Flexible Proof Production in an Industrial-Strength SMT Solver⋆

Haniel Barbosa[1], Andrew Reynolds[2], Gereon Kremer[3],
Hanna Lachnitt[3], Aina Niemetz[3], Andres Nötzli[3], Alex Ozdemir[3],
Mathias Preiner[3], Arjun Viswanathan[2], Scott Viteri[3], Yoni Zohar[4],
Cesare Tinelli[2], Clark Barrett[3]

[1] Universidade Federal de Minas Gerais, Belo Horizonte, Brasil
[2] The University of Iowa, Iowa City, USA
[3] Stanford University, Stanford, USA
[4] Bar-Ilan University, Ramat Gan, Israel

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 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 (Section 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 (Section 3), in particular how we support eager and lazy proof production with built-in proof reconstruction (Section 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 (Section 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 (Section 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 $(a_1, \ldots, a_n)$ 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 Figure 1. Proofs are produced and stored modularly by
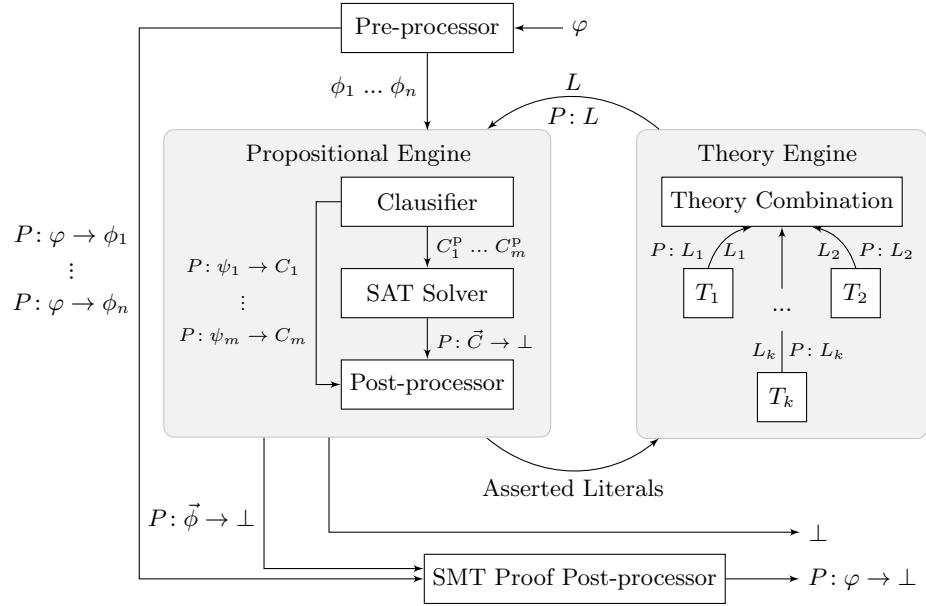
Fig. 1: Flexible proof-production architecture for CDCL($\mathcal{T}$)-based SMT solvers. In the above, $\psi_i \in \{\vec{\phi}, \vec{L}\}$ for each i, with ψ_i not necessarily distinct from ψ_{i+1}.

each solving component, which also checks they meet the expected proof structure 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 $\phi_1, \ldots, \phi_n$. For each ϕ_i, the pre-processor stores a proof $P : \varphi \to \phi_i$ justifying its derivation from φ.

The *propositional engine* receives the preprocessed formulas, and its *clausifier* converts them into a conjunctive normal form $C_1 \wedge \cdots \wedge C_l$. A proof $P : \psi \to C_i$ 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 $C_1^{\mathrm{p}}, \ldots, C_l^{\mathrm{p}}$, 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 $C_1, \ldots, C_m$ generated until then are jointly unsatisfiable, and the SAT solver yields a proof $P : C_1 \wedge \cdots \wedge C_m \to \bot$. Note that

the proof is in terms of the first-order clauses, as are the derivation rules that conclude $\bot$ from them.

The post-processor of the propositional engine connects the assumptions of the SAT solver proof with the clausifier proofs, building a proof $P : \phi_1 \wedge \cdots \wedge \phi_n \to \bot$. 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 $P : \varphi \to \phi_i$. The resulting proof $P : \varphi \to \bot$ 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

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

with *identifier* r, *premises* $\varphi_1, \ldots, \varphi_n$, *arguments* $t_1, \ldots, t_m$, *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

$$\text{refl } \frac{_- \mid t}{t \approx t} \qquad \text{trans } \frac{r \approx s \quad s \approx t}{r \approx t} \qquad \text{cong } \frac{\vec{s} \approx \vec{t} \mid f}{f(\vec{s}) \approx f(\vec{t})} \text{ if } f(\vec{s}) \text{ is well sorted}$$

$$\text{symm } \frac{s \approx t}{t \approx s} \qquad \text{sr } \frac{\varphi \quad \vec{\varphi} \mid \mathcal{S}, \mathcal{R}, \mathcal{D}, \psi}{\psi} \text{ if } \mathcal{S}(\varphi, \mathcal{D}(\vec{\varphi}))\!\uparrow\!\downarrow_{\mathcal{R}} = \mathcal{S}(\psi, \mathcal{D}(\vec{\varphi}))\!\uparrow\!\downarrow_{\mathcal{R}}$$

$$\text{eq\_res } \frac{\varphi \quad \varphi \approx \psi}{\psi} \qquad \text{atom\_rewrite } \frac{_- \mid \mathcal{R}, s}{s \approx t} \text{ if } s\!\downarrow_{\mathcal{R}} = t \qquad \text{witness } \frac{_- \mid k}{k \approx k\!\uparrow}$$

$$\text{assume } \frac{_- \mid \varphi}{\varphi} \qquad \text{scope } \frac{\varphi \mid \varphi_1, \ldots, \varphi_n}{\varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \varphi}$$

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

their children. We call a single-root graph rooted at node N a *proof*. A proof P is *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 $N = (r, (N_1, \ldots, N_m), \vec{t})$, $\Psi(N)$ is the formula returned by the proof checker for rule r when given premises $\Psi(N_1), \ldots, \Psi(N_n)$ 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. We will identify a well-formed proof with its root node.

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 Figure 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

---

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

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.

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 $(t_1 \mapsto s_1, \ldots, t_n \mapsto s_n)$ 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 $i = 1, \ldots, n$; *sequential*, which splits σ into n substitutions $(t_1 \mapsto s_1), \ldots, (t_n \mapsto s_n)$ 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* $(t_1 \mapsto s_1, \ldots, t_n \mapsto s_n)$ 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 $\mathsf{ite}(x \approx z, y, z)$ and $y - z$, respectively, and φ is the formula $\mathsf{ite}(x \approx k_2, k_1 \approx y, k_1 \approx z)$, the witness form $\varphi{\uparrow}$ of φ is the formula

---

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

$\mathsf{ite}(x \approx y - z, \mathsf{ite}(x \approx z, y, z) \approx y, \mathsf{ite}(x \approx z, y, z) \approx z)$. When a Skolem variable k 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 $\varphi_1, \ldots, \varphi_n$ made to prove a formula φ, inferring the formula $\varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \varphi$.

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

---

[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 $c_{\mathsf{pre}}, c_{\mathsf{post}}$ are sets of pairs of equalities and the proof generators justifying them.

$\mathsf{getProof}(g, \varphi)$ where g contains c_{pre}, c_{post} and φ is $t_1 \approx t_2$

1: $B := \emptyset$, $R := \emptyset$
2: $\mathsf{getTermConv}(t_1, c_{\mathsf{pre}}, c_{\mathsf{post}}, B, R)$
3: **if** $R[t_1] \neq t_2$ **then** fail **else return** $\mathsf{getProof}(B, t_1 \approx R[t_1])$

$\mathsf{getTermConv}(s, c_{\mathsf{pre}}, c_{\mathsf{post}}, B, R)$, where $s = f(s_1, \ldots, s_n)$

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

---

proof generators are used. For brevity, we only describe in detail (see Section 3.2) 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 $\varphi_1, \ldots, \varphi_n$ are jointly unsatisfiable, the SMT solver has a reference to a proof node P that concludes $\bot$ from the free assumptions $\varphi_1, \ldots, \varphi_n$. After the post-processor is run, the (closed) proof $(\mathsf{scope}, P', (\varphi_1, \ldots, \varphi_n))$ 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 $t_1 \approx t_2$, $\mathsf{getProof}$ 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$(s, c_{\mathsf{pre}}, c_{\mathsf{post}}, B, R)$ 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$(t_1, c_{\mathsf{pre}}, c_{\mathsf{post}}, B, R)$ 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 $f(R[s_1], \ldots, R[s_n])$, 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 $t \approx \bot$, 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 $c_{\mathsf{pre}} = \{(f(b) + f(a) \approx f(a) + f(b), g^{\mathsf{AC}}), (a - 0 \approx a, g_0^{\mathsf{Arith}})\}$ and $c_{\mathsf{post}} = \{(f(a) + f(b) < f(a) + f(b) \approx \bot, g_1^{\mathsf{Arith}})\}$. 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$(g, t \approx \bot)$ initiates the traversal with getTermConv$(t, c_{\mathsf{pre}}, c_{\mathsf{post}}, \emptyset, \emptyset)$. 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 $\bot$ 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 $f(a) + f(b) < f(a) + f(b) \approx \bot$. At this point, the execution terminates with $R[f(b)+f(a) < f(a+0)+f(b)] = \bot$, as expected. A proof for $t \approx \bot$ with the following structure can then be extracted from B:

$$
P_0 : \mathsf{cong} \cfrac{\mathsf{Lazy}\cfrac{g^{\mathsf{AC}}}{f(b)+f(a) \approx f(a)+f(b)} \quad P_1 \mid\; <}{f(b)+f(a) < f(a-0)+f(b) \approx f(a)+f(b) < f(a)+f(b)} \qquad P_2 : \mathsf{refl}\cfrac{\text{-}\mid f(b) \approx f(b)}{f(b) \approx f(b)}
$$

$$
\mathsf{trans}\cfrac{P_0 \qquad \mathsf{Lazy}\cfrac{g_1^{\mathsf{Arith}}}{f(a)+f(b) < f(a)+f(b) \approx \bot}}{f(b)+f(a) < f(a-0)+f(b) \approx \bot} \qquad P_1 : \mathsf{cong}\cfrac{\mathsf{cong}\cfrac{\mathsf{Lazy}\cfrac{g_0^{\mathsf{Arith}}}{a-0 \approx a} \mid\; f}{f(a-0) \approx f(a)} \qquad P_2 \mid\; +}{f(a-0)+f(b) \approx f(a)+f(b)}
$$

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 $(\mathsf{sr}, (Q_{\psi_0}, \vec{Q}), (\mathcal{S}, \mathcal{R}, \mathcal{D}, \psi))$. Recall from Figure 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:

$$(\mathsf{eq\_res}, (Q_{\psi_0}, (\mathsf{trans}, (R_0, (\mathsf{symm}, R_1)))))$$

with $R_i = (\mathsf{trans}, ((\mathsf{subs}, \vec{Q}_{\vec{\varphi}}, (\mathcal{S}, \mathcal{D}, \psi_i)), (\mathsf{rewrite}, (), (\mathcal{R}, \mathcal{S}(\psi_i, \mathcal{D}(\vec{\varphi}))))))$ for $i \in \{0, 1\}$ where $\vec{\varphi}$ are the conclusions of $\vec{Q}_{\vec{\varphi}}$, and subs and $\mathsf{rewrite}$ 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 $(\mathsf{subs}, \vec{Q}_{\vec{\varphi}}, (\mathcal{S}, \mathcal{D}, t))$ of R_i above proving $t \approx s$ with $s = \mathcal{S}(\psi_i, \mathcal{D}(\vec{\varphi}))$ and $\mathcal{D}(\vec{\varphi}) = (t_1 \mapsto s_1, \ldots, t_n \mapsto s_n)$. Substitution steps can be expanded to fine-grained proofs using a term-conversion proof generator. First, for each $j = 1, \ldots, n$, 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 $(c_{\mathsf{pre}}, c_{\mathsf{post}})$. 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 $\mathsf{getProof}(C, t \approx s)$, 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 $(\mathsf{trans}, (P_1, \ldots, P_n))$ where, for $j = 1, \ldots, n$: P_j is generated by $\mathsf{getProof}(C_j, s_{j-1} \approx s_j)$; $s_0 = t$; 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 $(\mathsf{rewrite}, (), (\mathcal{R}, t))$, which proves the equality $t \approx t{\uparrow}{\downarrow}_{\mathcal{R}}$. 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 $(t_1 \approx s_1, g), \ldots, (t_n \approx s_n, g)$ where g is a proof generator that returns $(\mathsf{atom\_rewrite}, (), (\mathcal{R}, t_i))$ 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 $(\mathsf{witness}, (), (k))$ 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 $\mathsf{getProof}(C, t \approx t{\uparrow}{\downarrow}_{\mathcal{R}})$.

5   SMT Proofs

Here we briefly describe each component shown in Section 2 and how it produces proofs with the infrastructure from Sections 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 clausification and resolution proofs, as explained in Section 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 Section 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 $\vec{\ell}$. A lemma over the combined theory is generated by replacing ℓ in ψ by $\vec{\ell}$. 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 Figure 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 $\bot$. 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.

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 $\bot$ from the resulting clauses. CNF proofs are a combination of Boolean transformations and introductions of Boolean formulas representing the definition of

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

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

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 Section 3, which was invaluable for a correct implementation, is disabled for evaluating performance. Experiments ran on a cluster with 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).
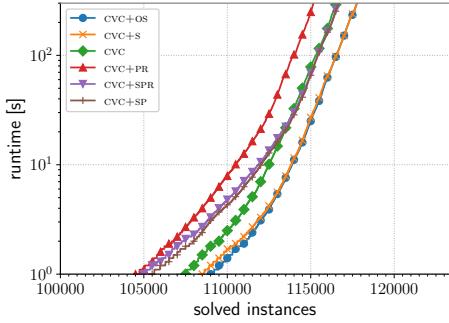
Fig. 3a: Cactus plot for NON-BVs
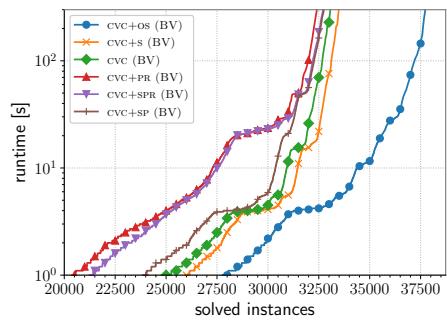
Fig. 3b: Cactus plot for BVs
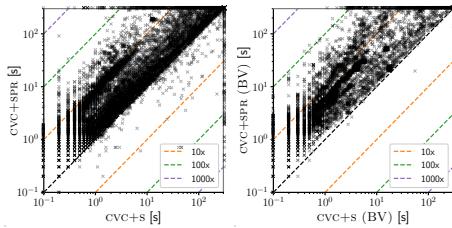
Fig. 3c: Scatter plot of overall proof cost
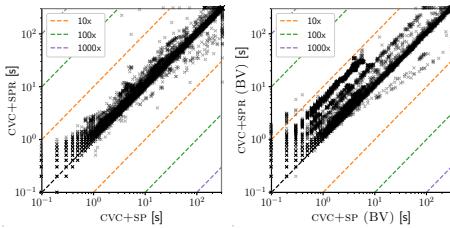
Fig. 3d: Reconstruction cost

Proof production cost The cost of proof production is summarized in Table 1 and Figures 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 Figure 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, Figure 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 Figures 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 Section 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 Figure 3b.

The importance of being able to produce proofs for simplification under global assumptions is made clear by Figure 3a: the impact of disabling S is virtually the same as that of adding P; moreover, CVC+SPR significantly outperforms CVC+PR. In Figure 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 Section 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 Sections 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. Erika Ábrahám, James H. Davenport, Matthew England, and Gereon Kremer. Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. *Journal of Logical and Algebraic Methods in Programming*, 119(100633), 2021.

2. Erika Abrahám, James H Davenport, Matthew England, and Gereon Kremer. Proving UNSAT in SMT: The case of quantifier free non-linear real arithmetic. *arXiv preprint arXiv:2108.05320*, 2021.

3. 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 (CPP)*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

4. John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based automated reasoning for AWS access policies using SMT. In Nikolaj Bjørner and Arie Gurfinkel, editors, *Formal Methods In Computer-Aided Design (FM-CAD)*, pages 1–9. IEEE, 2018.

5. Haniel Barbosa, Clark 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)*, Lecture Notes in Computer Science, 2022. To appear.

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. Clark Barrett, Leonardo de Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. In *All about Proofs, Proofs for All (APPA)*, 2014.

8. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

9. Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking.*, pages 305–343. Springer, 2018.

10. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

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 (CPP)*, volume 7086 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2011.

13. Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In Matt Kaufmann and Lawrence Paulson, editors, *Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

14. 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, *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

15. Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In Alastair F. Donaldson and David Parker, editors, *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012.

16. Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Satisfiability modulo transcendental functions via incremental linearization. In Leonardo de Moura, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 10395 of *Lecture Notes in Computer Science*, pages 95–113. Springer, 2017.

17. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

18. Cyril Cohen. Construction of real algebraic numbers in coq. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 67–82, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

19. Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017.

20. Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135, 2017.

21. Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

22. 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.

23. David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In *Proceedings of the 23rd International Conference on Automated Deduction*, Proc. Conference on Automated Deduction (CADE), pages 222–236, Berlin, Heidelberg, 2011. Springer-Verlag.

24. Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer Berlin Heidelberg, 2006.

25. Manuel Eberl. A decision procedure for univariate real polynomials in isabelle/hol. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, page 75–83, New York, NY, USA, 2015. Association for Computing Machinery.

26. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

27. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

28. 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 (CAV)*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.

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

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

31. Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Fernanto Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

32. Liana Hadarean, Clark W. Barrett, Andrew Reynolds, Cesare Tinelli, and Morgan Deters. Fine grained SMT proofs for the theory of fixed-width bit-vectors. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 9450 of *Lecture Notes in Computer Science*, pages 340–355. Springer, 2015.

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

34. Sebastiaan J. C. Joosten, René Thiemann, and Akihisa Yamada. A verified implementation of algebraic numbers in isabelle/hol. *Journal of Automated Reasoning*, 64:363–389, 2020.

35. 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.

36. 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.

37. Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 10900 of *Lecture Notes in Computer Science*, pages 516–531. Springer, 2018.

38. Tim King. Effective algorithms for the satisfiability of quantifier-free formulas over linear real and integer arithmetic, 2014.

39. Peter Lammich. Efficient verified (UN)SAT certificate checking. In Leonardo de Moura, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 10395 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2017.

40. Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2014.
41. Assia Mahboubi. Implementing the cylindrical algebraic decomposition within the coq system. *Mathematical Structures in Computer Science*, 17(1):99–127, 2007.
42. Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Rewriting Techniques and Applications (RTA)*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.
43. 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.
44. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
45. Alex Ozdemir, Aina Niemetz, Mathias Preiner, Yoni Zohar, and Clark W. Barrett. Drat-based bit-vector proofs in CVC4. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 11628 of *Lecture Notes in Computer Science*, pages 298–305. Springer, 2019.
46. Andrew Reynolds, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. High-level abstractions for simplifying extended string constraints in SMT. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification (CAV), Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2019.
47. Andrew Reynolds, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. Reductions for strings and regular expressions revisited. In *Formal Methods In Computer-Aided Design (FMCAD)*, pages 225–235. IEEE, 2020.
48. Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. Scaling up DPLL(T) string solvers using context-dependent simplification. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification (CAV)*, volume 10427 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2017.
49. Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.
50. Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). *CoRR*, abs/2107.02354, 2021.
51. 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, *Proc. Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 450–467. Springer, 2021.
52. 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.
53. René Thiemann and Akihisa Yamada. Algebraic numbers in isabelle/hol. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 391–408, Cham, 2016. Springer International Publishing.