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

Zakaria Chihani, Dale Miller and Fabien Renaud

INRIA and LIX, École Polytechnique, Palaiseau, France

Abstract

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

1 Introduction

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

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

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

2 Trustworthy communication of theorems and proofs

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

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

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

2.1 Size of the proofs

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

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

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

2.2 Proof reconstruction

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

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

3 Proof checking as (logic) programming

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

3.1 Important programming language features

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

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

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

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

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

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

3.2 Logic foundations for these features

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

4 The checker's architecture

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

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

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

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

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

5 Avenues to trust

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

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

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

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

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

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

6 Related and Future Work

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`Dedukti/index.html`, 2013.

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

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