Reconstructing Fine-Grained Proofs of Rewrites Using a Domain-Specific Language

Andres Nötzli*, Haniel Barbosa‡, Aina Niemetz*, Mathias Preiner*, Andrew Reynolds†, Clark Barrett*, and Cesare Tinelli†

*Stanford University, Stanford, USA, ✉{noetzli, niemetz, preiner, barrett}@cs.stanford.edu
†The University of Iowa, Iowa City, USA, ✉{andrew-reynolds, cesare-tinelli}@uiowa.edu
‡Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, ✉hbarbosa@dcc.ufmg.br

***Abstract*—Satisfiability modulo theories (SMT) solvers are widely used to prove security and safety properties of computer systems. For these applications, it is crucial that the result reported by an SMT solver be correct. Recently, there has been a renewed focus on producing independently checkable proofs in SMT solvers, partly with the aim of addressing this risk. These proofs record the reasoning done by an SMT solver and are ideally detailed enough to be easy to check. At the same time, modern SMT solvers typically implement hundreds of different term-rewriting rules in order to achieve state-of-the-art performance. Generating detailed proofs for applications of these rules is a challenge, because code implementing rewrite rules can be large and complex. Instrumenting this code to additionally produce proofs makes it even more complex and makes it harder to add new rewrite rules. We propose an alternative approach to the direct instrumentation of the rewriting module of an SMT solver. The approach uses a domain-specific language (DSL) to describe a set of rewrite rules declaratively and then reconstructs detailed proofs for specific rewrite steps on demand based on those declarative descriptions.**

I. Introduction

Satisfiability modulo theories (SMT) solvers are widely used to reason about the security and safety of critical systems [1, 2, 10, 13]. These applications require a high level of trust in the correctness of the underlying solver. SMT solvers, however, are complex pieces of software, in some cases consisting of hundreds of thousands of lines of code. As with any other large and complex software project, they are not immune to bugs [17], which may, in the worst case, cause incorrect results. Due to the size and complexity of SMT solvers and the fact that most of them continue to be in active development, their full verification is currently still out of reach. As a consequence, the best one can do is to check their individual answers based on evidence provided by the solvers themselves.

For quantifier-free inputs reported to be satisfiable, SMT solvers are typically capable of producing as evidence a satisfying model, which can then be used to validate the claim. Note that for quantified formulas, model validation for satisfiable queries is usually still possible although more complex. For unsatisfiable inputs, there have been efforts in recent years towards producing independently checkable *proofs*, which record the reasoning steps required to deduce unsatisfiability. These steps can later be replayed and checked efficiently by a *proof checker*. Proofs can also be used to automatically discharge proof obligations in interactive theorem provers such as Coq [25] and Isabelle [19]. For this use case, the SMT solver acts as an automated tactic. The proof obligation is encoded as an SMT problem and the proof generated by the SMT solver is then used, in essence, to reconstruct a proof in the proof assistant's native proof representation.

Producing and checking proofs for unsatisfiable problems requires considerably more effort than generating and validating models for satisfiable inputs. Additionally, proofs can be produced in many different forms, each with its own trade-offs. When it comes to the form of a proof, one characteristic of interest is the proof's *granularity*. *Fine-grained* proofs enable efficient proof checking since the proofs are detailed enough to not require any search during checking. Similarly, proof reconstruction for interactive theorem provers requires detailed proofs to minimize *holes* that must be proved manually. However, fine-grained proofs are generally more costly to produce. *Coarse-grained* proofs, on the other hand, are cheaper to produce but require more computation to check. Regardless of the proof form, the traditional approach for generating proofs is to instrument each component of the SMT solver to record its reasoning steps, and then consolidate the relevant recorded steps into a single proof.

Instrumentation can be particularly challenging and tedious for the components of the solver that implement *rewriting*. Modern SMT solvers implement hundreds of rewrite rules for normalizing and simplifying terms to achieve state-of-the-art performance. Because rewriting is an essential part of the reasoning done by the solver, a proof must contain a record of the rewriting steps performed. Previous work [6] has described how to generate rewriting proofs whose only holes are *atomic rewrites*, i.e., an application of a single rewrite step to a single term. Such proofs use a single generic rule for all atomic rewrites. This approach has two major drawbacks, however: (i) the proof checker has to guess or search for the rule to apply or trust that the rewriting was done correctly; and (ii) if used in a proof assistant, each rewrite step becomes a proof obligation that must be discharged by the user. On the other

This work was supported in part by DARPA (award no. FA8650-18-2-7861), the Stanford Agile Hardware Center, and by a gift from Amazon Web Services.

https://doi.org/

hand, if occurrences of atomic rewrites are proven using a fixed set of specific rules, we can prove the correctness of the rules in this set once and for all and then use those proofs during proof checking or during replay in a proof assistant.

As mentioned above, instrumenting rewriting code for proof generation is difficult and tedious. Additionally, since rewriting is applied not only as a preprocessing step but also repeatedly during the solving process, rewriting code (including any instrumentation) must be efficient. In this work, *we propose an alternative approach that does not rely on instrumenting the original rewriter.* Instead, our approach treats the rewriter as a black box and relies on a post-processing phase to expand coarse-grained rewriting steps ocurring in proofs into fine-grained proofs. We use a generic reconstruction algorithm that consults a separate database of core rewrite rules in order to produce the detailed proof using as input only the terms before and after an atomic rewrite. The core rewrite rules need not include every atomic rewrite. It is enough for every atomic rewrite to be reconstructable using one or more of the core rewrite rules. This simplifies the task of populating the database, as the rules used can be fewer and simpler than what is actually done in the solver. To specify the set of rules in the database, we propose the use of a high-level, domain-specific language (DSL) designed to succinctly express a set of core rules to be used in proofs. We have used this approach to reconstruct detailed proofs for the theory of strings in the SMT solver CVC5 [4]. In our experience, this approach greatly reduces the burden of proof production for rewriting code, as it allows a solver developer to quickly and incrementally define core rewrite rules to help fill holes in proofs. Also, note that rewrite steps are typically equality-preserving. Because we treat the rewriter as a black box (i.e., independently from any specific solver or implementation), our approach is quite general and could be used to produce or complete proofs for any tool or situation where proofs of equivalence are needed. By providing a DSL for specifying rewrites and an automatic reconstruction algorithm for coarse-grained atomic rewrites, we expect to greatly improve the flexibility and usability of proofs from SMT solvers. Our contributions are as follows:

- We propose an SMT-LIB-like domain specific language for defining rewrite rules.
- We describe an algorithm that can use such rules to reconstruct detailed proofs for rewrites in an SMT solver.
- We implement our approach in CVC5 and report on a case study reconstructing detailed proofs for rewrites in the theory of strings.
- We evaluate our implementation and show that it has reasonable performance in practice.

In the remainder of the paper, we provide an overview of our approach (Section II) and then describe the language (Section III) and the proof reconstruction algorithm (Section IV) in more detail. We then present a case study of using the approach to produce detailed proofs for the theory of strings in CVC5 (Section V) and evaluate our approach (Section VI) experimentally. Finally, we conclude with some future directions for the language and our approach (Section VII).

A. Related Work

Barbosa et al. [5] introduced a framework for modularizing the production of proofs for formula processing and term rewriting, a long-standing challenge for SMT solvers. A similar and more general framework for overall proof production [6] was recently implemented in CVC5. However, both frameworks produce proofs that are coarse-grained with respect to atomic rewrites, i.e., each atomic rewriting step is a single proof step without further justification.

In the integration between the veriT solver [11] and the Isabelle/HOL proof assistant [23], which leverages the framework from [5], the Sledgehammer tool [8] sends proof goals to veriT and then reconstructs proofs from those emitted by veriT in the Alethe proof format [22]. The reconstructed proofs can then be used to prove the original Isabelle/HOL proof goals. An initial version of this framework was similarly coarse-grained: every atomic rewrite applied by the solver was justified with a single Alethe proof rule. As shown by Schurr et al. [23], this led to failures and performance issues in the Isabelle/HOL reconstruction of Alethe proofs. One approach to address this issue is to extend the Alethe format to contain finer-grained rules for atomic rewrites, and to integrate each of these rules into both veriT and Sledgehammer. This has been shown to increase the success rate of proof reconstruction, but the process is fully manual: every new rule added requires updating the solver, the format, and the reconstruction.

Nötzli [20] proposed a language for rewrite rules in SMT solvers with the goal of automatically generating executable code that replaces parts of an existing rewriter. The DSL presented in this work is an evolution of that language and is focused on the needs of proof reconstruction. Our dedicated rewrite language bears some similarity to equational specification languages such as Maude [12], ELAN [9], and CafeOBJ [14]. In contrast to those more general-purpose languages, the DSL presented in this work has a much more narrow scope and includes specific features to support its use in proof reconstruction.

B. Formal Preliminaries

We formalize our work within the setting of many-sorted logic with equality (see e.g., [15, 26]). Let S be a set of *sort symbols*. For every sort $\tau \in S$, we assume an infinite set of variables of that sort. A *signature* Σ consists of a set $\Sigma^{\mathrm{s}} \subseteq S$ of sort symbols and a set Σ^{f} of function symbols. Constants are treated as 0-ary functions. We assume that Σ includes a sort Bool, interpreted as the Boolean domain, and the Bool constants $\top$ (*true*) and $\bot$ (*false*). Signatures do not contain separate predicate symbols and use instead function symbols that return a Bool value. We further assume that for all sorts $\tau \in S$, Σ contains an equality symbol $\approx\colon \tau \times \tau \to \mathsf{Bool}$, interpreted as the identity relation. Finally, we assume the usual definitions of well-sorted terms, literals, and formulas.

A Σ-*interpretation* I maps: each $\tau \in \Sigma^{\mathrm{s}}$ to a distinct non-empty set of values τ^I (the *domain* of τ in I); each variable

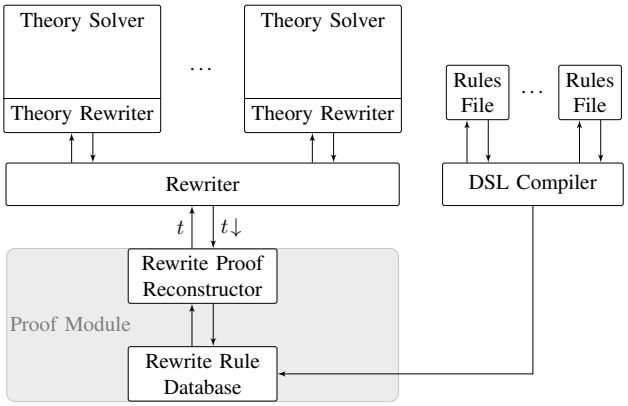


Fig. 1: Overview of the components of our approach

x of sort τ to an element $x^I \in \tau^I$; and each $f^{\tau_1 \cdots \tau_n \tau} \in \Sigma^{\mathrm{f}}$ to a total function $f^I : \tau_1^I \times \ldots \times \tau_n^I \to \tau^I$ if $n > 0$, and to an element in τ^I if $n = 0$. We use the usual notion of a satisfiability relation $\models$ between Σ-interpretations and Σ-formulas. A Σ-*theory* T is a non-empty class of Σ-interpretations closed under variable reassignment (i.e., every interpretation that only disagrees with an interpretation in T on how it interprets variables is also in T). A Σ-formula φ is T-*satisfiable* (resp., T-*unsatisfiable*, T-*valid*) if it is satisfied by some (resp., no, all) interpretations in T. We write $\models_T \varphi$ when φ is T-valid. We say that φ_1 T-*entails* φ_2, and write $\varphi_1 \vDash_T \varphi_2$, when $\vDash_T \varphi_1 \Rightarrow \varphi_2$.

II. Overview

In this paper, we assume a fixed theory T and consider only rewrite rules that preserve equivalence in T. Formally, let $t{\downarrow_{\mathrm{a}}}$ denote the result of performing atomic rewrite a on term t. Then, we require that $\models_T t \approx t{\downarrow_{\mathrm{a}}}$.

Figure 1 shows an overview of our proposed approach. Modern SMT solvers implement a large number of theory-specific rewrite rules. Conceptually, the implementation of these theory-specific rewrite rules can be seen as *theory rewriter* modules of the individual theory solvers. A *rewriter* is a module that traverses a given term and invokes the appropriate theory rewriter on each subterm. To determine which theory rewriter to call, the rewriter looks at the top-most symbol of the subterm and calls the theory whose signature contains that symbol. The *proof module*, which manages proofs, utilizes the *rewrite proof reconstructor* to fill in the missing subproofs for rewrites. The rewrite proof reconstructor bases its reconstruction on a set of rewrite rules, stored in the *rewrite rule database*. This database is generated at compile-time from a set of rewrite rules written in our DSL RARE (described in Section III). These rewrite rules are stored in text files, which are compiled to C++ code using the DSL compiler. The compiled code populates a discrimination tree [16] which is an index used for matching terms with applicable rewrite rules during proof reconstruction. Assuming that the rewrite rules in the rewrite rule database are correct, our reconstruction is sound since only these rules are used to construct the proofs.

⟨*rule*⟩ ::= `(` **`define-rule`** ⟨*symbol*⟩ `(` ⟨*par*⟩* `)` ⟨*expr*⟩ ⟨*expr*⟩ `)`
| `(` **`define-cond-rule`** ⟨*symbol*⟩ `(` ⟨*par*⟩* `)` ⟨*expr*⟩ ⟨*expr*⟩ ⟨*expr*⟩ `)`
| `(` **`define-rule*`** ⟨*symbol*⟩ `(` ⟨*par*⟩* `)` ⟨*expr*⟩ ⟨*expr*⟩ [⟨*expr*⟩] `)`

⟨*par*⟩ ::= ⟨*symbol*⟩ ⟨*sort*⟩ ⟨*attr*⟩*

⟨*sort*⟩ ::= `?` | ⟨*symbol*⟩ | `(` ⟨*symbol*⟩ ⟨*sort*⟩+ `)`
| `(` _ ⟨*symbol*⟩ ⟨*idx*⟩+ `)`

⟨*idx*⟩ ::= `?` | ⟨*numeral*⟩

⟨*expr*⟩ ::= ⟨*const*⟩ | ⟨*id*⟩ | `(` ⟨*id*⟩ ⟨*expr*⟩+ `)` | ⟨*let*⟩

⟨*id*⟩ ::= ⟨*symbol*⟩ | `(` _ ⟨*symbol*⟩ ⟨*idx*⟩+ `)`

⟨*let*⟩ ::= `(` **`let`** `(` ⟨*binding*⟩+ `)` ⟨*expr*⟩ `)`

⟨*binding*⟩ ::= `(` ⟨*symbol*⟩ ⟨*expr*⟩ `)`

Fig. 2: Overview of the grammar of RARE.

The output of the proof module consists of the proof with the subproofs for rewrites completed.

The rule database may also play a role in proof *checking*. In particular, a stand-alone proof checker may use the database to automatically generate code that can check whether a rule in the database is used correctly. While the syntax is checked in this scheme, the T-validity of the rules in the database is trusted. Checking the rules for T-validity is another task which can (and should) be done separately, perhaps using a proof assistant. We do not address these issues in this paper, but instead focus on the RARE language and the algorithm at the core of the rewrite proof reconstructor.

III. The Language

In this section, we describe the scope, design goals, syntax, and semantics of RARE, our domain-specific language for **r**ewrites, **a**utomatically **re**constructed. To reduce the cost of introducing such a new language into the development workflow of an existing SMT solver, we identify several requirements:

Succinctness: Writing rewrite rules should be simple and concise. Adding new rules should be far less costly than instrumenting existing code.

Expressiveness: The language should be able to express the majority of the rewrite rules used in a state-of-the-art SMT solver.

Accessibility: The language should be easy to parse and understand.

There is an inherent tension between making a DSL succinct and making it expressive. We designed RARE to be as expressive as possible without sacrificing succinctness. To aid with accessibility, its syntax reuses the syntax of the SMT-LIB [7] language standard whenever possible.

As we discuss in Section V, we do not aim for full generality, because certain rewrites, such as polynomial normalization, are less amenable to our approach. Similarly, we assume that constant folding is built into the reconstruction algorithm and therefore does not have to be explicitly defined with rewrite rules.

An input file for RARE consists of a list of rewrite rules whose syntax is defined by the BNF grammar in Figure 2. Rewrite rules are written as S-expressions. For symbols and concrete constants (e.g., integer numbers, string literals), RARE uses the same syntax as the SMT-LIB language. In contrast to SMT-LIB, parameterized sorts such as arrays and bit-vectors do not need to be concrete. Instead, RARE is gradually typed and allows the parameters of such sorts to remain abstract. This allows users to specify rewrites that are, e.g., independent of the bit-width or the sorts of indices and elements in arrays. In the following, we discuss all the different constructs of the language in detail.

Basic Syntax. As indicated in Figure 2, ⟨*rule*⟩ defines three different types of rewrite rules: basic rules (**`define-rule`**), conditional rules (**`define-cond-rule`**), and fixed-point rules (**`define-rule`**$\star$). A *basic rule* consists of a *name*, a list of match *parameters*, the *match* expression, and the *target* expression. The name identifies the rewrite rule and is later used to label steps in the rewrite proof; the list of parameters $\langle par \rangle^*$ introduces the term variables that appear in the rule, along with their sorts; the match expression defines the syntactic shape of terms the rewrite rule applies to; and the target expression defines how a matched term is rewritten. Both the match expression and the target expression have the same syntax as SMT-LIB terms. All the variables that appear in a rewrite rule must either be declared as a parameter or introduced locally with the **`let`** binder.

Basic rules define simple rewrite rules without preconditions. The following example shows such a rule, which defines the rewrite $\mathsf{substr}(\texttt{""}, m, n) \rightsquigarrow \texttt{""}$ from a term denoting the substring from position m to position n of the empty string to just the empty string, regardless of the value of m and n.

```
(define-rule substr-empty ((m Int) (n Int))
  (str.substr "" m n) "")
```

In this example, the match expression specifies that the rule applies to string terms of the form $\mathsf{substr}(\texttt{""}, s, t)$ where the first argument of substr is the empty string, the second argument s is matched by m, and the third argument t is matched by n. The compiler and the proof reconstruction algorithm have built-in knowledge of theory symbols such as substr as defined in the SMT-LIB standard.

Matching. If a variable x appears multiple times in a match expression, the rewrite rule only applies if each occurrence of x matches syntactically identical terms. For example, the match expression `(= (str.++ x1 x2) x2)` with variables `x1` and `x2` matches $a +\!\!+ b \approx b$, but not $a +\!\!+ b \approx c$. For a rewrite rule to apply, a term matched by a declared variable must be of the expected sort. We use `?` to denote that a term can be of any sort, or to match an arbitrary sort parameter. The following example illustrates the use of multiple variable occurrences and abstract sorts.

```
(define-rule eq-refl ((t ?)) (= t t) true)
```

This rule rewrites equalities of syntactically equivalent terms to $\top$, regardless of the sort of the term matched by variable `t`.

Lists. Some operators defined in SMT-LIB, e.g., string concatenation, can be applied to two or more terms. We use variables declared with the **`:list`** attribute to match an arbitrary number of arguments of an operator. The following example shows a rule for flattening string concatenations.

```
(define-rule str-concat-flatten (
    (xs String :list) (s String)
    (ys String :list) (zs String :list))
  (str.++ xs (str.++ s ys) zs) ; match
  (str.++ xs s ys zs))         ; target
```

This rule applies to any string concatenation with another string concatenation as a subterm. The prefix `xs` and the suffix `zs` may be empty (although not at the same time in this case).

Conditional Rules. The previous rewrite rule examples rely on purely syntactic matching. To make matching more expressive, RARE supports the conditional matching of terms using **`define-cond-rule`**. Such rules have an additional argument, the *precondition*, before the match expression. That is either a single condition, expressed by a literal, or a conjunction of them capturing all conditions that must be met for the rule to apply. When reconstructing a proof, these conditions introduce new proof obligations. The following example illustrates the use of conditional rules.

```
(define-cond-rule concat-clash (
    (s1 String) (s2 String :list)
    (t1 String) (t2 String :list))
  (and (= (str.len s1) (str.len t1))
       (not (= s1 t1)))
  (= (str.++ s1 s2) (str.++ t1 t2))
  false)
```

This rule rewrites a word equation $s_1 +\!\!+ s_2 \approx t_1 +\!\!+ t_2$ to $\bot$, provided that two conditions are met: the lengths of the prefixes s_1 and t_1 are the same and the prefixes are distinct in the theory T. For example, this rule applies to the equality $\texttt{"abc"} +\!\!+ x \approx \texttt{"def"} +\!\!+ y$ since both $|\texttt{"abc"}| \approx |\texttt{"def"}| \approx 3$ and $\texttt{"abc"} \not\approx \texttt{"def"}$ hold in the theory of strings. Note that the precondition $|s_1| \approx |t_1|$ does not require the evaluation of $|s_1|$ and $|t_1|$. Instead, it just requires some proof that they are equal. In practice, we prove the precondition by applying additional rewrite rules. This allows us to show that the precondition holds for equalities such as $|x +\!\!+ y| \approx |y +\!\!+ x|$, for instance.

Fixed-Point Rules. As an optimization, RARE allows the definition of fixed-point rules with **`define-rule`**$\star$. These rules are repeatedly applied until they no longer apply. They are most useful for rewrite rules that effectively iterate over arguments of n-ary operators, as we demonstrate in the example below. Fixed-point rules take a match expression, a target

$\mathsf{rc}(t \approx s, d)$

1: **if** $d < 0$ **then return** $\bot$
2: **if** $t \approx s \in \mathsf{P}$ **then**
3: **if** $\mathsf{P}[t \approx s] = (\mathsf{fail}, e)$ **and** $d \leq e$ **then return** $\bot$
4: **if** $\mathsf{P}[t \approx s] \neq (\mathsf{fail}, e)$ **then return** $\top$
5: **if** $t{\downarrow_\mathrm{e}} = s{\downarrow_\mathrm{e}}$ **then** $\mathsf{P}[t \approx s] := \mathsf{eval}$, **return** $\top$
6: **if** $(t \approx s){\downarrow} = \bot$ **then** $\mathsf{P}[t \approx s] := (\mathsf{fail}, \infty)$, **return** $\bot$
7: $\mathsf{P}[t \approx s] := (\mathsf{fail}, d)$
8: **if** $(t, s) = (f(\vec{u}), f(\vec{v}))$ **and**
9: $\mathsf{rc}(u \approx v, d)$ for all $u \approx v \in \vec{u} \approx \vec{v}$ **then**
10: $\mathsf{P}[t \approx s] := \mathsf{cong}$ **return** $\top$
11: **if** $t = f(\vec{u})$ **and** $\vec{u}{\downarrow} = \vec{c}$ **and** $f(\vec{c}){\downarrow_\mathrm{e}} = s{\downarrow_\mathrm{e}}$ **and**
12: $\mathsf{rc}(\vec{u} \approx \vec{c}, d)$ **then**
13: $\mathsf{P}[t \approx s] := \mathsf{ceval}$ **return** $\top$
14: **foreach** $(\mathsf{r}, \vec{p} \approx \vec{q} \Rightarrow u \approx v) \in \mathcal{R}$
15: s.t. $t = \sigma(u)$ for some σ **do**
16: **if** $\mathsf{rc}(\sigma(v) \approx s, d-1)$ **and**
17: $\mathsf{rc}(\sigma(p \approx q), d-1)$ for all $p \approx q \in \vec{p} \approx \vec{q}$ **then**
18: $\mathsf{P}[t \approx s] := \mathsf{r}$, **return** $\top$
19: **return** $\bot$

Fig. 3: The algorithm for reconstructing a proof sketch P from rule database $\mathcal{R}$. Calling $\mathsf{rc}(t \approx s, d)$ returns true if the proof of $t \approx s$ having depth at most d can be constructed.

expression, and, optionally, a context expression as arguments. The *target* expression indicates the recursion step, i.e., the term that should be rewritten next. The *context* expression indicates how to use the result of the recursion step to construct the final result. It is a term with a placeholder _ for the location of the result of the recursion step. Omitting the context expression is the same as providing a context of _, which indicates that the result of the recursion step is also the final result. The following example defines a rewrite rule that distributes the string length operator over the elements in a concatenation:

```
(define-rule* str-len-concat-rec (
    (s1 String) (s2 String)
    (rest String :list))
  (str.len (str.++ s1 s2 rest))
  (str.len (str.++ s2 rest))
  (+ (str.len s1) _))
```

This rule specifies that we rewrite $|s_1 +\!\!+ s_2 +\!\!+ \ldots|$ to $|s_1| + t$, where t is the result of recursively applying the rule to the term $|s_2 +\!\!+ \ldots|$.

Annotating rules to be fixed-point rules reduces the search space during reconstruction, because the reconstruction algorithm always applies these rules until a fixed-point is reached, without considering possible interleavings of other rules. This improves efficiency at the cost of not considering some possible reconstructions. Thus, there is a trade-off, and this feature must be used carefully.

IV. Reconstructing Proofs

In this section, we describe our approach for constructing proofs of rewrites $t \approx t{\downarrow_\mathrm{a}}$ using rules from a *rewrite rule database* $\mathcal{R}$ obtained by compiling RARE rules. To simplify the presentation, we do not consider fixed-point rules for now, postponing the general case to later in this section. The database $\mathcal{R}$ stores a set of labeled implications of the form $(\mathsf{r}, \vec{p} \approx \vec{q} \Rightarrow t \approx s)$, where r is a rule identifier, $\vec{p} \approx \vec{q}$ is a conjunction of term equalities, and $\vec{p} \approx \vec{q} \vDash_T t \approx s$. Operationally, the rule specifies that a term t can be rewritten to a term s when the premises $\vec{p} \approx \vec{q}$ hold. Note that using just equalities in the premises is without loss of generality since an arbitrary formula φ can be expressed as a premise of the form $\varphi \approx \top$. Unconditional rules are represented using the single, valid premise $\top \approx \top$.

$$\mathsf{eval}\frac{}{t \approx t{\downarrow_\mathrm{e}}} \qquad \mathsf{trans}\frac{r \approx s \quad s \approx t}{r \approx t}$$

$$\mathsf{cong}\frac{\vec{s} \approx \vec{t}}{f(\vec{s}) \approx f(\vec{t})} \qquad \mathsf{ceval}\frac{\vec{s}{\downarrow} \approx \vec{t}{\downarrow}}{f(\vec{s}) \approx (f(\vec{t})){\downarrow_\mathrm{e}}}$$

Fig. 4: The basic proof rules; $t{\downarrow_\mathrm{e}}$ is the evaluated form of t.

Our proof reconstruction for an equality $t \approx t{\downarrow_\mathrm{a}}$ based on the rule database $\mathcal{R}$ consists of two phases. In the first one, captured by the algorithm in Figure 3, we search for a *proof sketch* P, which is a map from term equalities to rules that can be used to prove them in a final proof. In the second, the discovered proof sketch, if any, is transformed into a full proof, which may consist of the application of multiple rules from $\mathcal{R}$, as described later in this section.

A. Finding Proof Sketches

Figure 3 shows our algorithm rc for recursively finding proof sketches for equalities $t \approx s$. The inputs are the (oriented) equality to prove and an integer d specifying an upper bound on the *depth* of rc's recursive calls. Some recursive calls are generated by the algorithm's attempt to justify the use of a conditional rule from $\mathcal{R}$ to prove the input equality. In that case, the algorithm attempts to prove the premises of the conditional rule, but does so for a decreased depth. The rationale behind the depth limit on the search is that there is no guarantee that preconditions are simpler than the current equality to be proved, and so there is no guarantee of termination in general. The depth limit can be chosen by the user at runtime to maximize the chances of successfully reconstructing a proof for a rewrite while minimizing the amount of work spent on unsuccessful parts of the search space. Note that d is decremented only in recursive calls over the premises of conditional rules. For other recursive calls, which are over subterms of the input equality, termination is ensured by the reduction in the size of the new input equality.

The algorithm returns $\top$ if it finds a proof sketch for $t \approx s$ within the given depth restriction d. During its search, it updates a (global) proof sketch map P from term equalities to rules r that can be used in the final proof, or to pairs (fail, e) indicating that no proof for that equality can be found within

depth e. We use the array-like notation $\mathsf{P}[t \approx s]$ to refer to the value that P associates with $t \approx s$. A few of the rewrite rules stored in P are built-in, the rest are from the database $\mathcal{R}$. The built-in rules are provided in Figure 4 in the style of inference rules. Note that trans is actually not used for proof sketches, but only for the construction of final proofs.

Going through the algorithm line by line, we see that it first returns $\bot$ if the given depth d is negative. Then, on line 2, it checks if a proof sketch for $t \approx s$ has already been determined. If so and the value was (fail, e), then no proof was found for $t \approx s$ using depth e. If e is at least d, then it is impossible to construct a proof with depth d, and $\bot$ is returned to indicate failure. On the other hand, if a proof already exists, then $\top$ is returned, indicating success.

If none of these quick-return cases hold, the algorithm tries to prove the equality using several techniques, which we informally call proof *tactics*. First, the algorithm checks if the equality can be quickly (dis)proven. Specifically, on line 5 the simplest tactic checks whether the equality can be proven by *evaluation*, and returns $\top$ if so. We write $t{\downarrow_{\mathrm{e}}}$ to denote the *evaluated* form of t, typically a concrete constant c equivalent to t, if one can be determined by recursively evaluating (i.e., constant-folding) subterms of t, or t itself otherwise. If the evaluated form of t and s are the same, the algorithm stores in P the information that $t \approx s$ can be proven by evaluation, denoted by built-in rule eval. This case applies for instance to simple equalities such as $1 + 3 \approx 2 + 2$. On line 6, the global rewriter of an SMT solver (denoted as $\downarrow$) is used as an oracle to check whether the current equality can be rewritten to $\bot$, which means that the search for a proof sketch is futile. In this case, failure is stored as (fail, ∞), indicating that a proof for $t \approx s$ cannot exist because $\vDash_T t \not\approx s$. This is a fast albeit incomplete check which is useful when the input $t \approx s$ is a precondition of some other rule. If that check fails, the search continues because the global rewriter is incomplete, and thus a proof for $t \approx s$ may still exist. On line 7, $t \approx s$ is tentatively marked in P as (fail, d), but then an attempt is made to prove $t \approx s$ using the remaining proof tactics. The equality is marked as a failure *before* running these tactics to avoid infinite recursion when $t \approx s$ happens to be a premise in some recursive call.

Line 9 gives our tactic for proving the given equality by congruence, which we associate with a proof rule cong. If t and s have the same top symbol f and our reconstruction algorithm succeeds in proving equalities pairwise for each of their arguments $\vec{u} \approx \vec{v}$, we mark $t \approx s$ as proven and return $\top$. Line 12 gives our tactic for congruence plus evaluation, which we associate with a proof rule ceval. This tactic uses the global rewriter again as an oracle to check whether all the arguments $\vec{u}$ of t can be rewritten to some constant values $\vec{c}$, i.e., whether $\vec{u}{\downarrow} = \vec{c}$. If additionally the evaluation of the top symbol f on $\vec{c}$ is equal to the evaluation of s, then the algorithm tries to construct a proof for equalities $\vec{u} \approx \vec{c}$ using a recursive call. If it finds a proof, then $t \approx s$ is marked proven and $\top$ is returned. Failing this, the algorithm applies the main proof tactic, which checks whether there is a rule r in rewrite rule database $\mathcal{R}$ whose conclusion's left-hand side u matches t under some substitution σ. In this case, it calls itself recursively, attempting to prove that: (i) the right-hand side s is equivalent to u; and (ii) each premise of that rule holds in the theory under the same substitution. If both of these checks succeed, $t \approx s$ is marked as proven by rule r. Note that the matching does not automatically take into consideration the commutativity of operators. Instead, the algorithm relies on the commutativity of operators being expressed as additional rewrite rules.

Database Implementation. The algorithm is implemented by using a discrimination tree data structure to index the conclusions of all rules in $\mathcal{R}$. When a rule is added to $\mathcal{R}$, it is normalized so that its variables are taken from a global list and assigned based on a left-to-right traversal of the conclusion. For example, $x+y \approx y+x$ is normalized to $x_1+x_2 \approx x_2+x_1$, where the global list of integer variables is $(x_1, x_2, \ldots)$. We enumerate matches for $t \approx s$ based on a single traversal of the discrimination tree, which both constructs the matching substitution and ends at the rewrite rule identifier.

Optimizations and Extensions. Our actual algorithm includes several optimizations and extensions not shown in Figure 3. First, our tactics use a fast failure heuristic that avoids making recursive calls for a set of equalities $\vec{u} \approx \vec{v}$ if a single $u_i \approx v_i$ can be shown to fail without recursion. For example, our congruence tactic for $f(u, 0) \approx f(v, 1)$ fails early since $(0 \approx 1){\downarrow} = \bot$. Second, we extend our techniques for evaluation of arithmetic equalities to incorporate *polynomial normalization*, where, for example, the arithmetic term $y + x + x$ can be shown to be equal to $2 * x + y$. Third, we use additional built-in tactics for Booleans, e.g., that prove $(t \approx s) \approx \top$ if $t \approx s$ can be proven. Finally, we account for fixed-point rules from $\mathcal{R}$ (as described in Section III) by an extension to the tactic in line 15. In particular, when considering a fixed point rule r with conclusion $u \approx v$ that matches $t \approx s$ with substitution σ, we immediately check if the subterm of $\sigma(v)$ occurring at the placeholder position denoted by r also produces a match using the rule r. If so, we store the proof sketch for $t \approx \sigma(v)$ and continue this process until we have proven the equality $t \approx v'$ for some v'. We then attempt to prove $s \approx v'$ along with the required preconditions for the application(s) we used to derive $t \approx v'$.

B. From Proof Sketches to Proofs

We now return to the question of how to transform a proof sketch into a final proof. A proof is built out of *proof nodes*. A proof node is a triple $(\mathsf{r}, \vec{q}, \vec{t})$, where r is a proof rule identifier, $\vec{q}$ is a list of proof nodes, and $\vec{t}$ is a list of terms. A *proof checker* for a proof rule r is a function taking a list of formulas $\vec{\varphi}$ and a list of terms $\vec{t}$, and returning either a *conclusion* formula ψ or failure. Intuitively, the proof checker returns ψ if r concludes ψ from premises $\vec{\varphi}$ and a side condition depending on terms $\vec{t}$. A well-formed proof in a proof system $\mathcal{S}$ is a directed acyclic graph over proof nodes whose conclusions can be assigned based on the proof checkers for the rules in $\mathcal{S}$. In

particular, a proof node $(\mathsf{r}, \vec{q}, \vec{t})$ can be assigned a conclusion ψ if the proof nodes in $\vec{q}$ are well-formed with conclusions $\vec{\varphi}$ and the proof checker for r on $(\vec{\varphi}, \vec{t})$ returns ψ.

Overall, the algorithm in Figure 3 maintains the invariant that equalities $t \approx s$ map to a rule r by the proof sketch P only if entries for the preconditions $\vec{p}$ of rule r also have been successfully added to P, and moreover these dependencies are acyclic. Thus, we can transform the proof sketch P into a final proof by first recursively reconstructing the proofs of the preconditions to the current rule. For equalities $t \approx s$ marked with the eval rule, we construct a proof whose proof rule is reflexivity or evaluation. For equalities $f(\vec{u}) \approx f(\vec{v})$ marked with the cong rule, we first construct proofs for each of $\vec{u} \approx \vec{v}$, and then construct the proof of $f(\vec{u}) \approx f(\vec{v})$ by congruence. For equalities $f(\vec{u}) \approx s$ marked ceval, after reconstructing the proofs of $\vec{u} \approx \vec{c}$, we prove $f(\vec{u}) \approx f(\vec{c})$ by congruence, $f(\vec{c}) \approx s$ by evaluation, and then $f(\vec{u}) \approx s$ by transitivity of these two equalities using the trans rule from Figure 4. For equalities $t \approx s$ marked with a rule r from our database having conclusion $u \approx v$, we reconstruct the substitution σ such that $t = \sigma(u)$ by matching. We prove $t \approx \sigma(v)$ by rule r, which implies the existence of a proof of $\sigma(v) \approx s$ (due to the recursive call on line 16), and we finally combine them to a proof for $t \approx s$ by transitivity.

Example 1: Suppose we wish to prove the correctness of the rewrite $\mathsf{substr}(\mathsf{substr}(\texttt{"abc"}, 4, 1), m, n) \rightsquigarrow \texttt{""}$. Furthermore, assume our rewrite database $\mathcal{R}$ contains:

```
(define-cond-rule substr-empty-s (
    (s String) (m Int) (n Int))
  (= s "") (str.substr s m n) "")
```

We call the method rc from Figure 3 on the equality $\mathsf{substr}(\mathsf{substr}(\texttt{"abc"}, 4, 1), j, k) \approx \texttt{""}$ with a chosen depth $d = 3$. Assume that the proof sketch map P is initially empty. For this input, none of the conditions on lines 1-6 apply. On line 7, we provisionally set $\mathsf{P}[\mathsf{substr}(\mathsf{substr}(\texttt{"abc"}, 4, 1), j, k) \approx \texttt{""}]$ to $(\mathsf{fail}, 3)$. The conditions on lines 8 and 10 also do not apply. In the loop on line 14, we find that the match term $\mathsf{substr}(\texttt{""}, j, k)$ from rule substr–empty–s matches the left-hand side of our equality with substitution $\sigma = \{s \mapsto \mathsf{substr}(\mathsf{substr}(\texttt{"abc"}, 4, 1), m \mapsto j, n \mapsto k\}$. On lines 16 and 17, we recursively call rc on $(\sigma(\texttt{""}) \approx \texttt{""}, 2)$ and on $(\sigma(s \approx \texttt{""}), 2)$, respectively. Both recursive calls succeed trivially on line 5, where the latter equality is $\mathsf{substr}(\texttt{"abc"}, 4, 1) \approx \texttt{""}$. Thus, we successfully prove the conditions for applying substr–empty–s to our input equality. We denote this in P and return $\top$, where P is the mapping $\{\texttt{""} \approx \texttt{""} \mapsto \mathsf{eval}, \mathsf{substr}(\texttt{"abc"}, 4, 1) \approx \texttt{""} \mapsto \mathsf{eval}, \mathsf{substr}(\mathsf{substr}(\texttt{"abc"}, 4, 1), j, k) \approx \texttt{""} \mapsto \mathsf{substr–empty–s}\}$. The proof of the original equality can then be constructed trivially based on this mapping, where, overall, the proof involves an application of substr–empty–s whose premise is proven by eval.

V. Implementation

We implemented both a compiler for RARE and the reconstruction algorithm, and integrated them with CVC5 [4], a state-of-the-art SMT solver, most of which is instrumented to produce proofs [6]. Notably, the rewriter is not instrumented, so proof reconstruction is an attractive option for CVC5. Our initial implementation focuses on the theory of strings, both because it is used in practical applications such as reasoning about access policies in the cloud [2], and because it presents a challenge due to the large number of complex rules in the strings theory rewriter, which are required to achieve good performance [21]. The theory of strings is frequently combined with the theory of linear integer arithmetic to reason about the length and indices of strings. Thus, reconstructing rewrite proofs for string problems requires reasoning about Boolean, linear integer arithmetic, and string terms. None of these theories require parameterized sorts, so the current implementation uses concrete types. Supporting rewrite rules with partially specified types is left for future work.

In the following, we discuss the integration of our approach in the existing proof infrastructure and our experience using RARE to define a set of rewrite rules. We implemented our reconstruction algorithm as a module in the existing proof infrastructure of CVC5. At compile-time, our compiler for RARE populates the rewrite rule database (referred to as $\mathcal{R}$ in the previous section). As mentioned earlier, RARE aims at being a compromise between succinctness and expressiveness. The limited expressiveness of RARE means that some desirable rewrite rules cannot be expressed in it. To overcome this limitation, our reconstruction module supports mixing RARE rules with rules implemented in C++. We use this feature, for example, for certain integer arithmetic rewrites, as discussed below. Reconstructing the proofs for rewrites happens during post-processing of the overall proof. If a proof for a given atomic rewrite cannot be reconstructed, a generic *theory rewrite* proof rule is used instead.

The proof module of CVC5 supports the production of proof certificates in different proof formats. One of the proof formats that is well-supported is LFSC [24]. Proofs in LFSC use the same language to define both the proof rules and the proofs themselves. As part of our implementation, we extended CVC5's LFSC back end to automatically generate LFSC proof rules for each rewrite that appears in a given proof.

The string theory rewriter in CVC5 is complex—its implementation, not including any of the helper functions, amounts to over 3,000 lines of C++ code and distinguishes over 200 different rewrite rules. Moreover, not all of those rules can be expressed as a single rewrite rule in RARE. In view of these difficulties, we took a pragmatic approach to proof reconstruction for the theory of strings: instead of trying to implement all of the rewrite rules in RARE, we focused on a set of challenging string benchmarks (see Section VI) of practical interest, and then defined rules on demand to fill in missing subproofs. We ended up with 40 RARE rules for the theory of strings.

The structure of the CVC5 theory rewriter for arithmetic, on the other hand, is quite different. Instead of a large number of different rewrite rules, most of the rewriting boils down to normalizing polynomials. Thus, for normalizing polynomials

we implemented a single rule, which is complemented with 25 rules for arithmetic that do not concern this normalization.

Finally, the rewriter for Booleans is far simpler than rewriters for other theories—its implementation is less than 350 lines of C++ code. For reconstructing Boolean rewrite rules, we took a similar approach to the one for string rewrites and defined RARE rules on demand to fill in missing subproofs on problems of interest. This led to 22 Boolean rules in RARE.

While using RARE is not possible or desirable for all rewrite rules, it did enable us to iterate quickly to cover the majority of missing subproofs for our target benchmarks.

VI. EVALUATION

Using our implementation in CVC5, we evaluated the following research questions:

- Can we generate fine-grained proofs for rewrites?
- What is the performance impact of generating fine-grained proofs?

We considered two benchmark sets, both over the theory of strings. The first consists of 25 unsatisfiable industrial benchmarks that are representative of challenging queries in a specific production environment. The second set consists of 26,626 unsatisfiable benchmarks from the logics `QF_S` and `QF_SLIA` in the SMT-LIB benchmark library. To determine the set of unsatisfiable benchmarks, we used the results from an artifact [3] of an earlier evaluation of CVC5, which ran the competition configuration of CVC5 for 1200s.

For the evaluation, we ran all benchmarks with three configurations of CVC5: CVC5, which does not generate any proofs; CVC5-C, which generates proofs with coarse-grained steps for rewrites; and CVC5-F, which uses our approach to generate fine-grained proofs for rewrites. For the proof reconstruction, we set the depth d to 3. The configurations involved in our evaluation are all variants of CVC5 since to the best of our knowledge, no other SMT solvers generate proofs for nontrivial theory rewrites. In particular, no other solver can generate fine-grained proofs for the theory of strings.

We ran all experiments on a cluster equipped with Intel Xeon E5-2620 v4 CPUs running Ubuntu 16.04. We allocated one physical CPU core and 8GB of RAM for each solver-benchmark pair and used a 900 seconds time limit.

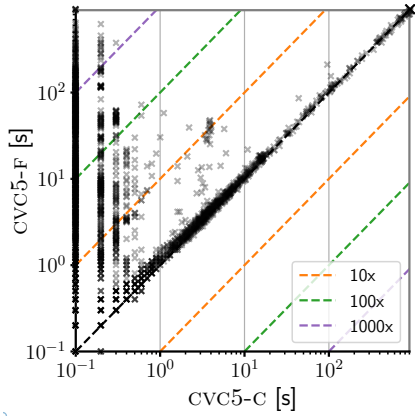
To measure the effectiveness of our reconstruction, we analyzed the generated proofs of benchmarks that were solved by all configurations. The proofs for the industrial benchmarks contain 43,819 rewrite steps, and the proofs for the SMT-LIB benchmarks contain 2,806,761. For those steps, CVC5-F reconstructed fine-grained proofs in terms of our current rewrite rule database for 95% of the rewrite steps for the industrial set, and for 92% of the rewrite steps for SMT-LIB. The lower rate in SMT-LIB can be explained by our greater focus on the rewrite steps from proofs of the industrial benchmarks. We expect that the SMT-LIB rate can be improved to the level of the industrial set without significant challenges, i.e., primarily by adding more rules to the rewrite rule database. We also note that for 20% (5 out of 25) benchmarks in the industrial set, CVC5-F manages to produce fine-grained proofs for *all* rewrites, whereas for SMT-LIB, 22% of CVC5-F's proofs with rewrite steps (5,945 out of 26,418) are fully fine-grained.

TABLE I: Number of solved benchmarks and cumulative solving times in seconds on commonly solved benchmarks, with the slowdown versus CVC5-C in parentheses.

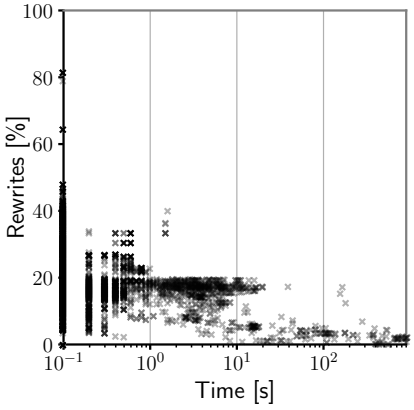
Division		CVC5	CVC5-C	CVC5-F	
Industrial (25)	Solved	25	25	25	
	Time	238	715	779	(1.09×)
SMT-LIB (26,626)	Solved	26,615	26,614	26,609	
	Time	34,028	35,932	114,330	(3.18×)
Total (26,651)	Solved	26,640	26,639	26,634	
	Time	34,266	36,647	115,109	(3.14×)

Table I summarizes the overhead incurred by our approach grouped by benchmark set. Figure 6 shows a cactus plot that compares the performance of the different configurations. In this experiment, we use CVC5 as a reference point to measure the general overhead of proof production, and to compare that overhead with the additional overhead of generating fine-grained proofs. Table I shows that the overhead on the industrial benchmarks for generating proofs is significant, but the additional overhead of generating the fine-grained proofs is negligible. For the benchmarks from SMT-LIB, the opposite is the case: the overhead for generating coarse-grained proofs is relatively small, but the overhead of generating fine-grained proofs is significant. For a better understanding of the origin of the overhead, we provide three scatter plots in Figure 5. Figure 5a compares the performance of CVC5-C with the performance of CVC5-F and shows that for benchmarks that are solved quickly with CVC5-C, there are cases where the overhead of the proof reconstruction is significant. For longer running benchmarks, the overhead seems to be less pronounced. In Figure 5b, we plot the solving time in relationship with the relative number of atomic rewrites in proofs generated by CVC5-C. The plot shows that atomic rewrites are featured more prominently in proofs of benchmarks that are solved quickly. This may explain part of the overhead for easy benchmarks: a larger portion of the proof is being post-processed with the reconstruction algorithm. Finally, Figure 5c shows the relationship between the difference in solving time between CVC5-F and CVC5-C and the number of atomic rewrites. The plot indicates two trends: more atomic rewrites lead to more overhead and—more surprisingly—there seems to be a large number of benchmarks with a relatively small number of rewrites that have a significant amount of overhead.

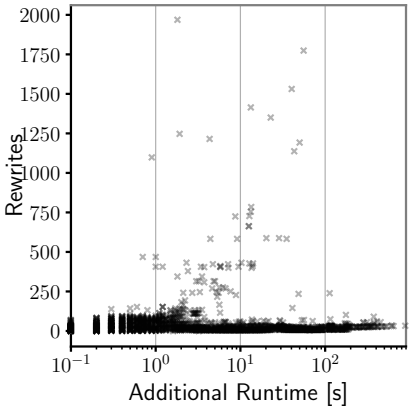
Overall, we find that our approach does not significantly affect the number of solved benchmarks. Additionally, it works well for the industrial use case that we originally targeted with our approach. Some of the SMT-LIB benchmarks, on the other hand, make use of complex rewrites such as the ones described in earlier work [21], which we have not explicitly optimized our current implementation for.



(a) Scatter plot that compares the performance of CVC5-C and CVC5-F.



(b) Scatter plot of the relationship between solving time for CVC5-C and the number of atomic rewrites.



(c) Scatter plot of the relationship between overhead of the rewrite proof reconstruction and the number of rewrites.

Fig. 5: Scatter plots that analyze the overhead of our rewrite proof reconstruction.

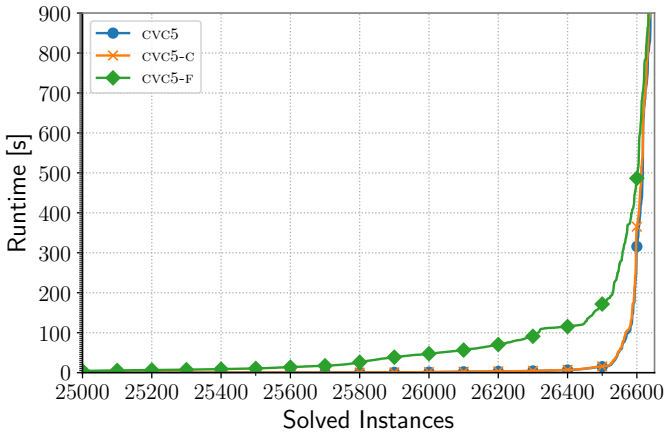


Fig. 6: Cactus plot that shows the general performance impact of generating proofs and the performance impact of generating fine-grained proofs for rewrites.

VII. Conclusion

We presented a DSL-based approach for reconstructing fine-grained proofs of rewrite rules. For the future, we plan to expand our implementation to other theories in CVC5, including theories with parameterized sorts, which will require adding support for gradual typing. The DSL proposed in this work is independent of the discussed use case and can be used to express rewrite rules for SMT solvers in other contexts.

Another direction for future work is to expand the DSL compiler to generate efficient code to replace parts of existing theory rewriters, i.e., code that actually performs the rewrites. This could make it much easier to explore different sets of rewrite rules. It would also make the rewriting code easier to understand and maintain. However, since the rewriter is called frequently during solving, its performance is critical. Therefore, integrating automatically generated code needs to be done carefully. Our primary targets in that context are the theories of fixed-size bit-vectors and floating-point arithmetic.

Another back end for the DSL could be used to generate verification conditions for the T-validity of rewrite rules. These conditions could be discharged using a third-party tool such as a proof assistant or another SMT solver. An interesting challenge here is that SMT solvers generally only support reasoning about fixed-size bit-vectors, whereas rewrite rules for the theory of bit-vectors are parameterized by the bit-width. We plan to explore approaches for bit-width independent verification (e.g., [18]) to discharge these verification conditions.

References

[1] J. Backes, U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. S. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan. Stratified abstraction of access control policies. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2020.

[2] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. S. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *FMCAD*, pages 1–9. IEEE, 2018.

[3] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. M. Y. Mohamed, A. Niemetz, A. Noetzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. Artifact for Paper cvc5: A Versatile and Industrial-Strength SMT Solver, Nov. 2021.

[4] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[5] H. Barbosa, J. C. Blanchette, M. Fleury, and P. Fontaine. Scalable fine-grained proofs for formula processing. *J. Autom. Reason.*, 64(3):485–510, 2020.

[6] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri, Y. Zohar, C. Tinelli, and C. W. Barrett. Flexible proof production in an industrial-strength SMT solver. In J. Blanchette, L. Kovács, and D. Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2022.

[7] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[8] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.

[9] P. Borovanský, C. Kirchner, H. Kirchner, and P. Moreau. ELAN from a rewriting logic point of view. *Theor. Comput. Sci.*, 285(2):155–185, 2002.

[10] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta, C. Schlesinger, C. Stephens, C. Varming, and A. Warfield. Block public access: trust safety verification of access control policies. In P. Devanbu, M. B. Cohen, and T. Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 281–291. ACM, 2020.

[11] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In R. A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[13] B. Cook. Formal reasoning about the security of amazon web services. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2018.

[14] R. Diaconescu and K. Futatsugi. *Cafeobj Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[15] H. Enderton and H. B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[16] W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Autom. Reason.*, 9(2):147–167, 1992.

[17] A. Niemetz, M. Preiner, and C. W. Barrett. Murxla: A modular and highly extensible API fuzzer for SMT solvers. In S. Shoham and Y. Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2022.

[18] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. Barrett, and C. Tinelli. Towards satisfiability modulo parametric bit-vectors. *Journal of Automated Reasoning*, 65(7):1001–1025, Oct. 2021.

[19] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[20] A. Nötzli. *Towards better simplifications in SMT solvers with applications in string solving*. PhD thesis, Stanford University, 2021.

[21] A. Reynolds, A. Nötzli, C. W. Barrett, and C. Tinelli. High-level abstractions for simplifying extended string constraints in SMT. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2019.

[22] H. Schurr, M. Fleury, H. Barbosa, and P. Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). *CoRR*, abs/2107.02354, 2021.

[23] H. Schurr, M. Fleury, and M. Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In A. Platzer and G. Sutcliffe, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 450–467. Springer, 2021.

[24] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. Smt proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[25] T. C. D. Team. The coq proof assistant, Jan. 2022.

[26] C. Tinelli and C. G. Zarba. Combining decision procedures for sorted theories. In J. J. Alferes and J. Leite, editors, *Logics in Artificial Intelligence*, pages 641–653, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.