



Flexible Proof Production in an Industrial-Strength SMT Solver

Haniel Barbosa¹, Andrew Reynolds², Gereon Kremer³, Hanna Lachnitt³,
Aina Niemetz³, Andres Nötzli³, Alex Ozdemir³, Mathias Preiner³,
Arjun Viswanathan², Scott Viteri³, Yoni Zohar⁴(✉), Cesare Tinelli²,
and Clark Barrett³

¹ Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

² The University of Iowa, Iowa City, USA

³ Stanford University, Stanford, USA

⁴ Bar-Ilan University, Ramat Gan, Israel

yoniz206@gmail.com

Abstract. Proof production for SMT solvers is paramount to ensure their correctness independently from implementations, which are often prohibitively difficult to verify. Historically, however, SMT proof production has struggled with performance and coverage issues, resulting in the disabling of many crucial solving techniques and in coarse-grained (and thus hard to check) proofs. We present a flexible proof-production architecture designed to handle the complexity of versatile, industrial-strength SMT solvers and show how we leverage it to produce detailed proofs, including for components previously unsupported by any solver. The architecture allows proofs to be produced modularly, lazily, and with numerous safeguards for correctness. This architecture has been implemented in the state-of-the-art SMT solver `cvc5`. We evaluate its proofs for SMT-LIB benchmarks and show that the new architecture produces better coverage than previous approaches, has acceptable performance overhead, and supports detailed proofs for most solving components.

1 Introduction

SMT solvers [9] are widely used as backbones of formal methods tools in a variety of applications, often safety-critical ones. These tools rely on the solver's correctness to guarantee the validity of their results such as, for instance, that an access policy does not inadvertently give access to sensitive data [4]. However, SMT solvers, particularly industrial-strength ones, are often extremely complex pieces of engineering. This makes it hard to ensure that implementation issues do not affect results. As the industrial use of SMT solvers increases, it is paramount to be able to convince non-experts of the trustworthiness of their results.

A solution is to decouple confidence from the implementation by coupling results with machine-checkable certificates of their correctness. For SMT solvers,

This work was partially supported by the Office of Naval Research (Contract No. 68335-17-C-0558), a gift from Amazon Web Services, and by NSF-BSF grant numbers 2110397 (NSF) and 2020704 (BSF).

© The Author(s) 2022

J. Blanchette et al. (Eds.): IJCAR 2022, LNAI 13385, pp. 15–35, 2022.

https://doi.org/10.1007/978-3-031-10769-6_3

this amounts to providing proofs of unsatisfiability. The main challenges are justifying a combination of theory-specific algorithms while keeping the solver performant and providing enough details to allow *scalable* proof checking, i.e., checking that is fundamentally simpler than solving. Moreover, while proof production is well understood for propositional reasoning and common theories, that is not the case for more expressive theories, such as the theory of strings, or for more advanced solver operations such as formula preprocessing.

We present a new, flexible proof-production architecture for versatile, industrial-strength SMT solvers and discuss its integration into the `cvc5` solver [5]. The architecture (Sect. 2) aims to facilitate the implementation effort via modular proof production and internal proof checking, so that more critical components can be enabled when generating proofs. We provide some details on the core proof calculus and how proofs are produced (Sect. 3), in particular how we support eager and lazy proof production with built-in proof reconstruction (Sect. 3.2). This feature is particularly important for substitution and rewriting techniques, facilitating the instrumentation of notoriously challenging functionalities, such as simplification under global assumptions [6, Section 6.1] and string solving [40, 46, 48], to produce detailed proofs. Finally, we describe (Sect. 5) how the architecture is leveraged to produce detailed proofs for most of the theory reasoning, critical preprocessing, and underlying SAT solving of `cvc5`. We evaluate proof production in `cvc5` (Sect. 6) by measuring the proof overhead and the proof quality over an extensive set of benchmarks from SMT-LIB [8].

In summary, *our contributions* are a flexible proof-producing architecture for state-of-the-art SMT solvers, its implementation in `cvc5`, the production of detailed proofs for simplification under global assumptions and the full theory of strings, and initial experimental evidence that proof-production overhead is acceptable and detailed proofs can be generated for a majority of the problems.

Preliminaries. We assume the usual notions and terminology of many-sorted first-order logic with equality (\approx) [29]. We consider signatures Σ all containing the distinguished Boolean sort `Bool`. We adopt the usual definitions of well-sorted Σ -terms, with literals and formulas as terms of sort `Bool`, and Σ -interpretations. A Σ -theory is a pair $T = (\Sigma, \mathbf{I})$ where \mathbf{I} , the *models* of T , is a class of Σ -interpretations closed under variable reassignment. A Σ -formula φ is *T-valid* (resp., *T-unsatisfiable*) if it is satisfied by all (resp., no) interpretations in \mathbf{I} . Two Σ -terms s and t of the same sort are *T-equivalent* if $s \approx t$ is *T-valid*. We write \vec{a} to denote a tuple (a_1, \dots, a_n) of elements, with $n \geq 0$. Depending on context, we will abuse this notation and also denote the set of the tuple's elements or, in case of formulas, their conjunction. Similarly, for term tuples \vec{s}, \vec{t} of the same length and sort, we will write $\vec{s} \approx \vec{t}$ to denote the conjunction of equalities between their respective elements.

2 Proof-Production Architecture

Our proof-production architecture is intertwined with the CDCL(\mathcal{T}) architecture [43], as shown in Fig. 1. Proofs are produced and stored modularly by each solving component, which also checks they meet the expected proof structure

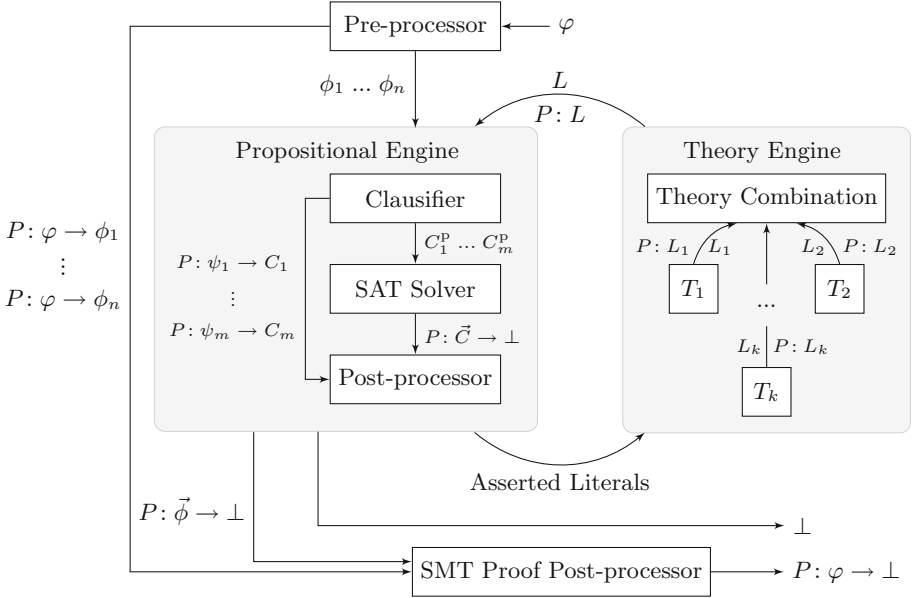


Fig. 1. Flexible proof-production architecture for CDCL(\mathcal{T})-based SMT solvers. In the above, $\psi_i \in \{\vec{\phi}, \vec{L}\}$ for each i , with ψ_i not necessarily distinct from ψ_{i+1} .

for that component, as described below. Proofs are combined only when needed, via post-processing. The *pre-processor* receives an input formula φ and simplifies it in a variety of ways into formulas ϕ_1, \dots, ϕ_n . For each ϕ_i , the pre-processor stores a proof $P: \varphi \rightarrow \phi_i$ justifying its derivation from φ .

The *propositional engine* receives the preprocessed formulas, and its *clausifier* converts them into a conjunctive normal form $C_1 \wedge \dots \wedge C_l$. A proof $P: \psi \rightarrow C_i$ is stored for each clause C_i , where ψ is a preprocessed formula. Note that several clauses may derive from each formula. Corresponding propositional clauses C_1^p, \dots, C_l^p , where first-order atoms are abstracted as Boolean variables, are sent to the SAT solver, which checks their joint satisfiability. The propositional engine enters a loop with the *theory engine*, which considers a set of literals asserted by the SAT solver (corresponding to a model of the propositional clauses) and verifies its satisfiability modulo a *combination of theories* T . If the set is T -unsatisfiable, a lemma L is sent to the propositional engine together with its proof $P: L$. Note that since lemmas are T -valid, their proofs have no assumptions. The propositional engine stores these proofs and clausifies the lemmas, keeping the respective clausification proofs in the clausifier. The clausified and abstracted lemmas are sent to the SAT solver to block the current model and cause the assertion of a different set of literals, if possible. If no new set is asserted, then all the clauses C_1, \dots, C_m generated until then are jointly unsatisfiable, and the SAT solver yields a proof $P: C_1 \wedge \dots \wedge C_m \rightarrow \perp$. Note that the proof is in terms of the first-order clauses, as are the derivation rules that

conclude \perp from them. The propositional abstraction does not need to be represented in the proof.

The post-processor of the propositional engine connects the assumptions of the SAT solver proof with the clausifier proofs, building a proof $P : \phi_1 \wedge \dots \wedge \phi_n \rightarrow \perp$. Since theory lemmas are T -valid, the resulting proof only has preprocessed formulas as assumptions. The final proof is built by the SMT solver’s post-processor combining this proof with the preprocessing proofs $P : \varphi \rightarrow \phi_i$. The resulting proof $P : \varphi \rightarrow \perp$ justifies the T -unsatisfiability of the input formula.

3 The Internal Proof Calculus

In this section, we specify how proofs are represented in the internal calculus of `cvc5`. We also provide some low-level details on how proofs are constructed and managed in our implementation.

The proof rules of the internal calculus are similar to rules in other calculi for ground first-order formulas, except that they are made a little more operational by optionally having *argument* terms and *side conditions*. Each rule has the form

$$r \frac{\varphi_1 \cdots \varphi_n}{\psi} \quad \text{or} \quad r \frac{\varphi_1 \cdots \varphi_n \mid t_1, \dots, t_m}{\psi} \text{ if } C$$

with *identifier* r , *premises* $\varphi_1, \dots, \varphi_n$, *arguments* t_1, \dots, t_m , *conclusion* ψ , and *side condition* C . The argument terms are used to construct the conclusion from the premises and can be used in the side condition together with the premises.

3.1 Proof Checkers and Proofs

The semantics of each proof rule r is provided operationally in terms of a *proof-rule checker* for r . This is a procedure that takes as input a list of argument terms \vec{t} and a list of premises $\vec{\varphi}$ for r . It returns `fail` if the input is malformed, i.e., it does not match the rule’s arguments and premises or does not satisfy the side condition. Otherwise, it returns a conclusion formula ψ expressing the result of applying the rule. All proof rules of the internal calculus have an associated proof-rule checker. We say that a proof rule *proves* a formula ψ , from given arguments and premises, if its checker returns ψ .

`cvc5` has an internal proof checker built modularly out of the individual proof-rule checkers. This checker is meant mostly for internal debugging during development, to help guarantee that the constructed proofs are correct. The expectation is that users will rely instead on third-party tools to check the proof certificates emitted by the solver.

A proof object is constructed internally using a data structure that we will describe abstractly here and call a *proof node*. This is a triple (r, \vec{N}, \vec{t}) consisting of a rule identifier r ; a sequence \vec{N} of proof nodes, its *children*; and a sequence \vec{t} of terms, its *arguments*. The relationships between proof nodes and their children induces a directed graph over proof nodes, with edges from proofs nodes to their children. We call a single-root graph rooted at node N a *proof*. A proof P is

$$\begin{array}{c}
\text{refl} \frac{- \mid t}{t \approx t} \quad \text{trans} \frac{r \approx s \quad s \approx t}{r \approx t} \quad \text{cong} \frac{\vec{s} \approx \vec{t} \mid f}{f(\vec{s}) \approx f(\vec{t})} \text{ if } f(\vec{s}) \text{ is well sorted} \\
\text{symm} \frac{s \approx t}{t \approx s} \quad \text{sr} \frac{\varphi \quad \vec{\varphi} \mid \mathcal{S}, \mathcal{R}, \mathcal{D}, \psi}{\psi} \text{ if } \mathcal{S}(\varphi, \mathcal{D}(\vec{\varphi})) \uparrow \downarrow_{\mathcal{R}} = \mathcal{S}(\psi, \mathcal{D}(\vec{\varphi})) \uparrow \downarrow_{\mathcal{R}} \\
\text{eq_res} \frac{\varphi \quad \varphi \approx \psi}{\psi} \quad \text{atom_rewrite} \frac{- \mid \mathcal{R}, s}{s \approx t} \text{ if } s \downarrow_{\mathcal{R}} = t \quad \text{witness} \frac{- \mid k}{k \approx k \uparrow} \\
\text{assume} \frac{- \mid \varphi}{\varphi} \quad \text{scope} \frac{\varphi \mid \varphi_1, \dots, \varphi_n}{\varphi_1 \wedge \dots \wedge \varphi_n \Rightarrow \varphi}
\end{array}$$

Fig. 2. Core proof rules of the internal calculus.

well-formed if it is finite, acyclic, and there is a total mapping Ψ from the nodes of P to formulas such that, for each node $N = (r, (N_1, \dots, N_m), \vec{t})$, $\Psi(N)$ is the formula returned by the proof checker for rule r when given premises $\Psi(N_1), \dots, \Psi(N_m)$ and arguments \vec{t} . For a well-formed proof P with root N and mapping Ψ , the *conclusion* of P is the formula $\Psi(N)$; a *subproof* of P is any proof rooted at a descendant of N in P . For convenience, we will identify a well-formed proof with its root node from now on.

3.2 Core Proof Rules

In total, the internal calculus of *cvc5* consists of 155 proof rules,¹ which cover all reasoning performed by the SMT solver, including theory-specific rules, rules for Boolean reasoning, and others. In the remainder of this section, we describe the *core* rules of the internal calculus, which are used throughout the system, and are illustrated in Fig. 2.

Proof Rules for Equality. Many theory solvers in *cvc5* perform theory-specific reasoning on top of basic equational reasoning. The latter is captured by the proof rules `eq_res`, `refl`, `symm`, `trans`, and `cong`. The first rule is used to prove a formula ψ from a formula φ that was proved equivalent to ψ . The rest are the standard rules for computing the congruence closure of a set of term equalities.

Proof Rules for Rewriting, Substitution and Witness Forms. A single *coarse-grained* rule, `sr`, is used for tracking justifications for core utilities in the SMT solver such as *rewriting* and *substitution*. This rule, together with other non-core rules with side conditions (omitted for brevity), allows the generation of coarse-grained proofs that trust the correctness of complex side conditions. Those conditions involve rewriting and substitution operations performed by *cvc5* during solving. More fine-grained proofs can be constructed from coarse-grained ones by justifying the various rewriting and substitution steps in terms of simpler proof rules. This is done with the aid of the equality rules mentioned above and the additional core rules `atom_rewrite` and `witness`. To describe `atom_rewrite`, `witness`, and `sr`, we first need to introduce some definitions and notations.

¹ See https://cvc5.github.io/docs/cvc5-1.0.0/proofs/proof_rules.html.

A *rewriter* \mathcal{R} is a function over terms that preserves equivalence in the background theory T , i.e., returns a term $t \downarrow_{\mathcal{R}}$ T -equivalent to its input t . We call $t \downarrow_{\mathcal{R}}$ the *rewritten* form of t with respect to \mathcal{R} . Currently, `cvc5` uses a handful of specialized rewriters for various purposes, such as evaluating constant terms, preprocessing input formulas, and normalizing terms during solving. Each individual rewrite step executed by a rewriter \mathcal{R} is justified in fine-grained proofs by an application of the rule `atom_rewrite`, which takes as argument both (an identifier for) \mathcal{R} and the term s the rewrite was applied to. Note that the rule’s soundness requires that the rewrite step be equivalence preserving.

A (*term*) *substitution* σ is a finite sequence $(t_1 \mapsto s_1, \dots, t_n \mapsto s_n)$ of oriented pairs of terms of the same sort. A *substitution method* \mathcal{S} is a function that takes a term r and a substitution σ and returns a new term that is the result of *applying* σ to r , according to some strategy. We write $\mathcal{S}(r, \sigma)$ to denote the resulting term. We distinguish three kinds of substitution methods for σ : *simultaneous*, which returns the term obtained by simultaneously replacing every occurrence of term t_i in r with s_i , for $i = 1, \dots, n$; *sequential*, which splits σ into n substitutions $(t_1 \mapsto s_1), \dots, (t_n \mapsto s_n)$ and applies them in sequence to r using the simultaneous strategy above; and *fixed-point*, which, starting with r , repeatedly applies σ with the simultaneous strategy until no further subterm replacements are possible. For example, consider the application $\mathcal{S}(y, (x \mapsto u, y \mapsto f(z), z \mapsto g(x)))$. The steps the substitution method takes in computing its result are the following: $y \rightsquigarrow f(z)$ if \mathcal{S} is simultaneous; $y \rightsquigarrow f(z) \rightsquigarrow f(g(x))$ if \mathcal{S} is sequential; $y \rightsquigarrow f(z) \rightsquigarrow f(g(x)) \rightsquigarrow f(g(u))$ if \mathcal{S} is fixed-point.

In `cvc5`, we use a *substitution derivation method* \mathcal{D} to derive a *contextual* substitution $(t_1 \mapsto s_1, \dots, t_n \mapsto s_n)$ from a collection $\vec{\varphi}$ of derived formulas. The substitution essentially orients a selection of term equalities $t_i \approx s_i$ entailed by $\vec{\varphi}$ and, as such, can be applied soundly to formulas derived from $\vec{\varphi}$.² We write $\mathcal{D}(\vec{\varphi})$ to denote the substitution computed by \mathcal{D} from $\vec{\varphi}$.

Finally, `cvc5` often introduces fresh variables, or *Skolem* variables, which are implicitly globally existentially quantified. This happens as a consequence of Skolemization of existential variables, lifting of if-then-else terms, and some kinds of flattening. Each Skolem variable k is associated with a term $k \uparrow$ of the same sort containing no Skolem variables, called its *witness term*. This global map from Skolem variables to their witness term allows `cvc5` to detect when two Skolem variables can be equated, as a consequence of their respective witness terms becoming equivalent in the current context [47]. Witness terms can also be used to eliminate Skolem variables at proof output time. We write $t \uparrow$ to denote the *witness form* of term t , which is obtained by replacing every Skolem variable in t by its witness term. For example, if k_1 and k_2 are Skolem variables with associated witness terms `ite(x ≈ z, y, z)` and $y - z$, respectively, and φ is the formula `ite(x ≈ k2, k1 ≈ y, k1 ≈ z)`, the witness form $\varphi \uparrow$ of φ is the formula `ite(x ≈ y - z, ite(x ≈ z, y, z) ≈ y, ite(x ≈ z, y, z) ≈ z)`. When a Skolem variable k

² Observe that substitutions are generated dynamically from the formulas being processed, whereas rewrite rules are hard-coded in `cvc5`’s rewriters.

appears in a proof, the **witness** proof rule is used to explicitly constrain its value to be the same as that of the term $k\uparrow$ it abstracts.³

We can now explain the **sr** proof rule, which is parameterized by a substitution method \mathcal{S} , a rewriter \mathcal{R} , and substitution derivation method \mathcal{D} . The rule is used to transform the proof of a formula φ into one of a formula ψ provided that the two formulas are equal up to rewriting under a substitution derived from the premises $\vec{\varphi}$. Note that this rule is quite general because its conclusion ψ , which is provided as an argument, can be any formula that satisfies the side condition.

Proof Rules for Scoped Reasoning. Two of the core proof rules, **assume** and **scope**, enable local reasoning. Together they achieve the effect of the \Rightarrow -introduction rule of Natural Deduction. However, separating the local assumption functionality in **assume** provides more flexibility. That rule has no premises and introduces a local assumption φ provided as an argument. The **scope** rule is used to *close the scope* of the local assumptions $\varphi_1, \dots, \varphi_n$ made to prove a formula φ , inferring the formula $\varphi_1 \wedge \dots \wedge \varphi_n \Rightarrow \varphi$.

We say that φ is a *free assumption* in proof P if P has a node (**assume**, $()$, φ) that is not a subproof of a **scope** node with φ as one of its arguments. A proof is *closed* if it has no free assumptions, and *open* otherwise.

Soundness. All proof rules other than **assume** are *sound* with respect to the background theory T in the following sense: if a rule proves a formula ψ from premises $\vec{\varphi}$, every model of T that satisfies $\vec{\varphi}$, and assigns the same values to Skolem variables and their respective witness term, satisfies ψ as well. Based on this and a simple structural induction argument, one can show that well-formed closed proofs have T -valid conclusions. In contrast, open proofs have conclusions that are T -valid only under assumptions. More precisely, in general, if $\vec{\varphi}$ are all the free assumptions of a well-formed proof P with conclusion ψ and \vec{k} are all the Skolem variables introduced in P , then $\vec{k} \approx \vec{k}\uparrow \wedge \vec{\varphi} \Rightarrow \psi$ is T -valid.

3.3 Constructing Proof Nodes

We have implemented a library of *proof generators* that encapsulates common patterns for constructing proof nodes. We assume a method `getProof` that takes the proof generator g and a formula φ as input and returns a proof node with conclusion φ based on the information in g . During solving, `cvc5` uses a combination of *eager* and *lazy* proof generation. In general terms, eager proof generation involves constructing proof nodes for inference steps at the time those steps are taken during solving. Eager proof generation may be required if the computation state pertinent to that inference cannot be easily recovered later. In contrast, lazy proof generation occurs for inferred formulas associated with proof generators that can do internal bookkeeping to be able to construct proof nodes for the formula *after* solving is completed. Depending on the formula, different kinds of proof generators are used. For brevity, we only describe in detail (see Sect. 3.2)

³ The proof rules that account for the introduction of Skolem variables in the first place are not part of the core set and so are not discussed here.

Algorithm 1. Proof generation for term-conversion generators, rewrite-once policy. B is a lazy proof builder, R a map from terms to their converted form, and $c_{\text{pre}}, c_{\text{post}}$ are sets of pairs of equalities and the proof generators justifying them.

```

getProof( $g, \varphi$ ) where  $g$  contains  $c_{\text{pre}}, c_{\text{post}}$  and  $\varphi$  is  $t_1 \approx t_2$ 
1:  $B := \emptyset, R := \emptyset$ 
2: getTermConv( $t_1, c_{\text{pre}}, c_{\text{post}}, B, R$ )
3: if  $R[t_1] \neq t_2$  then fail else return getProof( $B, t_1 \approx R[t_1]$ )

getTermConv( $s, c_{\text{pre}}, c_{\text{post}}, B, R$ ), where  $s = f(s_1, \dots, s_n)$ 
1: if  $s$  in dom( $R$ ) then return
2: if  $(s \approx s', g') \in c_{\text{pre}}$  for some  $s', g'$  then
3:    $R[s] := s', \text{addLazyStep}(B, s \approx s', g')$ 
4:   return
5: for  $1 \leq i \leq n$  do getTermConv( $s_i, c_{\text{pre}}, c_{\text{post}}, B, R$ )
6:  $R[s] := r$ , where  $r = f(R[s_1], \dots, R[s_n])$ 
7: if  $s \neq r$  then addStep( $B, \text{cong}, (s_1 \approx R[s_1], \dots, s_n \approx R[s_n]), f$ )
8: else addStep( $B, \text{rfl}, (), s \approx s$ )
9: if  $(r \approx r', g') \in c_{\text{post}}$  for some  $r', g'$  then
10:   $R[s] := r', \text{addLazyStep}(B, r \approx r', g'), \text{addStep}(B, \text{trans}, (s \approx r, r \approx r'), ())$ 

```

the proof generator most relevant to the core calculus, the *term-conversion proof generator*, targeted for substitution and rewriting proofs.

4 Proof Reconstruction for Substitution and Rewriting

Once it determines that the input formulas $\varphi_1, \dots, \varphi_n$ are jointly unsatisfiable, the SMT solver has a reference to a proof node P that concludes \perp from the free assumptions $\varphi_1, \dots, \varphi_n$. After the post-processor is run, the (closed) proof ($\text{scope}, P', (\varphi_1, \dots, \varphi_n)$) is then generated as the final proof for the user, where P' is the result of optionally expanding coarse-grained steps (in particular, applications of the rule sr) in P into fine-grained ones. To do so, we require the following algorithm for generating *term-conversion* proofs.

In particular, we focus on equalities $t \approx s$ whose proof can be justified by a set of steps that replace subterms of t until it is syntactically equal to s . We assume these steps are provided to a *term-conversion proof generator*. Formally, a term-conversion proof generator g is a pair of sets c_{pre} and c_{post} . The set c_{pre} (resp., c_{post}) contains pairs of the form $(t \approx s, g_{t,s})$ indicating that t should be replaced by s in a preorder (resp., postorder) traversal of the terms that g processes, where $g_{t,s}$ is a proof generator that can prove the equality $t \approx s$. We require that neither c_{pre} nor c_{post} contain multiple entries of the form $(t \approx s_1, g_1)$ and $(t \approx s_2, g_2)$ for distinct (s_1, g_1) and (s_2, g_2) .

The procedure for generating proofs from a term-conversion proof generator g is given in Algorithm 1. When asked to prove an equality $t_1 \approx t_2$, getProof traverses the structure of t_1 and applies steps from the sets c_{pre} and c_{post} from g .

The traversal is performed by the auxiliary procedure `getTermConv` which relies on two data structures. The first is a *lazy proof builder* B that stores the intermediate steps in the overall proof of $t_1 \approx t_2$. The proof builder is given these steps either via `addStep`, as a concrete triple with the proof rule, a list of premise formulas, and a list of argument terms, or as a *lazy step* via `addLazyStep`, with a formula and a reference to another generator that can prove that formula. The second data structure is a mapping R from terms to terms that is updated (using array syntax in the pseudo-code) as the converted form of terms is computed by `getTermConv`. For any term s , executing `getTermConv(s, cpre, cpost, B, R)` will result in $R[s]$ containing the converted form of s according to the rewrites in c_{pre} and c_{post} , and B storing a proof step for $s \approx R[s]$. Thus, the procedure `getProof` succeeds when, after invoking `getTermConv(t1, cpre, cpost, B, R)` with B and R initially empty, the mapping R contains t_2 as the converted form of t_1 . The proof for the equality $t_1 \approx R[t_1]$ can then be constructed by calling `getProof` on the lazy proof builder B , based on the (lazy) steps stored in it.

Each subterm s of t_1 is traversed only once by `getTermConv` by checking whether R already contains the converted form of s . When that is not the case, s is first preorder processed. If c_{pre} contains an entry indicating that s rewrites to s' , this rewrite step is added to the lazy proof builder and the converted form $R[s]$ of s is set to s' . Otherwise, the immediate subterms of s , if any, are traversed and then s is postorder processed. The converted form of s is set to some term r of the form $f(R[s_1], \dots, R[s_n])$, considering how its immediate subterms were converted. Note that B will contain steps for $\vec{s} \approx R[\vec{s}]$. Thus, the equality $s \approx r$ can be proven by congruence for function f with these premises if $s \neq r$, and by reflexivity otherwise. Furthermore, if c_{post} indicates that r rewrites to r' , then this step is added to the lazy proof builder; a transitivity step is added to prove $s \approx r'$ from $t \approx r$ and $r \approx r'$; and the converted form $R[s]$ is set to r' .

Example 1. Consider the equality $t \approx \perp$, where $t = f(b) + f(a) < f(a-0) + f(b)$, and suppose the conversion of t is justified by a term-conversion proof generator g containing the sets $c_{\text{pre}} = \{(f(b) + f(a) \approx f(a) + f(b), g^{\text{AC}}), (a - 0 \approx a, g_0^{\text{Arith}})\}$ and $c_{\text{post}} = \{(f(a) + f(b) < f(a) + f(b) \approx \perp, g_1^{\text{Arith}})\}$. The generator g^{AC} provides a proof based on associative and commutative reasoning, whereas g_0^{Arith} and g_1^{Arith} provide proofs based on arithmetic reasoning. Invoking `getProof(g, t ≈ ⊥)` initiates the traversal with `getTermConv(t, cpre, cpost, ∅, ∅)`. Since t is not in the conversion map, it is preorder processed. However, as it does not occur in c_{pre} , nothing is done and its subterms are traversed. The subterm $f(b) + f(a)$ is equated to $f(a) + f(b)$ in c_{pre} , justified by g^{AC} . Therefore R is updated with $R[f(b) + f(a)] = f(a) + f(b)$ and the respective lazy step is added to B . The subterms of $f(b) + f(a)$ are not traversed, therefore the next term to be traversed is $f(a-0) + f(b)$. Since it does not occur in c_{pre} , its subterm $f(a-0)$ is traversed, which analogously leads to the traversal of $a-0$. As $a-0$ does occur in c_{pre} , both R and B are updated accordingly and the processing of its parent $f(a-0)$ resumes. A congruence step added to B justifies its conversion to $f(a)$ being added to R .

No more additions happen since $f(a)$ does not occur in c_{post} . Analogously, R and B are updated with $f(b)$ not changing and $f(a - 0) + f(b)$ being converted into $f(a) + f(b)$. Finally, the processing returns to the initial term t , which has been converted to $R[f(b) + f(a)] < R[f(a + 0) + f(b)]$, i.e., $f(a) + f(b) < f(a) + f(b)$. Since this term is equated to \perp in c_{post} , justified by g_1^{Arith} , the respective lazy step is added to B , as well as a transitivity step to connect $f(b) + f(a) < f(a - 0) + f(b) \approx f(a) + f(b) < f(a) + f(b)$ and $f(a) + f(b) < f(a) + f(b) \approx \perp$. At this point, the execution terminates with $R[f(b) + f(a) < f(a + 0) + f(b)] = \perp$, as expected. A proof for $t \approx \perp$ with the following structure can then be extracted from B :

$$\begin{array}{l}
P_0 : \text{cong} \frac{\text{Lazy} \frac{g^{\text{AC}}}{f(b) + f(a) \approx f(a) + f(b)}}{f(b) + f(a) < f(a - 0) + f(b) \approx f(a) + f(b) < f(a) + f(b)} \quad P_1 \mid < \quad P_2 : \text{refl} \frac{\perp \mid f(b) \approx f(b)}{f(b) \approx f(b)} \\
\text{trans} \frac{P_0 \quad \text{Lazy} \frac{g_1^{\text{Arith}}}{f(a) + f(b) < f(a) + f(b) \approx \perp}}{f(b) + f(a) < f(a - 0) + f(b) \approx \perp} \quad P_1 : \text{cong} \frac{\text{Lazy} \frac{g_0^{\text{Arith}}}{a - 0 \approx a} \mid f}{f(a - 0) \approx f(a)} \quad P_2 \mid +
\end{array}$$

We use several extensions to the procedures in Algorithm 1. Notice that this procedure follows the policy that terms on the right-hand side of conversion steps (equalities from c_{pre} and c_{post}) are not traversed further. The procedure `getTermConv` is used by term-conversion proof generators that have the *rewrite-once* policy. A similar procedure which additionally traverses those terms is used by term-conversion proof generators that have a *rewrite-to-fixpoint* policy.

We now show how the term-conversion proof generator can be used for reconstructing fine-grained proofs from coarse-grained ones. In particular we focus on proofs P_{ψ_1} of the form $(\text{sr}, (Q_{\psi_0}, \vec{Q}), (\mathcal{S}, \mathcal{R}, \mathcal{D}, \psi))$. Recall from Fig. 2 that the proof rule `sr` concludes a formula ψ that can be shown equivalent to the formula ψ_0 proven by Q_{ψ_0} based on a substitution derived from the conclusions of the nodes \vec{Q} . A proof like P_{ψ_1} above can be transformed to one that involves (atomic) theory rewrites and equality rules only. We show this transformation in two phases. In the first phase, the proof is expanded to:

$$(\text{eq_res}, (Q_{\psi_0}, (\text{trans}, (R_0, (\text{symm}, R_1))))))$$

with $R_i = (\text{trans}, ((\text{subs}, \vec{Q}_{\vec{\varphi}}, (\mathcal{S}, \mathcal{D}, \psi_i)), (\text{rewrite}, (), (\mathcal{R}, \mathcal{S}(\psi_i, \mathcal{D}(\vec{\varphi}))))))$ for $i \in \{0, 1\}$ where $\vec{\varphi}$ are the conclusions of $\vec{Q}_{\vec{\varphi}}$, and `subs` and `rewrite` are auxiliary proof rules used for further expansion in the second phase. We describe them next.

Substitution Steps. Let $P_{t \approx s}$ be the subproof $(\text{subs}, \vec{Q}_{\vec{\varphi}}, (\mathcal{S}, \mathcal{D}, t))$ of R_i above proving $t \approx s$ with $s = \mathcal{S}(\psi_i, \mathcal{D}(\vec{\varphi}))$ and $\mathcal{D}(\vec{\varphi}) = (t_1 \mapsto s_1, \dots, t_n \mapsto s_n)$. Substitution steps can be expanded to fine-grained proofs using a term-conversion proof generator. First, for each $j = 1, \dots, n$, we construct a proof of $t_j \approx s_j$, which involves simple transformations on the proofs of $\vec{\varphi}$. Suppose we store all of these in an eager proof generator g . If \mathcal{S} is a simultaneous or fixed-point substitution, we then build a single term-conversion proof generator C , which

recall is modeled as a pair of mappings $(c_{\text{pre}}, c_{\text{post}})$. We add $(t_j \approx s_j, g)$ to c_{pre} for all j . We use the rewrite-once policy for C if \mathcal{S} is a simultaneous substitution, and the rewrite-fixed-point policy for C otherwise. We then replace the proof $P_{t \approx s}$ by $\text{getProof}(C, t \approx s)$, which runs the procedure in Algorithm 1. Otherwise, if \mathcal{S} is a sequential substitution, we construct a term-conversion generator C_j for each j , initializing it so that its c_{pre} set contains the single rewrite step $(t_j \approx s_j, g)$ and uses a rewrite-once policy. We then replace the proof $P_{t \approx s}$ by $(\text{trans}, (P_1, \dots, P_n))$ where, for $j = 1, \dots, n$: P_j is generated by $\text{getProof}(C_j, s_{j-1} \approx s_j)$; $s_0 = t$; s_i is the result of the substitution $\mathcal{D}(\vec{\varphi})$ after the first i steps; and $s_n = s$.

Rewrite Steps. Let P be the proof node $(\text{rewrite}, (), (\mathcal{R}, t))$, which proves the equality $t \approx t \downarrow_{\mathcal{R}}$. During reconstruction, we replace P with a proof involving only fine-grained rules, depending on the rewrite method \mathcal{R} . For example, if \mathcal{R} is the core rewriter, we run the rewriter again on t in proof tracking mode. Normally, the core rewriter performs a term traversal and applies atomic rewrites to completion. In proof tracking mode, it also return two lists, for pre- and post-rewrites, of steps $(t_1 \approx s_1, g), \dots, (t_n \approx s_n, g)$ where g is a proof generator that returns $(\text{atom_rewrite}, (), (\mathcal{R}, t_i))$ for all equalities $t_i \approx s_i$. Furthermore, for each Skolem k that is a subterm of t , we construct the rewrite steps $(k \approx k \uparrow, g')$ where g' is a proof generator that returns $(\text{witness}, (), (k))$ for equalities $k \approx k \uparrow$. We add these rewrite proof steps to a term-conversion generator C with rewrite-fixed-point policy, and replace P by $\text{getProof}(C, t \approx t \downarrow_{\mathcal{R}})$.

5 SMT Proofs

Here we briefly describe each component shown in Sect. 2 and how it produces proofs with the infrastructure from Sects. 3 and 3.2.

5.1 Preprocessing Proofs

The *pre-processor* transforms an input formula φ into a list of formulas to be given to the core solver. It applies a sequence of *preprocessing passes*. A pass may *replace* a formula φ_i with another one ϕ_i , in which case it is responsible for providing a proof of $\varphi_i \approx \phi_i$. It may also append a new formula ϕ to the list, in which case it is responsible for providing a proof for it. We use a (lazy) proof generator that tracks these proofs, maintaining the invariant that a proof can be provided for all (preprocessed) formulas when requested. We have instrumented proof production for the most common preprocessing passes, relying heavily on the *sr* rule to model transformations such as expansion of function definitions and, with witness forms, Skolemization and if-then-else elimination [6].

Simplification Under Global Assumptions. *cvc5* aggressively learns literals that hold globally by performing Boolean constraint propagation over the input formula. When a learned literal corresponds to a variable elimination (e.g., $x \approx 5$ corresponds to $x \mapsto 5$) or a constant propagation (e.g., $P(x)$ corresponds to

$P(x) \mapsto \top$), we apply the corresponding (term) substitution to the input. This application is justified via `sr`, while the derivation of the globally learned literals is justified via classification and resolution proofs, as explained in Sect. 5.3.

The key features of our architecture that make it feasible to produce proofs for this simplification are the automatic reconstruction of `sr` steps and the ability to customize the strategy for substitution application during reconstruction, as detailed in Sect. 3.2. When a new variable elimination $x \mapsto t$ is learned, old ones need to be normalized to eliminate any occurrences of x in their right-hand sides. Computing the appropriate simultaneous substitution for all eliminations requires quadratically many traversals over those terms. We have observed that the size of substitutions generated by this preprocessing pass can be very large (with thousands of entries), which makes this computation prohibitively expensive. Using the fixed-point strategy, however, the reconstruction for the `sr` steps can apply the substitution efficiently and its complexity depends on how many applications are necessary to reach a fix-point, which is often low in practice.

5.2 Theory Proofs

The theory engine produces lemmas, as disjunctions of literals, from an individual theory or a combination of them. In the first case, the lemma’s proof is provided directly by the corresponding theory solver. In the second case, a theory solver may produce a lemma ψ containing a literal ℓ derived by some other theory solver from literals $\bar{\ell}$. A lemma over the combined theory is generated by replacing ℓ in ψ by $\bar{\ell}$. This regression process, which is similar to the computation of *explanations* during solving, is repeated until the lemma contains only input literals. The proof of the final lemma then uses rules like `sr` to combine the proofs of the intermediate literals derived locally in various theories and their replacement by input literals in the final lemma.

Equality and Uninterpreted Function (EUF) Proofs. The EUF solver can be easily instrumented to produce proofs [31, 42] with equality rules (see Fig. 2). In `cvc5`, term equivalences are also derived via rewriting in some other theory T : when a function from T has all of its arguments inferred to be congruent to T -values, it may be rewritten into a T -value itself, and this equivalence asserted. Such equivalences are justified via `sr` steps. Since generating equality proofs incurs minimal overhead [42] and rewriting proofs are reconstructed lazily, EUF proofs are generated during solving and stored in an eager proof generator.

Extensional Arrays and Datatypes Proofs. While these two theories differ significantly, they both combine equality reasoning with rules for handling their particular operators. For arrays, these are rules for `select`, `store`, and array extensionality (see [36, Sec. 5]). For datatypes, they are rules reflecting the properties of *constructors* and *selectors*, as well as acyclicity. The justifications for lemmas are also generated eagerly and stored in an eager proof generator.

Bit-Vector Proofs. The bit-vector solver applies bit-blasting to reduce bit-vector problems to equisatisfiable propositional problems. Thus, its lemmas amount

to the rewriting of the bit-vector literals into Boolean formulas, which will be solved and proved by the propositional engine. The bit-vector lemmas are proven lazily, analogous to `sr` steps, with the difference that the reconstruction uses the bit-blaster in the bit-vector solver instead of the rewriter.

Arithmetic Proofs. The *linear* arithmetic solver is based on the simplex algorithm [24], and each of its lemmas is the negation of an unsatisfiable conjunction of inequalities. Farkas’ lemma [30, 49] guarantees that there exists a linear combination of these inequalities equivalent to \perp . The coefficients of the combination are computed during solving with minimal overhead [38], and the equivalence is proven with an `sr` step. To allow the rewriter to prove this equivalence, the bounds of the inequalities are scaled by constants and summed during reconstruction. Integer reasoning is proved through rules for branching and integer bound tightening, recorded eagerly.

Non-linear arithmetic lemmas are generated from incremental linearization [16] or cylindrical algebraic coverings [1]. The former can be proven via propositional and basic arithmetic rules, with only a few, such as the tangent plane lemma, needing a dedicated proof rule. The latter requires two complex rules that are not inherently simpler than solving, albeit not as complex as those for regular CAD-based theory solvers [2]. We point out that checking these rules would require a significant portion of CAD-related theory, whose proper formalization is still an open, if actively researched, problem [18, 25, 34, 41, 53].

Quantifier Proofs. Quantified formulas not Skolemized during pre-processing are handled via instantiation, which produces theory lemmas of the form $(\forall \vec{x} \varphi) \Rightarrow \varphi\sigma$, where σ is a grounding substitution. An instantiation rule proves them independently of how the substitution was actually derived, since any well-typed one suffices for soundness.

String Proofs. The strings solver applies a layered approach, distinguishing between core [40] and extended operators [48]. The core operators consist of (dis)equalities between string concatenations and length constraints. Reasoning over them is proved by a combination of equality and linear integer arithmetic proofs, as well as specific string rules. The extended operators are reduced to core ones via formulas with bounded quantifiers. The reductions are proven with rules defining each extended function’s semantics, and `sr` steps justifying the reductions. Finally, regular membership constraints are handled by string rules that unfold occurrences of the Kleene star operator and split up regular expression concatenations into different parts. Overall, the proofs for the strings theory solver encompass not only string-specific reasoning but also equality, linear integer arithmetic, and quantifier reasoning, as well as substitution and rewriting.

Unsupported. The theory solvers for the theories of floating-point arithmetic, sequences, sets and relations, and separation logic are currently not proof-producing in `cvc5`. These are relatively new or non-standard theories in SMT and have not been our focus, but we intend to produce proofs for them in the future.

Table 1. Cumulative solving times (s) on benchmarks solved by all configurations, with the slowdown versus CVC+S in parentheses.

| Logics | # | CVC+OS | CVC+S | CVC+SP | CVC+SPR |
|---------|---------|--------|-------|-------------|-------------|
| NON-BVs | 116,321 | 164k | 166k | 284k (1.7×) | 299k (1.8×) |
| BVs | 29,192 | 45k | 57k | 150k (2.6×) | 224k (3.9×) |

5.3 Propositional Proofs

Propositional proofs justify both the conversion of preprocessed input formulas and theory lemmas into conjunctive normal form (CNF) and the derivation of \perp from the resulting clauses. CNF proofs are a combination of Boolean transformations and introductions of Boolean formulas representing the definition of Tseytin variables, used to ensure that the CNF conversion is polynomial. The clausifier uses a lazy proof builder which stores the clausification steps eagerly, with the preprocessed input formulas as assumptions, and the theory lemmas as lazy steps, with associated proof generators.

For Boolean reasoning, `cvc5` uses a version of MiniSat [27] instrumented to produce resolution proofs. It uses a lazy proof builder to record resolution steps for learned clauses as they are derived (see [7, Chap 1] for more details) and to lazily build a refutation with only the resolution steps necessary for deriving \perp . The resolution rule, however, is ground first-order resolution, since the proofs are in terms of the first-order clauses rather than their propositional abstractions.

6 Evaluation

In this section, we discuss an initial evaluation of our implementation in `cvc5` of the proof-production architecture presented in this paper. In the following, we denote different configurations of `cvc5` by CVC plus some suffixes. A configuration using variable and clause elimination in the SAT solver [26], symmetry breaking [23] in the EUF solver, and black-box SAT solving in the bit-vector (BV) solver, is denoted by the suffix `O`. These techniques are currently incompatible with the proof production architecture. Other `cvc5` techniques for which we do not yet support fine-grained proofs, however, are active and have their inferences registered in the proofs as trusted steps. A configuration that includes simplification under global assumptions is denoted by `S`; one that includes producing proofs by `P`; and one that additionally reconstructs proofs by `R`. The default configuration of `cvc5` is `CVC+OS`.

We split our evaluation into measuring the proof-production cost as well as the performance impact of making key techniques proof-producing; the proof reconstruction overhead; and the coverage of the proof production. We also comment on how `cvc5`'s proofs compare with `CVC4`'s proofs. Note that the internal proof checking described in Sect. 3, which was invaluable for a correct implementation, is disabled for evaluating performance. Experiments ran on a cluster with

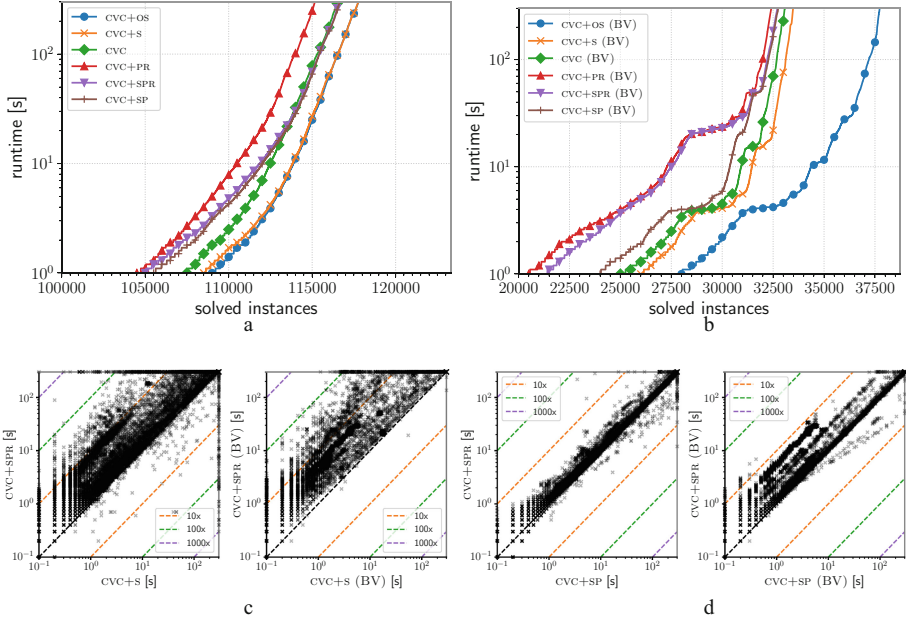


Fig. 3. (a) Cactus plot for NON-BVs (b) Cactus plot for BVs (c) Scatter plot of overall proof cost (d) Reconstruction cost

Intel Xeon E5-2620 v4 CPUs, with 300s and 8GB of RAM for each solver and benchmark pair. We consider 162,060 unsatisfiable problems from SMT-LIB [8], across all logics except those with floating point arithmetic, as determined by `cvc5` [5, Sec. 4]. We split them into 38,732 problems with the BV theory (the BVs set) and 123,328 problems without (the NON-BVs set).

Proof Production Cost. The cost of proof production is summarized in Table 1 and Figs. 3a to 3d. The impact of running without `o` is negligible overall in NON-BVs, but steep for BVs, both in terms of solving time and number of problems solved, as evidenced by the table and Fig. 3b respectively. This is expected given the effectiveness of combining bit-blasting with black-box SAT solvers. The overhead of `P` is similar for both sets, although more pronounced in BVs. While the total time is around double that of `CVC+S`, Fig. 3c shows a finer distribution, with most problems having a less significant overhead. Moreover, the total number of problems solved is quite similar, as shown in Figs. 3a and 3b, particularly for NON-BVs. The difference in overhead due to `P` between the BVs and NON-BVs sets can be attributed to the cost of managing large proofs, which are more common in BVs. This stems from the well-known blow-up in problem size incurred by bit-blasting, which is reflected in the proofs.

The cost of generating fine-grained steps for the `sr` rule and for the similarly reconstructed theory-specific steps mentioned in Sect. 5, varies again between

the two sets, but more starkly. While for NON-BVs the overall solving time and number of problems solved are very similar between CVC+SP and CVC+SPR, for the BVs set CVC+SPR is significantly slower overall. This difference again arises mainly because of the increased proof sizes. Nevertheless, R leads to only a small increase in unsolved problems in BVs, as shown in Fig. 3b.

The importance of being able to produce proofs for simplification under global assumptions is made clear by Fig. 3a: the impact of disabling s is virtually the same as that of adding P ; moreover, CVC+SPR significantly outperforms CVC+PR. In Fig. 3b the difference is less pronounced but still noticeable.

Proofs Coverage. When using techniques that are not yet fully proof-producing, but still active, *cvc5* inserts *trusted steps* in the proof. These are usually steps whose checking is not inherently simpler than solving. They effectively represent holes in the proof, but are still useful for users who avail themselves of powerful proof-checking techniques. Trusted steps are commonly used when integrating SMT solvers into proof assistants [11, 28, 51].

The percentage of CVC+SPR proofs *without* trusted steps is 92% for BVs and 80% for NON-BVs. That is to say, out of 145,683 proofs, 120,473 of them are fully fine-grained proofs. The vast majority of the trusted steps in the remaining proofs are due to theory-specific preprocessing passes that are not yet fully proof-producing. In NON-BVs, the occurrence of trusted steps is heavily dependent on the specific SMT-LIB logic, as expected. Common offenders are logics with datatypes, with trusted steps for acyclicity checks, and quantified logics, with trusted steps for certain α -equivalence eliminations. In non-linear real arithmetic logics, all cylindrical algebraic coverings proofs are built with trusted steps (see Sect. 5.2), but we note this is the state of the art for CAD-based proofs. As for non-linear integer arithmetic logics, our proof support is still in its early stages, so a significant portion of their theory lemmas are trusted steps.

We stress the extent of our coverage for string proofs, which were previously unsupported by any SMT solver. In the string logics without length constraints, 100% of the proofs are fully fine-grained. This rate goes down to 80% in the logics with length. For the remaining 20%, the overwhelming majority of the trusted steps are for theory-specific preprocessing or some particular string or linear arithmetic inference within the proof of a theory lemma.

Comparison with CVC4 Proofs. We compare the proof coverage of *cvc5* versus CVC4. The *cvc5* proof production replaces CVC4’s [32, 36], which was incomplete and monolithic. CVC4 did not produce proofs at all for strings, substitutions, rewriting, preprocessing, quantifiers, datatypes, or non-linear arithmetic. In particular, simplification over global assumptions had to be disabled when producing proofs. In fragments supported by both systems, CVC4’s proofs are at most as detailed as *cvc5*’s. The only superior aspect of CVC4’s proof production was to support proofs from external SAT solvers [45] used in the BV solver, which are very significant for solving performance, as shown above. Integrating this feature into *cvc5* is left as future work, but we note that there is no limitation in the proof architecture that would prevent it. We also point out that *cvc5* produces resolution proofs for the bit-blasted BV constraints, which can

be checked in polynomial time, whereas external SAT solvers produce DRAT proofs [33] (or reconstructions of them via other tools [19, 20, 37, 39]), which can take exponential time to check. So there is a significant trade-off to be considered.

7 Related Work

Two significant proof-producing state-of-the-art SMT solvers are z3 [22] and veriT [14]. Both can have their proofs successfully reconstructed in proof assistants [3, 12, 13, 51]. They can produce detailed proofs for the propositional and theory reasoning in EUF and linear arithmetic, as well as for quantifiers. However, z3’s proofs are coarse-grained for preprocessing and rewriting, and for bit-vector reasoning, which complicates proof checking. Moreover, to the best of our knowledge, z3 does not produce proofs for its other theories. In contrast, veriT can produce fine-grained proofs for preprocessing and rewriting [6], which has led to a better integration with Isabelle/HOL [51]. However, it does so eagerly, which requires a tight integration between the preprocessing and the proof-production code. In addition, it does not support simplification under global assumptions when producing proofs, which significantly impacts its performance. Other proof-producing SMT solvers are MathSAT5 [17] and SMTInterpol [15]. They produce resolution proofs and theory proofs for EUF, linear arithmetic, and, in SMTInterpol’s case, array theories. Their proofs are tailored towards unsatisfiable core and interpolant generation, rather than external certification. Moreover, they do not seem to provide proofs for preprocessing, clausification or rewriting.

While cvc5 is possibly the only proof-producing solver for the full theory of strings, CERTISTR [35] is a certified solver for the fragment with concatenation and regular expressions. It is automatically generated from Isabelle/HOL [44] but is significantly less performant than cvc5, although a proper comparison would need to account for proof-checking time in cvc5’s case.

8 Conclusion and Future Work

We presented and evaluated a flexible proof production architecture, showing it is capable of producing proofs with varying levels of granularity in a scalable manner for a state-of-the-art and industrial-strength SMT solver like cvc5.

Since currently, there is no standard proof format for SMT solvers, our architecture is designed to support multiple proof formats via a final post-processing transformation to convert internal proofs accordingly. We are developing backends for the LFSC [52] proof checker and the proof assistants Lean 4 [21], Isabelle/HOL [44], and Coq [10], the latter two via the Alethe proof format [50]. Since using these tools requires mechanizing the respective target proof calculi in their languages, besides external checking, another benefit is to decouple confidence on the soundness of the proof calculi from the internal cvc5 proof calculus.

A considerable challenge for SMT proofs is the plethora of rewrite rules used by the solvers, which are specific for each theory and vary in complexity. In

particular, string rewrites can be very involved [46] and hard to check. We are also developing an SMT-LIB-based DSL for specifying rewrite rules, to be used during proof reconstruction to decompose rewrite steps in terms of them, thus providing more fine-grained proofs for rewriting.

Finally, we plan to incorporate into the proof-production architecture the unsupported theories and features mentioned in Sects. 5.2 and 6, particularly those relevant for solving performance that currently either leave holes in proofs, such as theory pre-processing or non-linear arithmetic reasoning, or that have to be disabled, such as the use of external SAT solvers in the BV theory.

References

1. Ábrahám, E., Davenport, J.H., England, M., Kremer, G.: Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. *J. Log. Algebr. Methods Program.* **119**, 100633 (2021)
2. Abrahám, E., Davenport, J.H., England, M., Kremer, G.: Proving UNSAT in SMT: the case of quantifier free non-linear real arithmetic. arXiv preprint [arXiv:2108.05320](https://arxiv.org/abs/2108.05320) (2021)
3. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to coq through proof witnesses. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_12
4. Backes, J., et al.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N., Gurfinkel, A. (eds.) Formal Methods in Computer-Aided Design (FMCAD), pp. 1–9. IEEE (2018)
5. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS). LNCS, Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
6. Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P.: Scalable fine-grained proofs for formula processing. *J. Autom. Reason.* **64**(3), 485–510 (2020)
7. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. *All About Proofs Proofs All (APPA)* **55**(1), 23–44 (2014)
8. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
9. Barrett, C., Tinelli, C.: Satisfiability Modulo Theories. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
10. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer, Heidelberg (2004). <https://doi.org/10.1007/978-3-662-07964-5>
11. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. *J. Autom. Reason.* **51**(1), 109–128 (2013)
12. Böhme, S., Fox, A.C.J., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 183–198. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_15

13. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_14
14. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 151–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_12
15. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
16. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Satisfiability modulo transcendental functions via incremental linearization. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 95–113. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_7
17. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
18. Cohen, C.: Construction of real algebraic numbers in CoQ. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 67–82. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_6
19. Cruz-Filipe, L., Heule, M.J.H., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 220–236. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_14
20. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 118–135. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_7
21. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 625–635. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_37
22. de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) Workshops. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008)
23. Déharbe, D., Fontaine, P., Merz, S., Woltzenlogel Paleo, B.: Exploiting symmetry in SMT problems. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 222–236. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_18
24. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_11
25. Eberl, M.: A decision procedure for univariate real polynomials in Isabelle/HOL. In: Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, pp. 75–83. Association for Computing Machinery, New York (2015)
26. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_5
27. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37

28. Ekici, B., et al.: SMTCoq: a plug-in for integrating SMT solvers into Coq. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 126–133. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_7
29. Enderton, H.B.: A Mathematical Introduction to Logic, 2nd edn. Academic Press, Cambridge (2001)
30. Farkas, G.: A Fourier-féle mechanikai elv alkalmazásai. *Mathematikai és Természettudományi Értesítő* **12**, 457–472 (1894). Reference from Schrijver’s Combinatorial Optimization textbook (Hungarian)
31. Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 167–181. Springer, Heidelberg (2006). https://doi.org/10.1007/11691372_11
32. Hadarean, L., Barrett, C., Reynolds, A., Tinelli, C., Deters, M.: Fine grained SMT proofs for the theory of fixed-width bit-vectors. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 340–355. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_24
33. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR, abs/1610.06229 (2016)
34. Joosten, S.J.C., Thiemann, R., Yamada, A.: A verified implementation of algebraic numbers in Isabelle/HOL. *J. Autom. Reason.* **64**, 363–389 (2020)
35. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: Certistr: a certified string solver. In: Popescu, A., Zdancewic, S. (eds.) Certified Programs and Proofs (CPP), pp. 210–224. ACM (2022)
36. Katz, G., Barrett, C., Tinelli, C., Reynolds, A., Hadarean, L.: Lazy proofs for DPLL(T)-based SMT solvers. In: Piskac, R., Talupur, M. (eds.) Formal Methods in Computer-Aided Design (FMCAD), pp. 93–100. IEEE (2016)
37. Kiesl, B., Rebola-Pardo, A., Heule, M.J.H.: Extended resolution simulates DRAT. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 516–531. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_34
38. King, T.: Effective algorithms for the satisfiability of quantifier-free formulas over linear real and integer arithmetic (2014)
39. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 237–254. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_15
40. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 646–662. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_43
41. Mahboubi, A.: Implementing the cylindrical algebraic decomposition within the coq system. *Math. Struct. Comput. Sci.* **17**(1), 99–127 (2007)
42. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 453–468. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_33
43. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *J. ACM* **53**(6), 937–977 (2006)
44. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>

45. Ozdemir, A., Niemetz, A., Preiner, M., Zohar, Y., Barrett, C.: DRAT-based bit-vector proofs in CVC4. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 298–305. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_21
46. Reynolds, A., Nötzli, A., Barrett, C., Tinelli, C.: High-level abstractions for simplifying extended string constraints in SMT. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 23–42. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_2
47. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: Reductions for strings and regular expressions revisited. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 225–235. IEEE (2020)
48. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 453–474. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_24
49. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Hoboken (1998)
50. Schurr, H.-J., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: towards a generic SMT proof format (extended abstract). CoRR, abs/2107.02354 (2021)
51. Schurr, H.-J., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 450–467. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_26
52. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods Syst. Des. **42**(1), 91–118 (2013)
53. Thiemann, R., Yamada, A.: Algebraic numbers in Isabelle/HOL. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 391–408. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_24

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

