How to Get More Out of Your Oracles

Luís Cruz-Filipe, Kim S. Larsen, and Peter Schneider-Kamp

Dept. Mathematics and Computer Science, Univ. Southern Denmark
Campusvej 55, 5230 Odense M, Denmark
{lcf,kslarsen,petersk}@imada.sdu.dk

Abstract. Formal verification of large computer-generated proofs often relies on certified checkers based on oracles. We propose a methodology for such proofs, advocating a separation of concerns between formalizing the underlying theory and optimizing the algorithm implemented in the checker, based on the observation that such optimizations can benefit significantly from adequately adapting the oracle.

1 Introduction

During the last decade, we have seen the advent of larger and larger computer-generated proofs, often based on exhaustive case analysis. To allow for independent verification, the programs performing such proofs also generate a trace, detailing their reasoning steps. These *proof witnesses* have been growing significantly in size, from a few MB [15] to a few GB [5, 13], culminating with the impressive 200 TB proof of the Boolean Pythagorean Triples conjecture [12].

Formal verification of such proofs amounts to checking whether the proof witnesses can be used to reconstruct a proof. Directly importing the witnesses into a theorem prover [4] does not scale to the size of recent proofs due to memory and computational requirements. Instead, the witnesses can be obtained from an external untrusted source, the *oracle*, and checked for correctness before use [14]. The formal proof is thus split between two components: the untrusted oracle and the proven correct proof checker. The latter needs to be correct regardless of its input data, typically by ignoring or rejecting incorrect data.

The benefit of using the oracle is efficiency: since its results are not trusted, it can be optimized for performing difficult computations efficiently. (The point is, of course, that these results should be correct, but there is no burden of proof of this fact.) The certified checker, on the other hand, typically constitutes the computational bottleneck of the overall system. Thus, in order to minimize execution time, it is natural to try to shift as much computation as possible from the checker to the oracle.

Traditionally, this path has not been explored to its full potential. Often oracles are queried when needed [9, 11, 14], computing witnesses at runtime. In other cases [2, 15, 16], the oracle is pre-computed and responsible for controlling the flow of the algorithm; in this case, the checker's queries amount to asking "What should I do next?". Our simple observation is: *the overall system of untrusted oracle and trusted checker can be optimized by utilizing the oracle*

maximally. We have identified a successful strategy for approaching this, which we feel deserves to be communicated.

Based on this observation, we propose a systematic methodology for using oracles in large-scale proofs (Section 2), modulizing the cooperation between the untrusted oracle and the certified checker. We identify the characteristics of problems whose proofs could and should profit from this methodology, and illustrate it (Section 3) using two cases: an optimality proof on sorting networks [6], which inspired this methodology, and a formalized checker for unsatisfiability proofs [7], obtained by directly applying this methodology. The latter ultimately was able to verify the 200 TB proof from [12], as described in [8].

2 Methodology

We first identify the characteristics a problem should have in order to benefit from our methodology. We motivate and illustrate these requirements by small toy examples. Then we present the methodology as a simple step-by-step procedure, with a focus on its separation of concerns.

Problem requirements. A common element to most oracles is that they relate to problems where finding proof witnesses is computationally much more expensive than checking their correctness. Because of this commonality, there is a significant pay-off in being able to write highly optimized untrusted code for finding witnesses.

Example 1. As an example, consider a proof that involves properties of the image of a function. Assume that in the process, one needs to find a pre-image of this object. If the function is easier to compute than its inverse, then an *ad hoc* way of finding pre-images can greatly improve performance.

A concrete extreme example is the inversion of a hash function, for which there is no realistic way of computing pre-images of given concrete hashes. An oracle might use large distributed Internet databases to try to find them, though. Such an oracle would by its nature be both incomplete (it fails when a pre-image exists, but is not in the database) and untrustworthy (the database could be erroneous), and therefore impossible to implement and prove correct inside a theorem prover such as Coq. However, its result is very simple to check. □

Requirement 1 (Existential subproblems). *The problem contains multiple occurrences of existential statements as subproblems, for which witness checking is computationally easier than witness generation.*

By "computationally easier", we simply mean that more efficient algorithmic solutions are known for one of the problems; we are not claiming that the problems provably belong to different complexity classes. If the condition in the requirement is met, this is an indication that the use of an oracle may be beneficial.

In general, a pre-computed oracle cannot be omniscient, since it can only provide finitely many different answers. Even if the problem domain is finite, it is still typically prohibitive to precompute all possible answers. Therefore, our methodology requires the set of problems that the oracle may be called upon to be sufficiently restricted (for the answers to fit into memory, for example).

Requirement 2 (Known subproblems). *There is a manageable set of subproblems that contains all subproblems encountered during the proof.*

Our last requirement is that changes to the answers provided by the oracle should have an impact on the control flow and (consequently) on the efficiency of the remainder of the proof. We illustrate this point by an example.

Example 2. Imagine a proof that requires factorizing certain composite numbers into sorted lists of their prime factors as a recurring step. Suppose also that we have an efficient oracle that, given a composite number, delivers one of its prime factors. The oracle will have to be called multiple times in order to obtain the list of all factors, and this list has to be sorted (either at construction time or after obtaining all factors).

If we compute all prime factors, sort them, and have the oracle provide them in sorted order, we can replace the sorting step in the proof by a simple verification that the list provided is sorted, making the proof both simpler and more efficient. Note that this potentially changes the set of subproblems the oracle will be called upon, since it may change the control flow of the checker; a fact that needs to be taken into consideration in the implementation. □

Requirement 3 (Data-dependent proof). *The structure of the proof is dependent on the answers provided by the oracle.*

In Example 2, this requirement would not be satisfied if the subproblems consisted of just showing that certain numbers were composite. The case studies in Section 3 illustrate all three requirements in realistic settings.

Step-by-step guide to verifying large proofs. We now describe our methodology for verifying large proofs that fit the requirements discussed above. This consists of four phases.

- *Formalize* the theory underlying the results
- *Implement* a naive checker (using an oracle) and prove it correct
- *Optimize* the checker in lock-step with adapting the oracle
- *Reprove* the correctness of the checker

In the *Formalize* phase, the focus is on the mathematical theory needed to prove the soundness of the algorithm used in the checker. The key aspect here is to separate concerns by *not* worrying about how these results will be used in the actual implementation. In other words, we advocate formalizing the theory as close as possible to the mathematical definitions and other formal elements of the algorithm.

In the *Implement* phase, the goal is to implement a checker as simple as possible. The algorithm of the checker should do as little work as possible, using the information in the proof witnesses as much as possible. This straightforward implementation must then be proven correct.

The *Optimize* phase is the most complex and most interesting one. In this phase, we analyze the complexity of the checker to determine possible local improvements. These can be of two kinds. The first kind is to use standard computer science techniques to optimize performance – for example, by using binary search trees instead of lists, or by enriching the proof witnesses to lower the complexity of checking their correctness. The second is to use the fact that all answers needed from the oracle are available beforehand to implement a more efficient algorithm, as illustrated by Example 2. In both cases, this may also require changes to the implementation of the oracle.

The *Reprove* phase consists of reproving the correctness of the optimized checker. This phase may be interleaved with the previous one, as optimizations tend to be localized and, thus, only require localized changes to the soundness proof. This is the case for optimizations of the implementation, in particular, where soundness is a property of the algorithm, and thus not significantly effected by the low-level choice of data structures. By applying one optimization at a time and immediately proving its soundness, it is easier to connect the necessary changes in the formalization to the original change in the algorithm.

The key observation in this methodology is the realization that the formalizations involved in different stages are of very different natures, and benefit from being treated separately. In the *Formalize* phase, the emphasis is on the underlying theory, and it will present the challenges present in any formalization – choosing the correct definitions, adapting proofs that are not directly formalizable, etc. In the *Implement* phase, the results that need to be formalized deal directly with the correctness of the algorithm being implemented, and will use the results proved in the earlier stage. Typically, the complexity of these proofs will arise from having to consider different cases or finding the right invariants throughout program execution, but not from mathematical issues pertaining to the underlying theory.

This is particularly relevant for the *Reprove* phase, where the modularity of the approach will have an impact in two ways. First, the formalization of the underlying theory for its own sake (rather than as a library tailored towards proving the correctness of the original algorithm) will make it more likely that all the needed results are readily available, and that they have been stated in forms making them more generically applicable. Second, changes to the algorithm will typically require different inductive proofs, but their correctness will likely use the same underlying arguments, which will already be available from previous phases. For example: if an algorithm iterating over lists is replaced by one iterating over trees, the structure of the soundness proof changes, but the key inductive argument (regarding how an additional element is processed) is unchanged. Therefore, the iterative steps in alternating *Optimize* and *Reprove* phases will likely be much simpler and faster than the original *Implement* phase.

As a consequence, the final library will also be more likely to be reusable in future proofs.

The requirements identified earlier are essential for this methodology to be meaningful. In general, existential subproblems indicate that using an untrusted oracle can be a good strategy, since verifying the proof witnesses it provides is easier than implementing a certified algorithm for computing them in the theorem prover. The known subproblems requirement guarantees that the oracle can pre-compute all proof witnesses that will be needed in the proof, so that they can be suitably processed before the checker is executed. Finally, data dependency ensures that changing the implementation of the oracle is meaningful, as it can improve the overall performance of the checker.

3 Case studies

We illustrate our methodology by referring to two previously published formalizations. While we used Coq [1] as the theorem prover in both, our methodology should be portable to other formal generic proof environments.

Optimal sorting networks. In [5], we describe a computer-generated proof of the optimality of 25 comparisons for data-independently sorting 9 inputs. This proof is based on an exhaustive case analysis, made feasible by a powerful, but computationally expensive (NP-complete) subsumption relation [6]. A proof witness consists of two comparator networks and a permutation under which the relation holds. While known algorithms for solving the *existential subproblem* (by finding a permutation) have worst-case exponential runtime, checking the relation given a permutation is much easier.

The subsumption relation is used to eliminate comparator networks that are subsumed by others. The structure of the proof is thus highly *data-dependent*, with the order in which proof witnesses are provided by the oracle influencing the set of subproblems encountered during the remainder of the proof. This is a challenge for the *known subproblems* requirement, which is solved by oracle pre-processing based on transitivity of subsumption.

In the *Formalize* and *Implement* phases, we made a direct implementation of the algorithm in the original proof from [5], obtaining a checker with an expected runtime of 20 years to process the 27 GB of proof witnesses. In the *Optimize* and *Reprove* phases, we optimized this algorithm by changing the order in which the oracle provides proof witnesses, which allowed us to use asymptotically better algorithms and data structures. These optimizations reduced the execution time to just under 1 week [6]. Separating the formalization of the theory and the correctness proof of the checker meant that the cost of *Reprove* was marginal – at most 1 day per major change – compared to *Formalize*, which took approx. 3 months.

Unsatisfiability proofs. More and more applications are relying on SAT solvers for exhaustive case analysis, including computer-generated proofs [12, 13]. While

formally verifying satisfiability results is trivial given a witness, verifying unsatisfiability results returned by untrusted solvers requires showing that the original formula entails the empty clause. To this end, many SAT solvers provide the entailed clauses they learn during execution as proof witnesses. Finding meaningful such clauses is clearly a non-trivial *existential subproblem*.

To check an unsatisfiability proof, the clauses provided by the oracle are added to the original set of clauses after their entailment has been checked by reverse unit propagation. This addition of clauses determines which further clauses can be added, i.e., the structure of the proof is *data-dependent*. Since the proof checker simply follows the information provided by the oracle, the *known subproblems* requirement is trivially satisfied.

By applying our methodology directly, we were able to improve the state-of-the-art of formally verifying unsatisfiability [10, 16] by several orders of magnitude. Here, the *Formalize* phase consisted simply of building a deep encoding of propositional logic in the theorem prover Coq and defining notions of entailment and satisfiability, and the *Implement* phase yielded a simple checker based on reverse unit propagation. In the *Optimize* phase, we achieved a performance improvement of several orders of magnitude by observing that the core algorithm for checking reverse unit propagation can be simplified significantly by enriching the proof witnesses with information about the clauses used [7]. This improvement in the performance of the checker was obtained at the cost of a noticeable (yet manageable) increase in computation time on the oracle side, due to the need to enrich the proof witnesses, but this shift ultimately allowed us to verify the 200 TB proof from [12], as described in [7, 8].

4 Concluding Remarks

We have introduced a methodology based on distilling the key idea behind our two case studies: the overall system of proof checker and oracle can profit from shifting the computational burden from the trusted, inefficient proof checker to the untrusted, efficient oracle implementation. In other words, we let the proof checker be implemented as efficiently as possible, doing as little work as possible, while pre-processing the oracle information such that the right amount information was provided in the right order. Since all the data provided by the oracle is verified by the proof checker, this does not affect the reliability of the results. By revisiting the case studies in this unifying presentation, we hope to inspire others to obtain similar performance gains when formally verifying other large-scale proofs.

Acknowledgments. We would like to thank Pierre Letouzey for his suggestions and help with making our extracted code more efficient.

The authors were supported by the Danish Council for Independent Research, Natural Sciences, grant DFF-1323-00247, and by the Open Data Experimentarium at the University of Southern Denmark. Computational resources were generously provided by the Danish Center for Scientific Computing.

References

1. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer, 2004.
2. F. Blanqui and A. Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comp. Sci.*, 21:827–859, Aug. 2011.
3. S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors. *Interactive Theorem Proving, ITP 2013, Proceedings*, volume 7998 of *LNCS*. Springer, 2013.
4. G. Claret, L. González-Huesca, Y. Régis-Gianas, and B. Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In Blazy et al. [3], pages 67–83.
5. M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp. Sorting nine inputs requires twenty-five comparisons. *J. Comput. Syst. Sci.*, 82(3):551–563, 2016.
6. L. Cruz-Filipe, K. S. Larsen, and P. Schneider-Kamp. Formally proving size optimality of sorting networks. *J. Autom. Reasoning*, accepted for publication.
7. L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp. Efficient certified resolution proof checking. In A. Legay and T. Margaria, editors, *TACAS*, volume 10205 of *LNCS*, pages 118–135. Springer, 2017.
8. L. Cruz-Filipe and P. Schneider-Kamp. Formally proving the boolean triples conjecture. In T. Eiter and D. Sands, editors, *LPAR-21*, volume 46 of *EPiC Series in Computing*, pages 509–522. EasyChair Publications, 2017.
9. L. Cruz-Filipe and F. Wiedijk. Hierarchical reflection. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *LNCS*, pages 66–81. Springer, 2004.
10. A. Darbari, B. Fischer, and J. Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In A. Cavalcanti, D. Déharbe, M. Gaudel, and J. Woodcock, editors, *ICTAC*, volume 6255 of *LNCS*, pages 260–274. Springer, 2010.
11. A. Fouilhé, D. Monniaux, and M. Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In F. Logozzo and M. Fähndrich, editors, *SAS*, volume 7935 of *LNCS*, pages 345–365. Springer, 2013.
12. M. Heule, O. Kullmann, and V. Marek. Solving and verifying the Boolean Pythagorean Triples problem via cube-and-conquer. In N. Creignou and D. Le Berre, editors, *SAT*, volume 9710 of *LNCS*, pages 228–245. Springer, 2016.
13. B. Konev and A. Lisitsa. A SAT attack on the Erdős discrepancy conjecture. In C. Sinz and U. Egly, editors, *SAT*, volume 8561 of *LNCS*, pages 219–226. Springer, 2014.
14. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
15. C. Sternagel and R. Thiemann. The certification problem format. In C. Benzmüller and B. Paleo, editors, *UITP*, volume 167 of *EPTCS*, pages 61–72, 2014.
16. N. D. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. Mechanical verification of SAT refutations with extended resolution. In Blazy et al. [3], pages 229–244.