Designing Proof Formats: A User's Perspective — Experience Report

Sascha Böhme, Tjark Weber

Designing Proof Formats: A User's Perspective

— Experience Report —

Sascha Böhme
Technische Universität München

Tjark Weber
University of Cambridge

Abstract

Automatic provers that can produce proof certificates do not need to be trusted. The certificate can be checked by an independent tool, for example an LCF-style proof assistant such as Isabelle/HOL or HOL4. Currently, the design of proof formats is mostly dictated by internal constraints of automatic provers and less guided by applications such as checking of certificates. In the worst case, checking can be as involved as the actual proof search simply because important information is missing in the proof certificate. To address this and other issues, we describe design choices for proof formats that we consider both feasible for implementors of automatic provers as well as effective to simplify checking of certificates.

1 Introduction

The development of automatic theorem provers has seen much improvement in recent years, yet there is reason to doubt their soundness [6, 7]—the ultimate criterion for such tools. Solvers for propositional satisfiability (SAT), for quantified Boolean formulae (QBF), for satisfiability modulo theories (SMT), and automatic provers for first-order logic implement complicated algorithms. Hence, it is not surprising that bugs happen. As C. A. R. Hoare famously put it in his 1980 Turing Award Lecture [15]: "There are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies. The first method is far more difficult." State-of-the-art provers rely on sophisticated heuristics and ingenious optimizations to achieve striking performance. There is little hope of making them obviously correct.

A solution to this issue lies in requesting proof certificates from automatic provers that can be checked by an independent tool (Section 2). This approach pushes the soundness issue to the checker, which may be simple enough to be "obviously correct" or trustworthy for other reasons, for instance because it has been verified. In previous work we have used LCF-style proof assistants (Section 3), which are based on a small trusted kernel, to implement efficient checkers for SAT [25], QBF [17, 24] and SMT solvers [4], and have been working with proofs from first-order provers [3]. Our integrations achieved two goals. First, they provide high correctness assurances for these automatic provers. Second, they increase the degree of proof automation available in the LCF-style proof assistants Isabelle/HOL [20] and HOL4 [14].

The certificate-based approach is theoretically appealing, but our experience shows that there are practical pitfalls. Currently, the design of proof formats is mostly dictated by internal constraints of automatic provers. Checking should have lower complexity than proof search, yet proof certificates sometimes lack essential information, so that checking requires costly search already performed by the prover. Checking should be simple, yet proofs employ powerful inference steps with unclear semantics. Checking the results of a second prover should require little extra effort, yet similar provers use very different proof formats.

We report on our findings to alleviate the process of checking proof certificates, both technical as well as conceptual (Section 4), and abstract from specific problems of particular provers.

Pascal Fontaine, Aaron Stump (eds.); PxTP 2011, pp. 27-32

2 Automatic Provers and Their Proof Formats

The demand for proofs has been perceived many times by the automated reasoning community. Most automatic provers can now generate proof certificates. It is impossible to give an exhaustive list of systems, but we briefly review the state of proof formats for important classes of provers.

State-of-the-art SAT solvers expect their input to be in conjunctive normal form. Proofs of unsatisfiability can, therefore, be given as a sequence of resolution steps. Several SAT solvers, e.g., MiniSat and zChaff [25], define proof formats based on this observation. Despite the conceptual simplicity of resolution proofs, however, no two solvers seem to use the exact same proof format. Fostering the development of proof-producing SAT solvers, SAT competitions since 2005 have included a special *Verified UNSAT* track, where several different resolution-based proof formats are accepted. Among them, the RUP format [11] is relatively concise, while other (more verbose) formats can be checked more easily. Noting that simplicity of the checker is essential to obtain trustworthy results, Allen Van Gelder proposed that the language of certificates should be recognizable in deterministic log space [10].

Proofs of invalidity for quantified Boolean formulae are typically based on Q-resolution [8], an extension of propositional resolution. Proof formats for satisfiability show more diversity, with techniques including model generation and Skolemization. A recent overview can be found in [19]. Jussila et al. suggested a unified QBF proof format [16]. However, their calculus may not be rich enough to capture important algorithmic features found in certain QBF solvers. The SAT 2011 Competition will use another Q-resolution-based format, QIR [12], for proofs of invalidity. The QIR format can be checked in deterministic log space. Whether solver developers will adopt this new format remains to be seen.

Several SMT solvers produce proofs, e.g., clsat [21], CVC3 [2], Fx7 [18], veriT [5] and Z3 [9]. Each of them has a very distinct proof format. Only clsat and Fx7 specifically target efficient formal proof checking. The proof calculus underlying the CVC3, veriT and Z3 proof formats is natural deduction; CVC3 uses hundreds of inference rules, whereas less than 40 rules—some of which compress several reasoning steps—suffice for Z3. Fx7 uses natural deduction with a rewriting-based flavor, and clsat is based on the Edinburgh Logical Framework (LF). LF has been suggested as the basis for a standardized SMT proof format, as it promises efficient proof checking and generality, but its adoption is pending.

The first-order prover community has devised the TSTP proof format [23] as a standard for their proofs, although several first-order provers continue to provide certificates in their own formats. TSTP prescribes the syntax of proof certificates, and it is general enough to potentially cover certificates for other classes of provers. It does not, however, define or restrict the set of allowed inferences.

3 LCF-Style Proof Assistants

The term LCF-style [13] describes proof assistants based on a small inference kernel. Theorems are implemented as an abstract data type, and the only way to construct new theorems is through a fixed set of functions (corresponding to the underlying logic's axiom schemata and inference rules) provided by this data type, thus ensuring a small trusted code base. Proof procedures based on an LCF-style kernel cannot produce unsound theorems, as long as the implementation of the theorem data type is correct.

Using LCF-style proof assistants like Isabelle/HOL [20] and HOL4 [14] as proof checkers is instructive for three reasons. First, these systems implement rich logics, e.g., simply-typed higher-order logic with schematic polymorphism. Their language contains the subsets and extensions of first-order logic implemented by major automatic provers. We do not explicitly consider proof formats for higher-order logic in this paper, although much of our advice in Section 4 likely is transferable. Since proof assistants have not been optimized for a specific subset or background theory, they provide a generic testbed for the implementation of checkers.

Second, LCF-style proof assistants require that all inferences are validated by their kernel. Isabelle can even construct persistent proof terms during validation, but this feature is optional and not investigated here. Soundness errors in the proof checker, therefore, are extremely unlikely. No handwaving is possible: The checker must know all data required to make the necessary calls to kernel functions. Proof formats that lack essential information will be found out.

Third, various automatic proof procedures are available in Isabelle/HOL and other LCF-style proof assistants, notably a simplifier, which performs term rewriting, a decision procedure for propositional logic, tableau- and resolution-based first-order provers, and decision procedures for Presburger arithmetic and real algebra. Despite their simple, trustworthy inference kernel, LCF-style systems can, therefore, serve as checkers for arbitrarily complex inference rules and proof formats.

4 Guidelines for Proof Formats

The primary goals in designing a proof format are that generating, storing and checking certificates should be easy and fast. Unfortunately these goals are conflicting: Small certificates are easier to generate and store, but may not contain enough information to be checked efficiently. Based on our experience in using LCF-style proof assistants as checkers for various proof formats, we suggest more detailed guidelines for the design of new proof formats.

Use an existing format. There is general agreement among developers of automatic provers about standardized input formats. For instance, there is the DIMACS format for SAT solvers, its QDIMACS extension for QBF solvers, the SMT-LIB language for SMT solvers, and the TPTP language for first-order provers. These formats are widely supported by automatic provers. It is hence easy to run different provers on the same problem, and to exchange inferior provers for better ones.

When it comes to proof formats, the situation is very different. Few standards have emerged (see Section 2), and most automatic provers currently use their own format. This is unfortunate: Implementing an independent checker for any one format can require a considerable amount of time and effort. In the absence of standardized formats, users of proof certificates become locked in to specific provers. To avoid this, we need standardized proof formats in much the same fashion as we have standardized input formats.

Of course, a balance must be struck between standardization and progress. Designing proof formats for some classes of provers, e.g., for SMT solvers, is very much an active research area. We do not mean to discourage genuine advances in proof formats, but we want to point out the detrimental effect that the *Not Invented Here* syndrome has when prover developers needlessly reinvent concrete syntax. Sharing proof formats across provers will enable more re-use of checkers and other proof-based tools, and a better comparison of different automatic provers.

Provide a human-readable (yet easy to parse) representation. There are various ways for an automatic prover to make its proofs available to other tools, e.g., via a binary API, or via exporting proofs to binary or text files. An in-memory API may provide rich functionality, combined with short access times, and binary files may provide a storage-efficient representation.

In our experience, the perceived benefits of in-memory APIs and binary formats for proof exchange are small. Access times are usually insignificant when compared to runtimes for proof generation and proof checking. Even the size of proofs is rarely much of an issue, given that modern hard disk drives can routinely store terabytes of data. If necessary, generic or domain-specific data compression techniques [1] can be employed.

Moreover, binary formats are surprisingly brittle: Solvers and their interfaces keep changing. Documentation in practice is often outdated and incomplete. Bugs happen. This can make a significant amount of reverse engineering necessary to understand any binary proof format. In-memory APIs also require bridging different programming languages (between the prover and the checker), which can introduce additional complications.

Thus, in our experience, human-readable text files are clearly easiest to use. They facilitate proof debugging and provide a clear separation of tools. For instance, they make it easy to set up a remote prover that sends its certificates to a checker via HTTP [3, 22]. To simplify parsing of proofs, a standardized data format language, e.g., JSON, YAML or S-expressions, can be used.

Take theoretical considerations into account. Some proof formats are little more than quick hacks: Prover developers tend to include information in certificates only if it can readily be extracted from the prover's internal data structures. This sometimes leads to proof formats that contain redundant information, which could easily be reconstructed in the checker. More often, however, and much more gravely, it leads to proofs that contain an insufficient amount of detail. For instance, some proof-producing SMT solvers provide no justification for theorems derived by their theory-specific decision procedures.

Proof checking should not require significant amounts of search. While automatic provers may use sophisticated decision procedures and heuristics to find proofs, checkers should not have to re-implement those heuristics. Instead, proof checking should be reasonably easy—perhaps in deterministic log space, but certainly at most polynomial in the size of the proof. For a good proof format, considerations about complexity should take precedence over ease of prover implementation.

The complexity-theoretic perspective also provides a good guideline against including too much detail in proofs. It is not necessary to include every decision taken in the prover, if the result of that computation can instead be validated by a succinct (but easily checkable) certificate.

Use simple, canonical semantics. Proofs should not contain surprises. The number of different inference rules in a proof format will likely depend on the automatic prover: SAT solvers currently use just one rule (propositional resolution), while SMT solvers employ dozens or even hundreds of rules. Clearly, this can be attributed to the richer input language supported by the latter. But no matter what the number of rules is, each one should have simple and straightforward semantics. There is an analogy between inference rules in proofs and functions in programming—in fact, a checker will probably use the latter to implement the former. Small and focused inference rules are to be preferred, and complex rules should be broken into several smaller ones. Special cases and small, local optimizations are best avoided. These may be justified in the implementation of a prover, but they should not complicate the prover's interface—i.e., the proof format.

Use declarative proofs. Proofs apply inference rules to derive formulas. If each rule has clear, deterministic semantics (cf. the previous paragraph), the derived formulas could be left implicit. In our experience, however, the semantics of inference rules is often poorly defined and ambiguous. Are formulas treated modulo "obvious" equivalences, such as idempotence of logical connectives or α-equivalent renaming of bound variables? Are formulas implicitly transformed into a normal form of some kind? Are there corner cases where rules have unexpected semantics? The answers are prover-dependent and rarely obvious. Therefore, it is tremendously helpful when proofs contain not only inference rules, but also explicitly state the derived formulas. Declarative proofs facilitate proof debugging and enable checking of intermediate results. They allow to pinpoint errors in the proof (or in the checker) and to detect them early.

Provide exhaustive documentation. Independent of the proof format chosen, ample documentation should be provided. We have seen well-designed proof formats that were hard to check simply because the semantics of individual features was unclear, and no documentation gave any clue. Both the concrete and abstract syntax of the proof format, as well as its semantics, should be described in full detail. As an additional benefit, doing this thoroughly will likely help to identify proof features and inference rules that are overly complex.

Many automatic provers include a pre-processing phase to simplify the input and transform it into some canonical form. Pre-processing steps are rarely included in proof certificates: They typically do not involve search, so a checker can mimic them without additional information. Precisely for this reason, however, a prover's pre-processing phase needs to be documented as part of its proof format description.

An open-source reference implementation of a checker for the proof format is desirable, but it goes without saying that the source code of the checker (or of the automatic prover, if available) should not be considered sufficient documentation of the implemented proof format.

5 Conclusion

Automatic provers are complex software tools whose correctness is difficult to verify. Proof certificates that can be checked independently promise a solution to this issue. Certificates have several applications, from deciding the status of individual benchmark problems with great confidence to integrating automatic provers into LCF-style proof assistants without enlarging their trusted code base. Such integrations have been found to be fruitful for both worlds: Proof assistants benefit from increased automation, and automatic provers reach new domains and receive feedback as well as bug reports.

Currently, proof formats are usually designed by prover developers. Few independent proof checkers have been implemented, and the user community for proof certificates appears to be small. Consequently, the design of proof formats is mostly dictated by internal constraints of automatic provers and less guided by applications. We hope that by providing a user's perspective, this experience report can make a contribution to an informed debate over future proof formats.

A major nuisance for implementing checkers is the prevalent lack of standardized proof formats. There are only a handful of substantially different proof calculi in practical use, but hardly any two provers implement the same proof format. This situation is unfortunate as implementing independent checkers, especially efficient ones, tends to be involved. To overcome the lack of standardization is probably more of a social than a technical challenge. We believe that standard proof formats could advance the use of certificates as much as standard input formats have advanced the use of automatic provers in general.

Research into proof formats is continuing, and we look forward to better formats being proposed. We also expect to see new applications of proof certificates emerge, for instance in proof mining or machine learning. Our guidelines were designed to simplify proof checking; new applications will undoubtedly lead to additional requirements.

Acknowledgments. We acknowledge funding from EPSRC grant EP/F067909/1.

References

[1] H. Amjad. Data compression for proof replay. *J. Autom. Reasoning*, 41(3-4):193–218, 2008.

[2] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[3] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In *Automated Deduction*, 2011. To appear.

[4] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

[5] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *Automated Deduction*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.

[6] R. Brummayer and A. Biere. Fuzzing and delta-debugging SMT solvers. In *Satisfiability Modulo Theories*, 2009.

[7] R. Brummayer, F. Lonsing, and A. Biere. Automated testing and debugging of SAT and QBF solvers. In O. Strichman and S. Szeider, editors, *Theory and Applications of Satisfiability Testing*, volume 6175 of *LNCS*, pages 44–57. Springer, 2010.

[8] H. K. Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.

[9] L. M. de Moura and N. Bjørner. Proofs and refutations, and Z3. In P. Rudnicki, G. Sutcliffe, B. Konev, R. A. Schmidt, and S. Schulz, editors, *International Workshop on the Implementation of Logics*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[10] A. V. Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In J. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing*, volume 4501 of *LNCS*, pages 328–333. Springer, 2007.

[11] A. V. Gelder. Specification for reverse unit-propagation proof files, version 1.3, 2009. Retrieved May 5, 2011 from `http://users.soe.ucsc.edu/~avg/ProofChecker/fileformat_rup.txt`.

[12] A. V. Gelder. The QBF proof format: QIR, version 1.0, 2010. Retrieved May 5, 2011 from `http://users.soe.ucsc.edu/~avg/ProofChecker/fileformat_qir.txt`.

[13] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[14] M. J. C. Gordon and A. M. Pitts. The HOL logic and system. In *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems Series*, chapter 3, pages 49–70. Elsevier, 1994.

[15] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, 1981.

[16] T. Jussila, A. Biere, C. Sinz, D. Kröning, and C. M. Wintersteiger. A first step towards a unified proof checker for QBF. In J. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing*, volume 4501 of *LNCS*, pages 201–214. Springer, 2007.

[17] R. Kumar and T. Weber. Validating QBF validity in HOL4. In *Interactive Theorem Proving*, 2011. To appear.

[18] M. Moskal. Rocket-fast proof checking for SMT solvers. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 486–500. Springer, 2008.

[19] M. Narizzano, C. Peschiera, L. Pulina, and A. Tacchella. Evaluating and certifying QBFs: A comparison of state-of-the-art tools. *AI Communications*, 22(4):191–210, 2009.

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

[21] D. Oe, A. Reynolds, and A. Stump. Fast and flexible proof checking for SMT. In *Satisfiability Modulo Theories*, pages 6–13. ACM, 2009.

[22] G. Sutcliffe. System description: SystemOnTPTP. In D. McAllester, editor, *Automated Deduction*, volume 1831 of *LNAI*, pages 406–410. Springer, 2000.

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

[24] T. Weber. Validating QBF invalidity in HOL4. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 466–480. Springer, 2010.

[25] T. Weber and H. Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7(1):26–40, 2009.