Carcara: An efficient proof checker and elaborator for SMT proofs in the Alethe format[*]

Bruno Andreotti[1] , Hanna Lachnitt[2] , Haniel Barbosa[1]

[1] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
[2] Stanford University, Stanford, USA

Abstract. Proofs from SMT solvers ensure correctness independently from implementation, which is often a requirement when solvers are used in safety-critical applications or proof assistants. Alethe is an established SMT proof format generated by the solvers veriT and cvc5, with reconstruction support in the proof assistants Isabelle/HOL and Coq. The format is close to SMT-LIB and allows both coarse- and fine-grained steps, facilitating proof production. However, it lacks a stand-alone checker, which harms its usability and hinders its adoption. Moreover, the coarse-grained steps can be too expensive to check and lead to verification failures. We present CARCARA, an 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; and by providing elaboration for coarse-grained steps into fine-grained ones, increasing the potential success rate of checking Alethe proofs in performance-critical validators, such as proof assistants. We evaluate CARCARA over a large set of Alethe proofs generated from SMT-LIB problems and show that it has good performance and its elaboration techniques can make proofs easier to check.

1 Introduction

Satisfiability modulo theories (SMT) solvers are widely used as background tools in various formal method applications, ranging from proof assistants to program verification [9]. Since these applications rely on the SMT solver results, they must trust their correctness. However, state-of-the-art SMT solvers are often found to have bugs, despite the best efforts of developers [30, 38]. One way to address this issue is to formally verify the solvers' correctness ("certifying" them), but this approach can be prohibitively expensive and time consuming, besides often requiring performance compromises [19, 20, 27, 33] and increasing the evolution cost of the systems [14]. Alternatively, solvers can produce proofs: independently checkable certificates that justify the correctness of their results. Since proof checking generally has lower complexity than solving, small and trusted checkers can verify solver results in an scalable manner. Despite the successful adoption

---

of this approach by several SMT solvers [7,13,15,24,37], no standard SMT proof format has emerged, with each system using their own format and independent toolchain. The Alethe[1] format [35] for SMT proofs however can be emitted by the veriT solver for several years [10] and recently[2] also by the cvc5 solver [7]. Moreover, Alethe proofs can be reconstructed within the proof assistants Coq [4, 16] and Isabelle/HOL [11,36], which allows leveraging solvers who support the format (namely veriT and CVC4, the latter via a translator [16]) for automatic theorem proving. In Isabelle/HOL in particular this integration has been very successful with the veriT solver, significantly increasing the success rate of the popular Sledgehammer tactic [36]. The format has been refined and extended through the years [6], being now mature and used by multiple systems, with support for core SMT theories, quantifiers, and pre-processing. It allows different levels of granularity, so that solvers can provide coarse-grained proofs (which are easier to produce), or take the effort to produce more detailed, fine-grained proofs (which are often easier to check). It provides a term language close to SMT-LIB [8], facilitating printing from solvers as well as validating the connection between proofs and the corresponding proved problems. An overview of the Alethe proof format is given in Section 2.

A significant drawback of the Alethe format, however, is that it does not have an independent proof checker. This makes it harder for solvers to adopt the format, since to test their proof production they must be directly integrated with the proof assistants with Alethe reconstructions available. Moreover, these reconstruction methods do not check whether proof steps comply to the format's semantics, but rather are used as hints for internal tactics. Finally, the reconstruction techniques struggle with scalability due to well-known performance issues in the proof assistants [12,36].

In this paper we introduce CARCARA[3] (Section 3), an independent proof checker for Alethe proofs, implemented in a high-performance programming language, Rust. CARCARA is open-source and available under the Apache 2.0 license. Proof checking (Section 3.1) is performed by a collection of modules specific for each rule being checked. The presence of coarse-grained steps in Alethe requires special handling in the checker to account for missing information, which are discussed in detail. CARCARA also provides proof elaboration methods (Section 3.2) for particularly impactful coarse-grained steps, so that they can be automatically translated, offline from the solver, into easier-to-check fine-grained steps. We evaluate (Section 4) CARCARA's proof checking on a large set of proofs generated by veriT from SMT-LIB problems, analyzing its performance and effectiveness. The same set of proofs is used to evaluate the proof elaboration methods, where we analyze how checking elaborated proofs compares with the

---

[1] The format was previously known as the "veriT format", but it has recently been renamed to reflect its independence from any individual solver.

[2] cvc5's support for Alethe is still experimental and is under active development. CARCARA can actually be instrumental for improving cvc5's support for Alethe.

[3] We follow on the bird theme of the "Alethe" name. Carcará is the Portuguese word for the crested caracara, a resourceful bird of prey native of South America.

originals. Our analysis shows that CARCARA has performant proof checking and can identify wrong proofs produced by veriT. It also shows that elaboration can in some cases generate proofs significantly easier to check than the original ones.

1.1 Related work

CARCARA is inspired by the highly-successful DRAT-trim [23] proof checker for SAT proofs, which has been instrumental to the extensive usage of proofs in toolchains involving SAT solvers. It has also provided a basis for numerous advances in SAT proofs, with new proof formats and new checking techniques. We see its performant proof checking and elaboration techniques as the key elements to its success, serving both as an independent checker and as a bridge between solvers and performance-critical checkers, such as proof assistants or certified checkers. Providing both these features is the main goal of CARCARA.

The checker for the Logical Framework with Side Conditions (LFSC) [37], an extension of Edinburgh's Logical Framework (LF) [22], written in C++, is also a stand-alone, non-certified, highly efficient proof checker. The logical framework, where new rules can be mechanized in a language understood by the checker, provides great flexibility, and LFSC has been successfully used as a proof format for CVC4 [28] and cvc5 [5]. Similarly, Dedukti [25] is an OCaml checker for the $\lambda\Pi$-calculus, another extension of LF, and has been applied to SMT proofs, including to Alethe[4]. However, we are not aware of any mature implementation for this end. Elaboration techniques have not been the focus in these tools. Another difference is that they are based on dependently-typed languages far-removed from SMT-LIB, and generating proofs from SMT solvers for them can be more challenging, as well as relating the resulting proofs to the original problems.

An independent checker has been proposed for SMT proofs [34] from the OpenSMT [26] solver. The checker targets problems with uninterpreted functions and linear arithmetic, but does not support quantifiers nor pre-processing. It leverages DRAT-trim for the propositional reasoning and employs Python components for checking the other parts of the proof. Different components can use different proof formats, and to the best of our knowledge no comprehensive specification of the overall format is available. Some SMT solvers, such as SMT-Interpol [24] and cvc5 [7], have internal checkers for their proofs. Since these are not independent from the solvers, they are incomparable to our approach.

2 The Alethe Proof Format

Alethe was originally designed [10] as a proof-assistant friendly, easy-to-produce proof format for SMT solvers. A clear specification of the rules in a reference document [2] is provided, facilitating reconstruction within proof assistants by avoiding ambiguous syntax or semantics. To facilitate proof production, Alethe uses a term language that directly extends SMT-LIB, thus not requiring solvers

---

[4] "Verine" library available at https://deducteam.github.io/data/libraries/verine.tar.gz

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 [7]. The set of rules in the format captures SMT solving (as generally performed by CDCL($\mathcal{T}$)-based SMT solvers [31]) for problems containing a mix of any of quantifiers, uninterpreted functions, and linear arithmetic, as well as multiple pre-processing techniques. As a testament of the format's success, it has been refined and extended throughout the years [6], and has been used as the basis for the integration, with the proof assistants Isabelle/HOL and Coq, of the SMT solvers veriT [6, 36], CVC4 [16] and cvc5 [5, Sec. 3].

Here we briefly overview the Alethe proof format. For the full description of its syntax and semantics please see [2]. We assume the reader is familiar with basic notions of many-sorted equational first-order logic [17]. 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. 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. The indexed steps induce a directed acyclic graph rooted on the step concluding $\bot$ and with the assumptions $\varphi_1, \ldots, \varphi_n$ as leaves. Steps represent inferences and abstractly have the form

$$c_1, \ldots, c_k \; \triangleright \; i. \; \psi_1, \ldots, \psi_l \; (\texttt{rule} \; p_1, \ldots, p_n) \, [a_1, \ldots, a_m]$$

where `rule` names the inference rule used in this step. Every step has an identifier i and concludes a clause, represented as a list of literals $\psi_1, \ldots, \psi_l$. The premises are identifiers $p_1, \ldots, p_n$ of previous steps or assumptions, and rule-dependent arguments are terms $a_1, \ldots, a_m$; steps may occur under a *context*, which is defined by bound variables or substitutions $c_1, \ldots, c_k$. 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. Besides arbitrary formulas, Alethe has support for contexts which put in scope bound variables and substitutions, which are useful for representing pre-processing techniques in the presence of binders [6], such as Skolemization, let elimination and alpha-conversion.

The structure of Alethe proofs is motivated by SMT solvers generally operating with a cooperation of a SAT solver and multiple engines to perform theory reasoning, deriving new facts and applying simplifications. The overall proof may be seen as a ground first-order resolution proof with theory lemmas justified by closed subproofs. Thus the emphasis on steps concluding clauses as term lists, which avoids ambiguity as to what clause a disjunction represents. An example is that whether a resolution step concluding the term $A \vee B$ corresponds to the clause $[A, B]$ or $[A \vee B]$ depends on the premises. The use of identifiers for steps allows representing proofs as directed acyclic graphs rather than trees. Similarly,

```
(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.

term sharing can be achieved via the SMT-LIB `:named` attribute or `define-fun` commands [8, Sec1 4.1.6], which both allow naming subterms. These measures are essential for compact representation of proofs, which can be prohibitively large otherwise. Explicitly providing the conclusion of proof steps aims to both facilitate proof checking (as it allows steps to be verified locally) and proof production, so coarse-grained rules that do not uniquely define their conclusions from premises and arguments can be effectively checked.

Example 1. Figure 1 shows an SMT-LIB problem and an Alethe proof of its unsatisfiability. Note that in Alethe's concrete syntax clauses are represented via the `cl` operator (the only exception are conclusions of `assume` commands, which are considered unit clauses) and the context is not explicitly put in the steps, but rather assumed for all steps under (potentially nested) anchors introducing its elements. For this proof to be valid, three conditions need to be met: each `assume` command must correspond to an `assert` command in the original problem, every `step` command must be valid according to the semantics of its rule, and the proof must end with a step that concludes the empty clause (`cl`). The proof satisfies the first condition, as the terms in the `assume` commands are precisely the asserted terms in the SMT problem. The third condition holds as `t5`, the last step, concludes the empty clause. For the second condition, step `t4` is a direct consequence of the equivalence in its premise, `t3`, so it remains to check step `t3`, which is derived from a subproof. The anchor for `t3` introduces a bound variable y and a substitution $\{x \mapsto y\}$. The steps in the subproof contain terms with this new variable and operate under this substitution. The rule `refl` models reflexivity modulo the cumulative, capture-avoiding substitution in the (potentially nested) context, and thus `t3.t1` holds since $x = y\{x \mapsto y\}$. Step `t3.t2` is regular congruence with the operator ">" and does not depend on the context. Finally, step `t3` holds because its subproof shows the equivalence of the

Rust API	CLI

Carcará

Parser	Elaborator

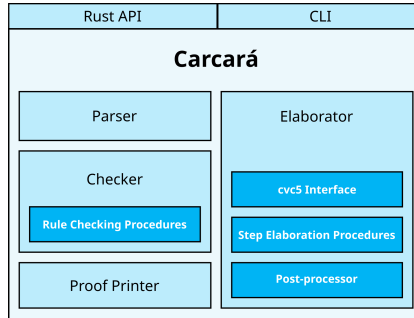| Checker	
Rule Checking Procedures	**cvc5 Interface**
	Step Elaboration Procedures
Proof Printer	**Post-processor**

Fig. 2: Overview of the architecture of Carcara.

bodies of the quantifiers under the renaming, introduced in the context, into a fresh variable relative to the left-hand side quantifier. Since all steps follow the expected semantics, all conditions are met and the proof is valid.

In the next section we show how Carcara checks the above conditions, highlighting some challenging rules and showing how some coarse-grained steps are elaborated into proofs potentially simpler to check.

3   Architecture and core components

Carcara is developed in the Rust programming language, and is publicly available[5] under the Apache 2.0 license. Its architecture is shown in Figure 2. It provides both a command line interface and bindings for a Rust API. The main component is the proof checking one, with 6.5k LOC, which is a collection of procedures for each rule to be checked (Section 3.1). The elaborator has 1k LOC and has an interface to the cvc5 solver, as well as a collection of elaboration methods and a post-processing module to knit together the elaborated proof (Section 3.2). The other components together have 6k LOC, including a handwritten 2k LOC SMT-LIB and Alethe parser, and an Alethe printer.

The inputs of Carcara are an SMT-LIB problem φ and an Alethe proof $\pi : \varphi \to \bot$. In proof-checking mode it checks each step in π with the respective procedure for its rule and prints either `valid`, when all steps are successfully checked and the proof concludes the empty clause (`cl`), `holey` when π is valid but contains steps that are not checked ("holes"), and `invalid` otherwise, together with an error message indicating the first step where checking failed and why. In proof-elaboration mode it converts π into $\pi' : \varphi \to \bot$, where some steps may be replaced by a series of steps elaborating them, and prints π'.

---

[5] https://github.com/ufmg-smite/carcara

3.1 Checking Alethe proofs

First the original SMT-LIB problem and its Alethe 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. Symbol definitions in the proof for term sharing are expanded during parsing. Terms are internally represented as directed acyclic graphs, using *hash consing* for maximal sharing and constant-time syntactically-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. Step identifiers are converted into indices for the arrays, so that access is constant-time.

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, Carcara has 90 rule checkers. We highlight below some of the rule checkers as well as some challenges for checking Alethe proofs and how we addressed them.

Term equality tests. Terms introduced by Alethe rules may have equality subterms implicitly reordered, but the rules are still valid if the conclusion changes only in this way. This flexibility is motivated by solvers often internally representing equalities ignoring order, which may lead to equalities being implicitly reordered when appearing in facts derived by these components. The congruence closure procedure [29] commonly used in SMT is an example of such a component. Since equality symmetry justifies these reorderings, but keeping track of all the changes can be challenging, the format allows them to be implicit.

As a consequence, syntactic equality cannot be the only test for whether two terms are the same. For example, the terms `(and p (= a b))` and `(and p (= b a))` may be required to be equal. Thus Carcara tests equality in two phases: first if they are syntactically equal, in which case they can be compared in constant time; otherwise they are simultaneously traversed and equality subterms in the same position are compared modulo equality reordering, failing as soon as subterms differ. We refer to this as a *polyequal* test. As we will see in Section 4.1, these tests can be a substantial portion of overall checking time in some cases.

Checking initial assumptions. The initial `assume` commands in an Alethe proof must correspond to assertions in the original problem, so their checker searches through the assertions to find a match. In general, this can be done efficiently: assertions are stored in a hash set during parsing, and these `assume` commands are valid if their conclusions occur in the set. However, `assume` commands are also impacted by implicit equality reordering, thus requiring polyequal tests. When an assumption does not occur in the assertions hash set, the checker attempts to match it to each assertion in turn, performing a polyequal test. As a consequence, when the original problem is large and the assertions similar and deep, checking `assume` steps may dominate overall checking time, as our experiments show (Section 4.1).

Checking contextual steps. Steps within subproofs may depend on their context to be valid, so before checking these steps, a context object is built based on the `anchor` opening the subproof. As shown in Section 2, context elements on which rules may depend are bound variables and substitutions. The former make new symbols available to build terms, while the latter allows steps to be valid modulo applying these substitutions.

Substitutions in Alethe are capture-avoiding, renaming bound variables during application, which facilitates producing proofs with binders [6]. However, it has the side effect of also preventing constant-time equality tests, since we must rather check α-equivalence, i.e., a term with bound variables may be required to be equal[6] to the result of applying a substitution that may have renamed some of these variables. To avoid spurious renaming when applying substitutions, the checker only renames bound variables which occur as free variables in the substitution range. Since computing free variables is itself costly, it is done lazily, only when the substitution is to be applied under a binder, and the result is cached.

Note that, as subproofs can be nested, the substitution in context for a step is the composition of a stack of substitutions $\sigma_1, \ldots, \sigma_n$. To avoid sequential application of substitutions, Alethe requires the substitution σ in context to be a cumulative substitution in which every term t in the range of the substitution σ_{i+1} is replaced by $t\sigma_i$. Thus σ can be applied simultaneously and correspond to a sequential application of $\sigma_1, \ldots, \sigma_n$. As a result of these requirements, handling and applying substitutions can be expensive in Alethe, as shown in Section 4.1.

Finally, the rules enclosing subproofs must be checked to whether their conclusions are valid from the introduced context and resulting subproof. For example, the `bind` rule in Example 1 requires that the bound variable in the quantifier at the right-hand side of the equality matches the range of the substitution put in context for its subproof. The `subproof` rule, which introduces local assumptions $a_1, \ldots, a_n$, and concludes a formula $\neg a'_1 \vee \cdots \neg a'_n \vee \varphi$, requires that the enclosed subproof derives φ and that each a_i match a'_i.

We now highlight coarse-grained rules whose checking is more intricate and expensive.

Resolution. The rule `resolution` in Alethe captures hyper-resolution on ground first-order clauses, i.e.,

$$\frac{C_1 \ \cdots \ C_n}{C} \ \texttt{resolution}, p_1, p_2, \ldots, p_{n-1}$$

where $C_1, \ldots, C_n$ are premises; p_i the pivot for the binary resolution between C_i and C_{i+1}, occurring as is in C_i and as $\neg p_i$ in C_{i+1}; and C the conclusion. While it is simple to check such steps, Alethe allows `resolution` steps to not provide the pivots, for the sake of facilitating proof-production in solvers. Checking such steps requires searching for the pivots and in which binary resolution they are to

---

[6] Since Alethe has bound-variable renaming rules, the checker requires names to be handled properly, rather than normalizing all binders internally via De Brujin indices.

be used, but Carcara applies an incomplete heuristic where pivots are inferred between the difference of literals in the premises and in the conclusion (i.e., literals not in the conclusion must have been pivots eventually eliminated). If that fails, we apply a reverse unit propagation (RUP) test [21], i.e., the step is valid if we can derive a conflict via Boolean Constraint Propagation from the premises and the negated conclusion. Note that Carcara also allows the pivots to be provided as arguments, in which case checking is simple, as expected.

AC simplification. Normalization modulo associativity and commutativity for conjunction and disjunction can be represented in Alethe via the `ac_simp` rule, which establishes the equality between a term t and a term t' that is t but with nested occurrences of these connectives flattened and duplicate arguments removed, until a fix-point. While this simplification is performance-critical [6, Sec. 4.6], checking the corresponding rule requires traversing t and performing the normalization, which is proportional to t's depth.

Arithmetic reasoning. Apart from simplification rules, arithmetic reasoning in Alethe is mainly captured by two rules: `la_generic` and `lia_generic`. Both rules conclude a clause of negated linear inequalities, which is valid due to the Farkas' lemma [18] guaranteeing that there exists a linear combination of these inequalities equivalent to $\bot$. The `la_generic` rule takes as arguments the coefficients of this linear combination, with which the rule can be checked by applying simple (but costly) operations on the coefficients to reduce the linear combination to $\bot$ (see [2, Sec 5.4, Rule 9] for the algorithm). The checker uses GMP [1] for efficiently performing the required computations with the coefficients.

While `la_generic` can be checked effectively, `lia_generic` cannot. It provides only the negated inequalities, which would require searching for the coefficients to perform the checking, essentially requiring the arithmetic solving to be repeated in the checker. As a consequence this rule is considered a hole and Carcara ignores it during proof checking, issuing a warning.

3.2   Elaborating Alethe proofs

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. Formally, a proof as the one below on the left, with a coarse step concluding ψ from premises $\psi_1, \ldots, \psi_n$, is elaborated into the proof on the right where the coarse step is replaced by a proof π, with *fine-grained* steps, rooted on ψ and with $\psi_1, \ldots, \psi_n$ as leaves:

$$\cfrac{\cfrac{\psi_1 \ \cdots \ \psi_n}{\psi} \text{ coarseStep} \qquad \cdots}{\Theta} \text{ rule} \qquad \Rightarrow_{\text{elab}} \qquad \cfrac{\cfrac{\cfrac{\psi_1 \ \cdots \ \psi_n}{\pi}}{\psi} \qquad \cdots}{\Theta} \text{ rule}$$

```
(step t2.t1 (cl (not (= a b)) (not (= b c)) (not (= c d)) (= a d))
  :rule eq_transitive)
(step t2.t2 (cl (not (= b a)) (= a b)) :rule eq_symmetric)
(step t2.t3 (cl (not (= c b)) (= b c)) :rule eq_symmetric)
(step t2.t4 (cl (not (= c d)) (= a d) (not (= b a)) (not (= c b)))
  :rule resolution :premises (t2.t1 t2.t2 t2.t3))
(step t2 (cl (not (= b a)) (not (= c d)) (not (= c b)) (= a d))
  :rule reordering :premises (t2.t4))
```

Fig. 3: Elaboration of an `eq_transitive` step. Note the new `eq_transitive` step is easy to check, and the new `t2` step has the same conclusion as the original.

Note the expansion only affects the proof locally, since any step using the conclusion of the coarse step as a premise may use the conclusion of π interchangeably.

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, and thus polyequal tests, which affects virtually every Alethe rule; and providing checkable justifications for `lia_generic` steps, to remove holes from proofs. Before detailing these methods, we illustrate the elaboration process with an example.

Elaborating transitivity steps. The `eq_transitive` rule concludes a valid clause composed of negated equalities followed by a single positive equality, such that the negated equalities form a transitive chain resulting in the final equality. However, the specification does not impose an order on the negated equalities (which can, remember, also be implicitly reordered). So the following step must also be valid, with a "shuffled" chain:

```
(step t2 (cl (not (= b a)) (not (= c d)) (not (= c b)) (= a d))
  :rule eq_transitive)
```

This permissive specification again facilitates proof production (particularly from congruence closure procedures), but requires the `eq_transitive` checker, for every link in the chain, to potentially traverse the whole clause searching for the next one, performing polyequal tests throughout. The goal of elaborating `eq_transitive` steps is that steps like `t2` are justified in a fine-grained manner. If we changed the conclusion of the step, this would impact the rest of the proof, if `t2` is used anywhere as a premise. We instead introduce a fine-grained proof for `t2`'s conclusion, as shown in Figure 3: an easy-to-check `eq_transitive` step (`t2.t1`), `eq_symmetric` steps to flip the equalities (`t2.t2`, `t2.t3`), `resolution` (`t2.t4`) and `reordering` (`t2.t5`) steps to derive the original conclusion.

Elaborating implicit equality reordering. Similarly to above, steps concluding a term t, with some subterm equality implicitly reordered, have their conclusion replaced by t' where that subterm is not reordered and a fine-grained proof of the conversion of t' into t is added. Figure 4 illustrates this process for an `assume`

```
(set-logic QF_UF)
(declare-const a Bool)
(declare-const b Bool)
(declare-const p Bool)
(assert (not (or p (= a b))))
(assert (or p (= b a)))
(check-sat)
```

Fig. 4a: An example SMT problem instance.

```
(assume h1 (not (or p (= a b))))
(assume h2 (or p (= a b)))
(step t3 (cl) :rule resolution
                :premises (h1 h2))
```

Fig. 4b: An Alethe proof for the SMT problem in Figure 4a. Notice that this proof makes use of implicit reordering of equalities in h2.

```
(assume h1 (not (or p (= a b))))
(assume h2 (or p (= b a)))
(step h2.t1 (cl (= (= b a) (= a b))) :rule equiv_simplify)
(step h2.t2 (cl (= (or p (= b a)) (or p (= a b))))
    :rule cong :premises (h2.t1))
(step h2.t3 (cl (not (or p (= b a))) (or p (= a b)))
    :rule equiv1 :premises (h2.t2))
(step h2.t4 (cl (or p (= a b))) :rule resolution :premises (h2 h2.t3))
(step t3 (cl) :rule resolution :premises (h1 h2.t4))
```

Fig. 4c: The elaborated proof without implicit equality reordering.

Fig. 4: An example of the elaboration to remove implicit equality reordering.

command, where note that step h2.t1 is the rewriting justifying the equality reordering of the subterm and the following steps rebuild the original conclusion.

In the original proof, the assume command h2 introduces the term (or p (= a b)), which is the original assertion (or p (= b a)) with the equality (= b a) implicitly reordered. In the elaborated proof (Figure 4c), the conclusion of h2 is replaced by one without implicit equality reordering, but step t3 expects the original conclusion. The steps h2.t1 to h2.t4 convert the new h2 conclusion into the original one, relying on standard equality reasoning and on resolution to connect the introduced steps. Notice that the t3 step, which originally refered to h2 as a premise, now refers to h2.t4.

When applied to every concluding terms with implicit equality reordering, the result of this elaboration method is a proof where equality tests are only syntactic, erasing the overhead of checking assumptions and polyequal tests.

Elaborating lia_generic steps. As discussed in Section 3.1, CARCARA considers lia_generic steps holes in the proof, as their checking is as hard as solving. Since our goal is to keep CARCARA as simple as possible, we rely on an external tool to elaborate the step by solving a problem corresponding to it in a proof-producing manner, then import the proof, checking it and guaranteeing that it is sound to replace the original step. Any tool producing detailed Alethe proofs for linear-integer arithmetic reasoning can be used to this end, but currently only cvc5 can do so [7]. We note that cvc5 currently has the limitation that its Alethe

proofs may contain rewrite steps not yet modeled in the Alethe simplification rules [2, Sec 5.11], and are thus not supported by CARCARA. They are considered holes, but since these are generally simple simplification rules, are much less harmful than `lia_generic` ones.

In detail, the elaboration method, when encountering a `lia_generic` step S concluding the negated inequalities $\neg l_1 \vee \cdots \vee \neg l_n$, generates an SMT-LIB problem asserting $l_1 \wedge \cdots \wedge l_n$ and invokes cvc5 on it, expecting an Alethe proof $\pi : (l_1 \wedge \cdots \wedge l_n) \rightarrow \bot$. CARCARA will check each step in π and, if they are not invalid, will replace step S in the original proof by a proof of the form:

```
(anchor :step S.t_m+1)
(assume S.h_1 l1)
...
(assume S.h_n ln)
...
(step S.t_m (cl false) :rule ...)
(step S.t_m+1 (cl (not l1) ... (not ln) false) :rule subproof)
(step S.t_m+2 (cl (not false)) :rule false)
(step S (cl (not l1) ... (not ln))
  :rule resolution :premises (S.t_m+1 S.t_m+2))
```

where steps `S.h_1` until `S.t_m` are imported from the cvc5 proof. As a result the `lia_generic` step `S` in the original proof will have been replaced by a detailed justification whose correctness can be independently established by CARCARA.

4   Evaluation

We evaluate CARCARA for proof-checking performance and the impact of elaboration methods. We use the veriT solver [13], version 2021.06-40-rmx, to generate Alethe proofs from all problems in the SMT-LIB benchmark library[7] whose logic it supports, with a 120 seconds timeout. We did not consider cvc5 as its support for Alethe is not yet as mature or complete. The veriT solver produced 39,229 proofs. They total 92gb, but vary greatly in size. The biggest proof has 4.5gb, fourteen have at least 1gb and over a hundred have more than 100mb, while almost 90% are under 1mb. All the experiments were run on a server equipped with AWS Graviton2 2.5 GHz ARM CPUs, with 4 GB of memory for each job.

4.1   Proof checking

We ran CARCARA on each proof until checking succeeded or failed. Only 378 had checking failures, which were due to incorrect[8] steps for quantifier simplifications (Skolemization and elimination of one-point quantifiers) and AC normalization. The issues have been communicated to the solver developers. For the successful proofs, a summary is given in Table 1, for each SMT-LIB logic, with the cumulative solving time by veriT and checking time by CARCARA.

---

[7] https://smtlib.cs.uiowa.edu/benchmarks.shtml
[8] In a superficial analysis the steps seemed sound, but the proofs were incorrect.

Logic	Problems	Solving time (s)	Checking time (s)	Ratio
AUFLIA	2135	1094.67	12.51	87.53
AUFLIRA	19200	248.95	144.03	1.73
UF	2885	2858.14	30.95	92.35
UFIDL	55	0.54	0.66	0.82
UFLIA	7221	3547.78	136.21	26.05
UFLRA	10	0.02	0.01	3.05
QF_ALIA	16	0.79	1.39	0.57
QF_AUFLIA	256	0.34	0.11	3.04
QF_IDL	609	3316.08	2240.10	1.48
QF_LIA	1018	5975.36	742.73	8.05
QF_LRA	537	3629.39	258.60	14.03
QF_RDL	81	620.46	123.14	5.04
QF_UF	4180	3857.34	1881.55	2.05
QF_UFIDL	66	396.74	87.58	4.53
QF_UFLIA	167	1194.51	4.70	254.41
QF_UFLRA	415	141.82	65.14	2.18
Total:	38851	26882.93	5729.39	4.69

Table 1: Total solving and proof-checking time per logic for veriT and CARCARA.

As expected, the comparison is heavily logic-dependent. In quantified logics (top of the table), checking is generally significantly cheaper than solving. An outlier is AUFLIRA, which is explained by the problems to which veriT could produce proofs being all both simple to solve and check. In logics such as QF_UF and QF_IDL, which can have very large proofs, overall checking time is comparable to solving time, if still noticeably smaller in total.

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.

We also evaluate the per-rule frequency, as shown in Figure 5b, and checking time, with Figure 6a showing the cumulative checking times and Figure 5a a box plot considering individual rule checks. The lower whisker represents the 5th percentile, the lower bound of the box represents the first quartile, the line inside the box represents the median, the upper bound of the box represents the third quartile, and the upper whisker represents the 95th percentile[9]. Rules that are rare and have negligible checking time are omitted. The data is gathered from proof checking in all proofs, even those that failed.

The `assume` commands account for a large proportion of the total time. This is justified by their checking, due to implicit equality reordering, being potentially proportional to both the quantity and the depth of assertions in the original problems. The box plot shows that the worse cases lead to the most expensive rule checks among all rules.

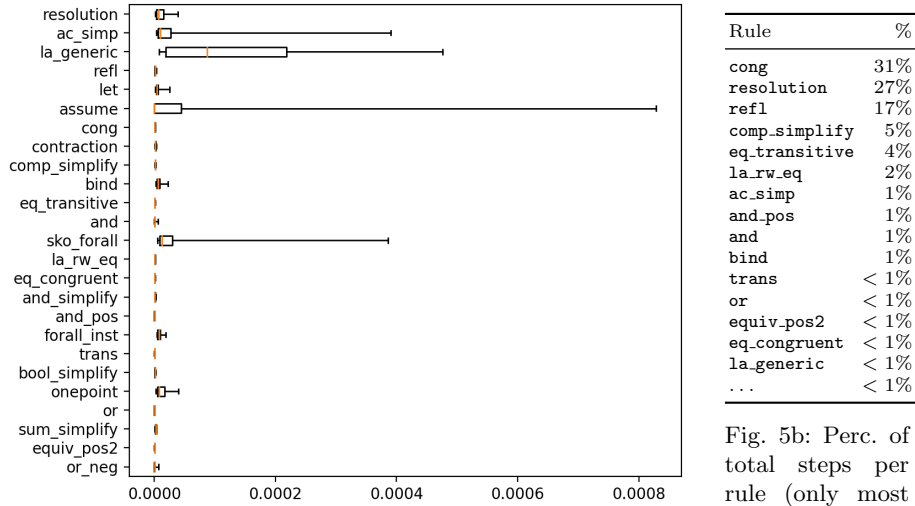[9] The plots follow the same criteria of the evaluation in [36].

Fig. 5a: Box plot for checking time per rule.

Rule	%
cong	31%
resolution	27%
refl	17%
comp_simplify	5%
eq_transitive	4%
la_rw_eq	2%
ac_simp	1%
and_pos	1%
and	1%
bind	1%
trans	< 1%
or	< 1%
equiv_pos2	< 1%
eq_congruent	< 1%
la_generic	< 1%
...	< 1%

Fig. 5b: Perc. of total steps per rule (only most frequent shown).

Rules with highest overall time are `resolution`, `ac_simp` and `la_generic`. For `resolution` this is explained mainly by its high frequency (this is similarly the case for `cong`), as well as by some more expensive checks (veriT does not provide pivots), as shown in the box plot. As for `ac_simp` and `la_generic`, while they are much less frequent, their checking is expensive (Section 3.1).

Other expensive rules to note are those related to contexts involving substitutions[10], specially `let`, for let elimination, and `refl`. It is common for `let` subproofs to be deeply nested, leading to large cumulative substitutions needing to be computed. As for `refl`, besides being one of the most frequent rules, about a third of its total time is spent on polyequal tests, and most of the rest is related to handling and applying substitutions, as well as checking alpha-equivalence.

4.2   Proof elaboration

We ran CARCARA, on each successfully checked proof, in proof-elaboration mode with the elaboration of transitivity steps and, more importantly, the removal of implicit equality reordering. On average, excluding parsing, elaboration takes 40% of the time required for checking. We focus on the impact on proof checking of the result of elaboration.

In Figure 7 we have the comparison, per proof, of the proof-checking time on the original proof and on the elaborated one (excluding parsing time). There is not a clear winner, but note that for harder proofs (those originally requiring at least 1s), checking the elaborated proof is often significantly faster. A per-rule analysis is shown in Figure 6b, with the proportion of the checking time spent

---

[10] The ones shown in the plots are `let`, `bind`, `sko_forall`, and `onepoint`.
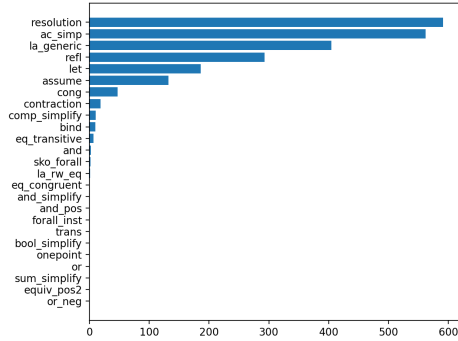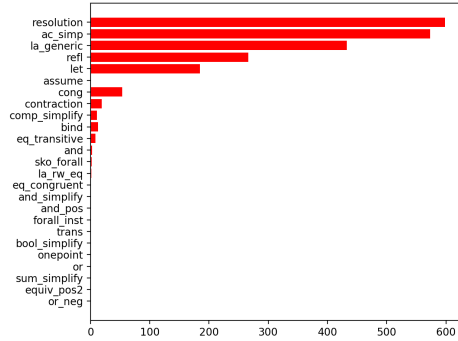
Fig. 6a: Total checking time per rule.

Fig. 6b: Times after elaboration.

in each rule, for the elaborated proofs. Comparing to Figure 6a, the checking time for `assume` steps becomes negligible in the elaborated proofs, as checking them now amounts to checking occurrence in a hash set. The overall time for `refl` also decreases, but only by 10%. This can be explained by the `refl` steps added during elaboration. While checking each `refl` is now potentially cheaper, this is offset by their increased number. Note that these additions also impact other rules, specially `cong`, whose cumulative time increased by 13%. Overall, proof elaboration resulted in a net improvement in checking time of 6%. Parsing time, however, increased, which made the overall runtime for proof-checking the original proofs virtually the same as for the elaborated proofs.

The results indicate that elaborating implicit equality reordering is not always worth it, specially for high-performant tools. However, it successfully yields proofs not requiring polyequal tests, which may help performance in other scenarios. For example, the reconstruction of Alethe proofs in Isabelle/HOL requires equality tests to be done by applying a normalizer to both terms and then testing them for syntactic equality. This leads to performance issues for reconstructing some rules [36], which this elaboration method would avoid.
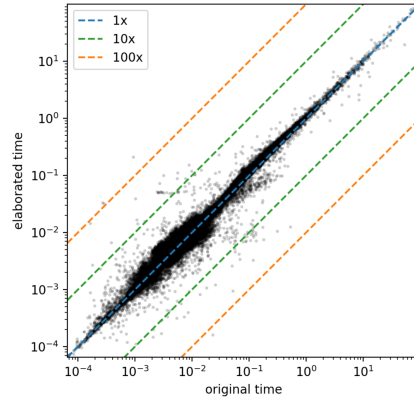

Fig. 7: Before vs after elaboration.

Elaborating `lia_generic` *steps.* In our benchmark set, 276 proofs contain a total of 127k `lia_generic` steps. As a proof of concept we instrumented CARCARA to apply the elaboration method described in Section 3.2 via a connection with cvc5[11]. Due to the still experimental Alethe proof production in cvc5, we only considered SMT problems derived

[11] cvc5-1.0.2, modified for better Alethe support, provided by the cvc5 team.

from `lia_generic` steps in proofs for the QF_UFLIA and QF_LIA logics. This excluded only 15 proofs, each containing exactly one `lia_generic` step. We ran CARCARA on proof-elaboration mode with a 30 minute timeout for each proof. For each `lia_generic` step, cvc5 was invoked with a 30s timeout and the resulting Alethe proof, if any, replaced the original `lia_generic` step, as described in Section 3.2.

Of the 261 proofs, CARCARA timed out on only 13 of them. Of the remaining 248 proofs, 82 still contained `lia_generic` steps after elaboration, either because cvc5 timed out when solving the generated problem, or because the cvc5 proofs contained `lia_generic` steps of their own. Note however that they are still improvements over the original `lia_generic` steps, since generally less inequalities are involved and the steps are potentially simpler to solve, were the process to be repeated. Similarly, although all elaborated proofs contained holes from cvc5 rewriting steps, these are much simpler than the original `lia_generic` ones.

As with the elaboration of implicit equality reordering, this elaboration method would be particularly impactful in scenarios such as Alethe reconstruction in Isabelle/HOL. Steps such as `lia_generic` are reconstructed via limited internal automation for arithmetic reasoning, which is known to fail [36, Sec. 4.3].

5   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. Extending CARCARA to convert Alethe proofs into other formats would also allow the elaboration techniques to benefit other toolchains.

As future work, we will add support for parallel proof checking, since steps in the same context can be checked completely independently. We will also add new elaboration methods for `resolution` and `ac_simp`, which occasionally are bottlenecks, and will provide elaboration for rewrite rules, which can change significantly between different solvers, complicating proof-production if solvers have to phrase their rewrites with a fixed set of rules. An automatic conversion into a defined set of rewrite rules, as described in [32], would address this issue.

Finally, we expect CARCARA to facilitate improving how we use Alethe proofs. For example, our large-scale evaluation shows the significant time spent on contextual substitutions, which is mainly due to the Alethe requirement of only applying substitutions simultaneously. Extending the proof format to allow other substitution application strategies may be beneficial for different scenarios, as proof production in some solvers has indicated [7, Sec 5.1]. In general, extensions to the format (for example, to other logical theories) can be done in a more informed way with the help of an independent checker.

Data Availability Statement. The datasets generated and analyzed during the current study are available in the Zenodo repository: https://zenodo.org/record/7574451 [3].

References

1. GNU Multiple Precision Arithmetic Library. http://gmplib.org/, Oct 2022.
2. The Alethe Proof Format: A Speculative Specification and Reference. https://verit.loria.fr/documentation/alethe-spec.pdf, Oct 2022.
3. Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. Carcara artifact, 2023. figshare, https://doi.org/10.5281/zenodo.7574451.
4. 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 Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.
5. Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.
6. Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury, and Pascal Fontaine. Scalable fine-grained proofs for formula processing. *Journal of Automated Reasoning*, 64(3):485–510, 2020.
7. Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark W. Barrett. Flexible proof production in an industrial-strength SMT solver. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2022.
8. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
9. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.
10. Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for SMT: a proposal. In *Workshop on Proof eXchange for Theorem Proving (PxTP)*, 2011.
11. Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.
12. Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. Reconstruction of z3's bit-vector proofs in HOL4 and isabelle/hol. In Jean-Pierre Jouannaud

and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2011.

13. Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In Renate A. Schmidt, editor, *Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

14. Lilian Burdy and David Déharbe. Teaching an old dog new tricks - the drudges of the interactive prover in atelier B. In Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings*, volume 10817 of *Lecture Notes in Computer Science*, pages 415–419. Springer, 2018.

15. Leonardo Mendonça de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) Workshops*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

16. Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. Smtcoq: A plug-in for integrating SMT solvers into coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.

17. Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 2 edition, 2001.

18. G. Farkas. A Fourier-féle mechanikai elv alkamazásai. *Mathematikaiés Természettudományi Értesítö*, 12:457–472, 1894. reference from Schrijver's Combinatorial Optimization textbook (Hungarian).

19. Mathias Fleury. Optimizing a verified SAT solver. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2019.

20. Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using imperative HOL. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 158–171. ACM, 2018.

21. Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008.

22. Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

23. Marijn J. H. Heule. The DRAT format and drat-trim checker. *CoRR*, abs/1610.06229, 2016.

24. Jochen Hoenicke and Tanja Schindler. A simple proof format for SMT. In David Déharbe and Antti E. J. Hyvärinen, editors, *International Workshop on Satisfiability Modulo Theories (SMT)*, volume 3185 of *CEUR Workshop Proceedings*, pages 54–70. CEUR-WS.org, 2022.

25. Gabriel Hondet and Frédéric Blanqui. The new rewriting engine of dedukti (system description). In Zena M. Ariola, editor, *International Conference on Formal*

Structures for Computation and Deduction (FSCD), volume 167 of *LIPIcs*, pages 35:1–35:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

26. Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina. Opensmt2: An SMT solver for multi-core and cloud computing. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *Lecture Notes in Computer Science*, pages 547–553. Springer, 2016.

27. Shuanglong Kan, Anthony Widjaja Lin, Philipp Rümmer, and Micha Schrader. Certistr: a certified string solver. In Andrei Popescu and Steve Zdancewic, editors, *Certified Programs and Proofs (CPP)*, pages 210–224. ACM, 2022.

28. Guy Katz, Clark W. Barrett, Cesare Tinelli, Andrew Reynolds, and Liana Hadarean. Lazy proofs for dpll(t)-based SMT solvers. In Ruzica Piskac and Muralidhar Talupur, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, pages 93–100. IEEE, 2016.

29. Greg Nelson and Derek C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, 1980.

30. Aina Niemetz, Mathias Preiner, and Clark W. Barrett. Murxla: A modular and highly extensible API fuzzer for SMT solvers. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification (CAV), Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2022.

31. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006.

32. Andres Nötzli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. Reconstructing fine-grained proofs of complex rewrites using a domain-specific language. In Alberto Griggio and Neha Rungta, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, 2022. To appear.

33. Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. versat: A verified modern SAT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7148 of *Lecture Notes in Computer Science*, pages 363–378. Springer, 2012.

34. Rodrigo Otoni, Martin Blicha, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. Theory-specific proof steps witnessing correctness of SMT executions. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 541–546. IEEE, 2021.

35. Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In Chantal Keller and Mathias Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 49–54, 2021.

36. Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In André Platzer and Geoff Sutcliffe, editors, *Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 450–467. Springer, 2021.

37. Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

38. Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In Alastair F. Donaldson and Emina Torlak, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 718–730. ACM, 2020.