikolaj Bjørner on Z3 to also generate proofs on-the-fly, but it is not clear how to support preprocessing and quantifiers.⁶

3 Conclusion

We have presented an overview of the current state of the Alethe proof format and some ideas on how we intend to improve and extend the format, as well as supporting tools. In designing a new proof format supported across two solvers we hope to provide a first step towards a format adopted by more solvers. This format allows several levels of detail, and is thus flexible enough to reasonably easily produce proofs in various contexts. We intend to define a precise semantics at each level though. This distinguishes our format from other approaches, such as the TSTP format [16], that are probably easier to adopt but only specify the syntax, leading to very different proofs generated by the various provers supporting it.

One limit of our approach for proofs is that we cannot express global transformations like symmetry breaking. SAT solvers are able to add clauses (DRAT clauses) such that the overall problem is equisatisfiable. It is unclear however how to add such clauses in the SMT context.

Overall, we hope to get feedback from users and developers to see what special needs they have and exchange ideas on the proof format.

Acknowledgment We thank Bruno Andreotti for ongoing work on the proof checker and Hanna Lachnitt for ongoing work on the cvc5 support. We are grateful for the helpful comments provided to us by the anonymous reviewers. The first author has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). The second author is supported by the LIT AI Lab funded by the State of Upper Austria.

References

- [1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In Jean-Pierre Jouannaud & Zhong Shao, editors: *Certified Programs and Proofs*, LNCS 7086, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

⁶<https://github.com/Z3Prover/z3/discussions/4881>

- [2] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury & Pascal Fontaine (2019): *Scalable Fine-Grained Proofs for Formula Processing*. *Journal of Automated Reasoning*, doi:10.1007/s10817-018-09502-y.
- [3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification (CAV)*, Springer, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.
- [4] Clark Barrett, Leonardo De Moura & Pascal Fontaine (2015): *Proofs in satisfiability modulo theories*. *All about proofs*, *Proofs for all* 55(1), pp. 23–44.
- [5] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2017): *The SMT-LIB Standard: Version 2.6*. Technical Report, Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.
- [6] Clark W. Barrett & Cesare Tinelli (2018): *Satisfiability Modulo Theories*. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith & Roderick Bloem, editors: *Handbook of Model Checking*., Springer, pp. 305–343, doi:10.1007/978-3-319-10575-8_11.
- [7] Frédéric Besson, Pascal Fontaine & Laurent Théry (2011): *A Flexible Proof Format for SMT: A Proposal*. In Pascal Fontaine & Aaron Stump, editors: *PxTP 2011*, pp. 15–26.
- [8] Thomas Bouton, Diego C. B. de Oliveira, David Déharbe & Pascal Fontaine (2009): *veriT: An Open, Trustable and Efficient SMT-solver*. In Renate A. Schmidt, editor: *CADE 22, LNCS 5663*, Springer Berlin Heidelberg, pp. 151–156, doi:10.1007/978-3-642-02959-2_12.
- [9] David Déharbe, Pascal Fontaine & Bruno Woltzenlogel Paleo (2011): *Quantifier Inference Rules for SMT Proofs*. In Pascal Fontaine & Aaron Stump, editors: *PxTP 2011*, pp. 33–39.
- [10] Peter J. Downey, Ravi Sethi & Robert Endre Tarjan (1980): *Variations on the Common Subexpression Problem*. *J. ACM* 27(4), pp. 758–771, doi:10.1145/322217.322228.
- [11] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds & Clark Barrett (2017): *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*. In Rupak Majumdar & Viktor Kunčák, editors: *CAV 2017, LNCS 10426*, Springer, pp. 126–133, doi:10.1007/978-3-319-63390-9_7.
- [12] Mathias Fleury & Hans-Jörg Schurr (2019): *Reconstructing veriT Proofs in Isabelle/HOL*. In Giselle Reis & Haniel Barbosa, editors: *PxTP 2019, EPTCS 301*, pp. 36–50, doi:10.4204/EPTCS.301.6.
- [13] Greg Nelson & Derek C. Oppen (1980): *Fast Decision Procedures Based on Congruence Closure*. *J. ACM* 27(2), pp. 356–364, doi:10.1145/322186.322198.
- [14] Hans-Jörg Schurr, Mathias Fleury & Martin Desharnais (2021): *Reliable Reconstruction of Fine-Grained Proofs in a Proof Assistant*. In: *CADE 28, LNCS*.
- [15] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean & Cesare Tinelli (2013): *SMT proof checking using a logical framework*. *Formal Methods in System Design* 42(1), pp. 91–118, doi:10.1007/s10703-012-0163-3.
- [16] Geoff Sutcliffe, Jürgen Zimmer & Stephan Schulz (2004): *TSTP Data-Exchange Formats for Automated Theorem Proving Tools*. In Weixiong Zhang & Volker Sorge, editors: *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, Frontiers in Artificial Intelligence and Applications* 112, IOS Press, pp. 201–215.