Carcara: A proof checker and elaborator for SMT proofs

Bruno Andreotti, Haniel Barbosa
Universidade Federal de Minas Gerais

Introduction

Proofs from SMT solvers allow us to trust the correctness of their results without having to trust their implementation, which is often a requirement when solvers are used in safety-critical applications or proof assistants. Alethe is an established SMT proof format produced by the solvers veriT and cvc5, whose syntax is close to SMT-LIB and allows both coarse- and fine-grained steps, facilitating proof production. Alethe proofs can already be checked via reconstruction in the proof assistants Isabelle/HOL and Coq — however, the lack of a stand-alone independent checker harms its usability and hinders its adoption. Moreover, reconstruction in proof assistants typically presents poor performance, meaning coarse-grained steps can be too expensive to check and lead to verification failures. In this work, we present Carcara, an efficient and independent proof checker and elaborator for Alethe, implemented in Rust. It aims to increase the adoption of the format by providing push-button proof-checking for Alethe proofs, focusing on efficiency and usability. Carcara is also capable of transforming coarse-grained steps into more detailed, fine-grained ones. This process, called *proof elaboration*, can improve checking performance and increases the potential success rate of checking Alethe proofs in performance-critical validators, such as proof assistants.

The Alethe proof format

Alethe is a proof-assistant friendly, easy-to-produce proof format for SMT solvers. To facilitate proof production, Alethe uses a term language that directly extends SMT-LIB, thus not requiring solvers to translate between different term languages when outputting proofs. More importantly, Alethe's proof calculus provides rules with varying levels of granularity, allowing coarse-grained steps and relying on powerful proof checkers for filling in the gaps. This reduces the burden on developers to track all reasoning steps performed by the solver, a notoriously difficult task.

```
(set-logic LIA)
(assert (forall ((x Int)) (> x 0)))
(assert (not (forall ((y Int)) (> y 0))))
(check-sat)
```

```
(assume h1 (forall ((x Int)) (> x 0)))
(assume h2 (not (forall ((y Int)) (> y 0))))
(anchor :step t3 :args ((y Int) (:= x y)))
(step t3.t1 (cl (= x y)) :rule refl)
(step t3.t2 (cl (= (> x 0) (> y 0))) :rule cong :premises (t3.t1))
(step t3 (cl (= (forall ((x Int)) (> x 0)) (forall ((y Int)) (> y 0))))
  :rule bind)
(step t4 (cl (not (forall ((x Int)) (> x 0))) (forall ((y Int)) (> y 0)))
  :rule equiv1 :premises (t3))
(step t5 (cl) :rule resolution :premises (t4 h1 h2))
```

Fig. 1: A simple SMT-LIB problem and an Alethe proof of its unsatisfiability.

Alethe proofs have the form $\pi : \varphi_1 \wedge \cdots \wedge \varphi_n \to \bot$, i.e., they are refutations, where $\bot$ is derived from assumptions $\varphi_1, \ldots, \varphi_n$ corresponding to the original SMT instance being refuted. Figure 1 shows an example of an SMT-LIB problem and an Alethe proof of its unsatisfiability.

Proofs are a series of steps represented as an indexed list of `step` commands. The command `assume` is analogous to `step` but used only for introducing assumptions. Contexts are introduced by the `anchor` command, which opens subproofs. Subproofs simulate the effect of the $\Rightarrow$-introduction rule of Natural Deduction, where local assumptions are put in context and the last step in a subproof represents its conclusion and the closing of its context.

Proof checking

In order to check an Alethe proof, first, the original SMT-LIB problem and its proof are parsed. The problem provides the declaration of sorts and symbols that may be used in the proof, as well as the original assertions, which must match the assumptions in the proof. Terms are internally represented as directed acyclic graphs, using *hash consing* for maximal sharing and constant-time equality tests. The proof is represented internally as an array of command objects, each corresponding either to an Alethe `assume` or `step` command, or a subproof, which is represented as a step with an (arbitrarily) nested array of command objects.

Each command is checked individually by the rule checker corresponding to the rule in that command. That component takes as input the conclusion, the conclusions of its premises, and the arguments of the command, as well as the context it is in. As the Alethe format currently has 90 possible rules specified, Carcara has 90 rule checkers.

Some rules allow very coarse-grained steps, which presents particular challenges when checking. For example: steps of the `assume` and `refl` rules allow the implicit reordering of equalities, meaning the terms `(= a b)` and `(= b a)` are implicitly considered equal. This means that, when checking these rules, Carcara may need to completely traverse the terms to compare them for equivalence modulo equality reordering.

Proof elaboration

In order to mitigate bottlenecks in checking some Alethe steps, Carcara can also elaborate Alethe proofs into easier-to-check ones by filling in missing details from the original proofs. This is done by replacing coarse-grained steps with fine-grained proofs of their conclusions, producing a new overall proof equivalent to the original, but with some coarse-grained steps broken down into fine-grained ones. There are many Alethe rules whose checking would be simpler if elaborated, but we have focused initially on what we believe can be more impactful: removing implicit equality reordering, which affects virtually every Alethe rule; and providing checkable justifications for some linear arithmetic steps.

Evaluation

To evalute Carcara's proof checking performance, we used the veriT SMT solver to generate Alethe proofs from all problems in the SMT-LIB benchmark library whose logic it supports. This resulted in 39,299 proofs from 16 different logics, totalling 92 GB.

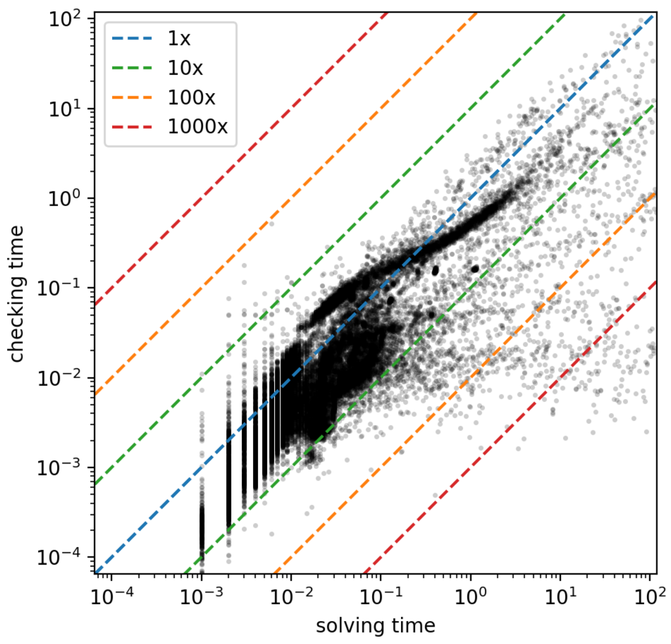

Fig. 2: Solving vs proof checking time.

Figure 2 shows, for every problem, the time taken to solve it and to check the proof produced. When comparing per-problem, for the large majority of proofs (81.61%) the checking time was smaller than the solving time. Furthermore, for 3.96% of the proofs, checking was more than 10 times faster than solving the problem, and for 0.96%, that ratio was of 100 times. There were only 24 instances where the checking time was more than 10 times bigger than the solving time, and, in all of them, the checking time was less than 0.6 seconds.

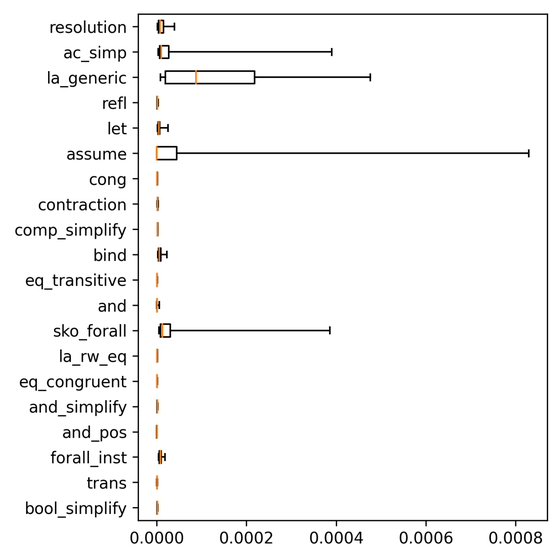

Fig. 3: Box plot for checking time per rule.

We also used Carcara to elaborate each successfully checked proof. While elaboration improved the time taken to actually check the proof steps by 6%, parsing time increased (because extra steps are added when elaborating), such that overall there is no clear performance gain from elaboration on average. However, some particular proofs benefit greatly from elaboration, and checking elaborated proofs is overall simpler, not requiring, for instance, to deal with the implicit reordering of equalities.

Conclusion and future work

Our evaluation shows that Carcara has good performance and can identify shortcomings in the proof-production of established SMT solvers. Carcara can also elaborate proofs into demonstrably easier-to-check ones, which can have a significant impact, for example, if it is used as a bridge between solvers and proof assistants. There is also ongoing work to extend Carcara to convert Alethe proofs into other formats, which would also allow the elaboration techniques to benefit other toolchains.