The Proof Certifier Checkers*

Zakaria Chihani, Tomer Libal, and Giselle Reis

INRIA Saclay, France
{hichem.chihani,tomer.libal,giselle.reis}@inria.fr

Abstract. Different theorem provers work within different formalisms and paradigms, and therefore produce various incompatible proof objects. Currently there is a big effort to establish *foundational proof certificates* (FPC), which would serve as a common "specification language" for all these formats. Such framework enables the uniform checking of proof objects from many different theorem provers while relying on a small and trusted kernel to do so. Checkers is an implementation of a proof checker using foundational proof certificates. By trusting a small kernel based on (focused) sequent calculus on the one hand and by supporting FPC specifications in a prolog-like language on the other hand, it can be used for checking proofs of a wide range of theorem provers. The focus of this paper is on the output of equational resolution theorem provers and for this end, we specify the paramodulation rule. We describe the architecture of Checkers and demonstrate how it can be used to check proof objects by supplying the FPC specification for a subset of the inferences used by E-prover and checking proofs using these inferences.

1 Introduction

Many times software development faces the challenge of formal verification. This task can be accomplished by using a number of methods and available tools. Among such tools are theorem provers, which, upon proving a statement (automatically or interactively), provide a proof evidence. The problem faced nowadays is that such evidence comes in various formats, generally incompatible with each other. So if one is using a theorem prover, she must blindly trust the evidence provided, as it is not understood by any other system.

ProofCert [7] is a research project whose main goal is to bridge the gap between proof evidences. By using well-established concepts of proof theory, ProofCert proposes *foundational proof certificates* (FPC) as a framework to specify proof evidence formats. Describing a format in terms of an FPC allows softwares to check proofs in this format, much like a context-free grammar allows a parser to check the syntactical correctness of a program. The parser in this case would be a kernel: a small and trusted component that checks a proof evidence with respect to an FPC specification.

* Funded by the ERC Advanced Grant ProofCert.

H. De Nivelle (Ed.): TABLEAUX 2015, LNAI 9323, pp. 201–210, 2015.
DOI: 10.1007/978-3-319-24312-2_14

Checkers is the first implementation of a proof checking software which is based on FPC's. It uses an *LKF* (focused classical logic) kernel and comes with the FPC specification for paramodulation. It is applied to E-prover's [11] proof objects and therefore has also FPC's for some of the prover's inferences. Additionally, it includes a parser that translates E-prover's proofs into proof certificates. Checkers is a proof-of-concept implementation validating the feasibility of applying the ideas of ProofCert to "real life" theorem provers. Its development provided insights on practical challenges of such systems and clarified the kind of compromises the provers and the checkers need to deal with. Fortunately, we have found that proof objects using the TPTP syntax [9] can be straightforwardly translated to a proof certificate for checkers. Unfortunately, as far as we know, no prover uses exactly this syntax, but an approximation of it. We explain this point further in Section 2.3 and discuss what are the characteristics required for our tool. We also use a simple and modular architecture which can be extended with other FPC's for other inferences and formats.

This paper is organized as follows: Section 2 explains the architecture of the proof checker software, each of its components in detail and an overview on the syntax for FPC's. Section 3 explains the experiments on E-prover's proof objects, the challenges faced and the solutions implemented. Section 4 compares checkers with other proof checking software. Finally, Section 5 concludes the paper pointing to future work.

2 Checkers

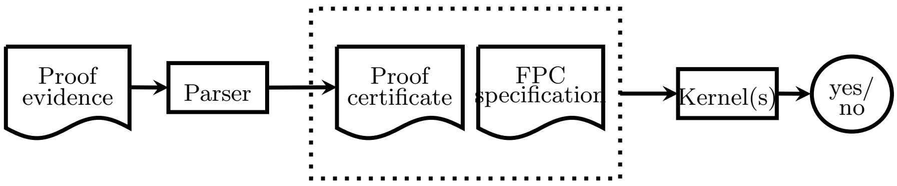

Fig. 1. High-level architecture of checkers.

The main components of checkers are depicted in Figure 1. Right now, we explain only briefly the function of each one and how they relate to each other. On the next sections we give more details.

Proof Evidence. This is the actual proof we aim to check and the input for checkers. It is the output of a theorem prover which is supposed to describe a proof.

Parser. The parser is a software component that translates the proof evidence into a proof certificate in a format that can be understood by the kernel. Such translation should be purely syntactical, not performing any logical or semantic transformation on the proof evidence. Because of this requirement, it might be the case that the proof evidence of some provers need to be adapted to give more (or less) structured information[1]. The parser in checkers is implemented in OCaml using `ocamlyacc` and `ocamllex`.

Proof Certificate. This file is generated by the parser for a given proof evidence file. Ideally it contains the same proof as the later, only in a different syntax. In practice, we are skipping a pre-processing step (clausal normal form transformation) for simplification purposes, but checking E-prover's CNF transformation is trivial as it is the standard deterministic one. Therefore, as of now, checkers will verify that the proof evidence represents a proof (or refutation) of the clauses after the input problem is transformed to clausal normal form. We expect to fill in this gap in the future. The clauses and inferences operating on them are the content of the proof certificate. The certificate needs to be in a comprehensible syntax for the kernel. As the kernel is implemented in λProlog [8], the certificate is composed of a λProlog module (`.mod` file) and signature (`.sig` file).

FPC Specification. In contrast to the proof certificate, which is generated for each proof evidence, an FPC specification corresponds to the proof format of a theorem prover. Every proof evidence file of the same theorem prover should be checked with the same FPC specification. The content of these files is a specification of *clerks* and *experts* [3]. These are predicates which will interface with the kernel and provide directives during proof checking (e.g. which term to choose for an existential quantifier or which formula to decompose next). The files are also a λProlog module and signature.

Kernel(s). The kernel is the key component of checkers. It consists of a small and trustable implementation of the focused sequent calculus for classical logic (*LKF*). The choice to implement this component in λProlog is due to the fact that rules in sequent calculi are straightforwardly encoded in a logic programming language and proof construction is directly represented by the execution of logic programs. Inferences in the theorem prover's system are ultimately translated to derivations in the kernel's system, i.e., *LKF*. The correctness of each inference is guaranteed by the correctness of the corresponding derivation (*adequacy*). A small and trusted kernel increases the confidence that such derivations are correct.

2.1 Kernels

The kernel is an implementation of the LKF^a [4] sequent calculus. The λProlog language [8] was chosen for the implementation because of its direct support for λtree syntax, hypothetical reasoning capabilities, typing mechanism and logic-based module system. It is a logic programming language based on so called

[1] In fact, this has already triggered a dialog between us and developers of theorem provers, including Stephan Schultz, the developer of E-prover.

hereditary Rasiowa-Harrop formulas (or hRHf) instead of the less expressive Horn clauses. The *LKF*^{a} system is obtained by augmenting the *LKF* system[6] with a communication protocol. Relying on the focusing behavior, this protocol allows the proof certificate to interact with the kernel, providing guiding information at specific moments. These moments are justified from the focusing paradigm itself: focused systems organize a proof into invertible phases, where only reversible rules are used, and focused phases, where a single formula is selected to be subject to a sequence of potentially non-invertible rules. The robustness of the kernel is twofold. First, augmenting *LKF* with this protocol is soundness-preserving, i.e., the kernel will never accept a falsehood regardless of how badly (or even maliciously) a proof certificate is written. Second, the kernel is an inference-based system whose implementations, in presence of backtracking and unification, are concise: for each inference rule in *LKF*^{a}, there is a hRHf predicate (taking 2 to 4 lines) that is a direct writing of that rule, making the code highly readable. Some additional basic predicates are added for basic testing such as list membership. The current implementation of the kernel is about one hundred lines long. Understanding focusing or background in logic programming are not required (but particularly helpful) for using this system.

2.2 Semantics of Proof Evidence

One of the main desiderata for the ProofCert project is the ability to check proof evidence in a wide variety of formats or languages. One way of doing so is to *translate* all proof evidence into one language and check that language. This method, reminiscent of Automath, is used by the proof checker Dedukti [2] which translates all outputs of provers based on the lambda-cube, as well as some classical theorem provers, into $\lambda\Pi$-modulo theory. The ProofCert project takes a different approach. Instead of *translating* proof evidence from the original language L to some other unique language, (thus altering the notion of proof known to the user), the semantics of the language L are defined in a relational setting such that the kernel checker can *perform* any proof evidence written in that language, much like one can, based on the semantics of a programming language, define interpreters compatible with that semantics that perform any programs written in that language. Using a relational instead of a functional setting allows for various level of details in proof evidence when the kernel is supplied with proof reconstruction abilities. For instance, a witness for an existential can be left out of proof evidence and found through unification during proof checking. A semantics definition for a language L, added to parsed proof evidence written in that language, yields a proof certificate.

A paramodulation proof, in the sense of Robinson & Wos [10], is a series of steps where each step introduces a non axiomatic formula from the paramodulation *from* a second formula *into* a third formula. These steps may or may not exactly specify the subterms on which the paramodulation is done. In this *base* language, a proof consists of *ordered* triples of formulas, or of indices of formulas.

Consider the following example:

1. $h(g(g(c))) \neq g(g(g(c)))$
3. $\forall X_1.\forall X_2.h(f(g(X_1), X_2)) = g(X_2)$
4. $\forall X_1.f(X_1, g(X_1)) = g(X_1)$
2. $\forall X_1.h(g(g(X_1))) = g(g(g(X_1)))$ (from 3 into 4)
0. $false$ (rewriting on 1 and 2)

This is an arguably reasonable output to request from a paramodulation-based prover. Indeed, many possible sophisticated strategies and heuristics can be used by such a tool but if it is, indeed, based on paramodulation, it should come at close to no cost to output a paramodulation-like proof in a language that resembles the *base* language mentioned above. In the case of E-prover, the output does not always resemble a paramodulation proof and we restricted our efforts to those E-prover outputs that *are* paramodulation-like.

2.3 Certificate

As mentioned before, the certificate built from a proof evidence is a λProlog module [8]. This means it is composed of two files: a module (extension `.mod`) and its signature (extension `.sig`). The module contains one predicate describing the proof of the following shape:

```
resProblem Name Clauses Inferences Map.
```

The `resProblem` predicate specifies a resolution refutation of an unsatisfiable set of clauses. Additionally, the module contains predicates of the form `inSig f.` for declaring each function symbol `f` occurring in the proof. The arguments of `resProblem` are:

– `Name`: This is a string representing the name of the problem. It can be any name chosen by the user and it should be enclosed in double quotes.
– `Clauses`: This is a list of clauses which are refuted. They are represented using the `pr` (for *pair*) term constructor that takes as arguments an integer (the index of the clause) and a clause.
– `Inferences`: The actual inferences of the proof are encoded with the `resteps` term constructor. The argument of this constructor is a list of inferences in the shape `inf (id (idx i)) (id (idx j)) k`, where `inf` is an inference name declared in the FPC and corresponding exactly to an inference used by E-prover, `id` is a constructor mapping an index to a clause and `k` is an index. The semantics of this constructor is that inference `inf` is applied to clauses with indices `i` and `j`, resulting in the clause with index `k`. The order of such inferences must be the same one as in the proof.
– `Map`: This is a function `map` which takes a list of `pr` terms mapping indices to clauses. These are all the clauses used in the proof.

```
type f i -> i -> i. type c i. type g i -> i. type h i -> i.
```

Fig. 2. Proof certificate: signature

```
resProblem "simple" [
  (pr 4 (all (X1\ (n ((f X1 (g X1)) == (g X1)))))),
  (pr 3 (all (X1\ (all (X2\
    (n ((h (f (g X1) X2)) == (g X2)))))))),
  (pr 1 (p ((h (g (g c))) == (g (g (g c))))))]
(resteps [pm (id (idx 3)) (id (idx 4)) 2,
          rw (id (idx 1)) (id (idx 2)) 0,
          cn (id (idx 0)) 0])
(map [
  pr 4 (all (X1\ (n ((f X1 (g X1)) == (g X1))))),
  pr 3 (all (X1\ (all (X2\
    (n ((h (f (g X1) X2)) == (g X2))))))),
  pr 0 f-,
  pr 2 (all (X1\ (n ((h (g (g X1))) == (g (g (g X1))))))),
  pr 1 (p ((h (g (g c))) == (g (g (g c)))))
]).
inSig h.
inSig g.
inSig f.
```

Fig. 3. Proof certificate: module

The signature file contains simply the type declarations of all the symbols used in the certificate. Figures 2 and 3 show the signature and module files for a proof certificate of the paramodulation proof in Section 2.2.

The TPTP format for problems and proofs consists of a set of predicates of the following shape [9]:

$$language(name, role, formula, source, useful\_info).$$

In the case of proofs, *source* is a `file` predicate (in case the formula is obtained from the input file) or an `inference` predicate (in case the formula is the result of applying an inference to other formulas). Most provers do not use exactly such format though, but some variant of it. One of the reasons for choosing E-prover for our experiments was because its output in TPTP syntax comes closest to the formal specitication of the TPTP format[2].

Transforming a file with such predicates into a proof certificate in our syntax is fairly straightforward. One needs simply to collect all the formulas and how they were derived. If *source* is `file`, then the formula is an axiom. If *source* is `inference`, it has the shape:

$$\texttt{inference(}inference\_name, inference\_info, parents\texttt{)}.$$

[2] By private communication with Geoff Sutcliffe.

This means the formula was a result of applying *inference_name* to *parents*. An important requirement is that the parents must be names of previously computed clauses or axioms (as specified in [9]). Unfortunately, a large number of proofs from E-prover contain nested inferences: the parents are not names of clauses but other `inference` predicates. Take the following line coming from an actual E-prover proof (where `F` is some formula):

```
cnf(c_0_6, negated_conjecture, F, inference(rw, [status(thm)],
    [inference(rw, [status(thm)], [c_0_3, c_0_4]), c_0_4])).
```

This line specifies that a rewriting step is done on clauses c_0_3 and c_0_4, obtaining some intermediate clause c, which is used in another rewriting step with c_0_4 to obtain c_0_6. Wihtout knowing the intermediate clause, we would need to perform proof search in order to guess what is derived and how it is used afterwards. This search might be non-terminating, and therefore, would not be much different than theorem proving itself. It is admissible for a proof checker to perform simple and decidable proof search, but anything more complicated than that will defeat its purpose. For this reason we have decided to work only with what we call *proper* proof objects, i.e., those that name all clauses used in the proofs and list the parents of each inference using these names.

Such requirement should not be considered a drawback of our approach, but a step in the direction of uniformization. We note that, for the SAT community, the `tracecheck` format has a similar requirement. Moreover, the feature of outputting the proof with all intermediary steps is in the future plans for E-prover (as we were informed by its principal developer).

Given a proof in the TPTP syntax, we can build a directed acyclic graph, from where the proof certificate can be extracted. Since we do not yet have a checking procedure for the normalization of input formulas (transformation to clausal normal form), when traversing the graph, whenever a clause resulting from normalization is encountered, we consider it to be an axiom instead of searching for its parents.

3 Experiments

A natural set of problems on which to experiment Checkers is the output of theorem provers on the TPTP library. Since we only support a subset of E-prover's inferences, namely paramodulation and rewriting, we restrict this set to 755 unsatisfiable problems of all difficulties and sizes and from different domains using the TPTP problem finder[3] by allowing only proofs having pure equations with unit equality.

A very interesting experiment would have been to try Checkers on wrong E-prover refutations of satisfiable clause sets. We could not find, however, such cases using the problem finder.

In the rest of this section, we describe our attempt of trying to certify E-prover refutations on the above set of problems. Our experiment consisted of running

[3] `http://www.cs.miami.edu/~tptp/cgi-bin/TPTP2T`

E-prover using a ten-minute timeout, parsing the result using our parser and then running Checkers on the generated files. The results of this paper were obtained using Checkers on `SHA a754baf` and Teyjus on `SHA 469c04e`.

As can be seen from Fig. 4, Checkers managed to certify all problems which we managed to parse, but we have managed to parse only a fraction of the produced proofs. Out of 639 TPTP proofs which were produced by E-prover using our timeout (giving enough time, E-prover can refute all 755 problems in the set), we were able to parse only 10 problems into proof certificates which could be fed to Checkers. The certificates obtained for these problems can be found in `src/tests/eprover` and can be checked by running the program. 39 problems indeed failed because of our missing support for the whole range of E-prover inferences. But the vast majority failed to be parsed because they contained nested inferences. As discussed in Section 2.3, such constructions would require the proof checking software to perform (possibly non-terminating) proof search.

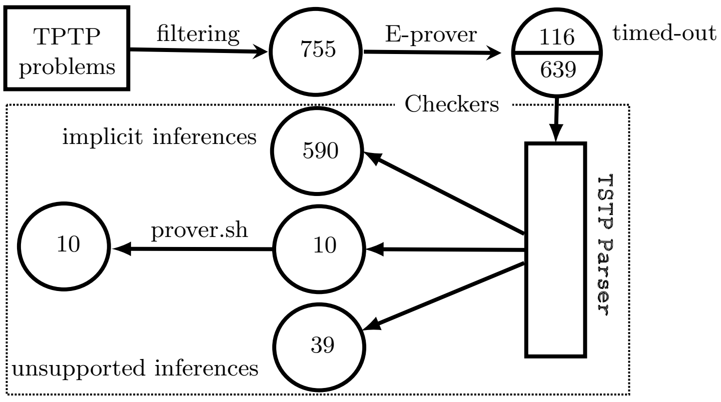

Fig. 4. Experimenting with Checkers, E-prover and the TPTP library.

4 Related Work

Currently there are several well-established tools for checking proofs, such as Mizar [5] for mathematical proofs, the EDACC [1] verifier for SAT solvers, LFSC [12] for SMT solvers and Dedukti for general proofs. Dedukti, being a universal proof checker (see Section 2.2 for a brief description), is the closest to our approach. The soundness of of Dedukti depends on the soundness of the translation into $\lambda\Pi$-terms and on the soundness of the rewriting rules. In general, most of the proof checkers mentioned above combine formal proof verification with non-verifiable computation component. This makes these proof checkers more practical on the one hand, but less trustable on the other. Since Checkers does not require translations of the theorem, its soundness depends only on that of its trusted kernel, making it a relatively trustable solution. The fact that its kernel is only about 100 lines, compared to about 1500 lines of Dedukti for example, increases even further its trustiness since the kernel can be implemented by various people and in various programming languages. On the other hand, Checkers can support computational steps in the form of modules of relational specifications (FPC), each giving the semantics of certain computations and which can be used by other FPC to define coarser deduction rules or computations. In this paper we have presented the FPC for the semantics of the standard paramodulation rule, which is used by the FPC of E-prover. But, one can provide modules of general term-rewriting rules as well.

Checkers can also be compared with theorem provers and proof assistants, such as Coq and Isabelle, which have a trusted kernel for checking their proofs. In order to use these tools, one has to translate the proofs objects to those used by these tools and also trust their kernels, which consist of thousands of lines of code. Therefore, the aim in the community is to use dedicated and more trusted checkers for certifying even the proofs of these tools, as can be seen by the translations of both Coq and HOL proofs into the language of Dedukti.

5 Conclusion

In this paper we have described a new tool for proof checking which is based on a small and trusted kernel and which aims on supporting a wide range of proof calculi and prover's outputs. The need for such a tool is growing since theorem provers are getting more complex and therefore, less trustable. For demonstration purposes, we have chosen to interface with E-prover, one of the leading theorem provers. Our choice was mainly based, as we mentioned before, on the fact that, while still not perfect, E-prover has the best support for TPTP syntax.

The main obstacle is the use of implicit inferences inside the proofs. In order to overcome that, one must replace these inferences by actual proofs obtained by search and this search might not terminate. This solution is both contradictory to the role of proof checkers and impractical due to our attempt to certify proofs using the sequent calculus.

Checkers supports a modular construct for the definition of the semantics of proof calculi (see Sec. 2.2). By writing FPC's, it is possible for implementors of theorem provers to give the semantics of their proof calculus which is required in order to complement the proof objects. The FPC for the semantics of E-prover's `pm` and `rw` inferences (see `/src/fpc/resolution/eprover`) is a good example for that as it consists only of a few lines of code. Understanding the FPC's for resolution and paramodulation, while not required, can help the implementors produce better proof objects which might improve proof certification.

The main shortcoming of using Checkers to certify the proofs of a certain theorem prover lies in the fact that there is no clear definition of what is a "proper" proof object. We hope that the further development of Checkers for other proof formats will make it clear how such object should be defined and thus help implementors of theorem provers to have some guidelines on what information proof evidences should contain.

The next step is extending Checkers with other inferences from E-prover as well as experimenting with other formats. In the future we expect to use Checkers to certify the proofs generated in the CASC[4] competition. As of now, there is no possibility of checking whether the competitors are producing a valid proof and we hope that this feature will be greatly appreciated by the community. This will definitely require an effort on both sides and we wish to collaborate with theorem prover writers to agree on a proof evidence format that suits both sides.

References

1. Balint, A., Gall, D., Kapler, G., Retz, R.: Experiment design and administration for computer clusters for SAT-solvers (EDACC). JSAT (2010)
2. Boespflug, M., Carbonneaux, Q., Hermant, O.: The $\lambda\Pi$-calculus modulo as a universal proof language. In: Pichardie, D., Weber, T. (eds.) Proceedings of PxTP2012: Proof Exchange for Theorem Proving, pp. 28–43 (2012)
3. Chihani, Z., Miller, D., Renaud, F.: Checking foundational proof certificates for first-order logic (extended abstract). In: Blanchette, J.C., Urban, J. (eds.) Third International Workshop on Proof Exchange for Theorem Proving (PxTP 2013). EPiC Series, vol. 14, pp. 58–66. EasyChair (2013)
4. Chihani, Z., Miller, D., Renaud, F.: Foundational proof certificates in first-order logic. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 162–177. Springer, Heidelberg (2013)
5. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a Nutshell. Journal of Formalized Reasoning 3(2), 153–245 (2010)
6. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. Theoretical Computer Science 410(46), 4747–4768 (2009)
7. Miller, D.: Proofcert: Broad spectrum proof certificates, February 2011, an ERC Advanced Grant funded for the five years 2012-2016
8. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press, June 2012
9. Otten, J., Sutcliffe, G.: Using the tptp language for representing derivations in tableau and connection calculi. In: Schmidt, R.A., Schulz, S., Konev, B. (eds.) PAAR-2010. EPiC Series, vol. 9, pp. 95–105. EasyChair (2012)
10. Robinson, G., Wos, L.: Paramodulation and theorem-proving in first-order theories with equality. In: Automation of Reasoning, pp. 298–313. Springer (1983)
11. Schulz, S.: System description: E 1.8. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR-19 2013. LNCS, vol. 8312, pp. 735–743. Springer, Heidelberg (2013)
12. Stump, A., Reynolds, A., Tinelli, C., Laugesen, A., Eades, H., Oliver, C., Zhang, R.: LFSC for SMT Proofs: Work in Progress

[4] http://www.cs.miami.edu/~tptp/CASC/