Syntax-Guided Rewrite Rule Enumeration for SMT Solvers

Andres Nötzli[1], Andrew Reynolds[2],
Haniel Barbosa[2], Aina Niemetz[1], Mathias Preiner[1],
Clark Barrett[1], and Cesare Tinelli[2]

[1] Stanford University, Stanford, USA
[2] University of Iowa, Iowa City, USA

Abstract. The performance of modern Satisfiability Modulo Theories (SMT) solvers relies crucially on efficient decision procedures as well as static simplification techniques, which include large sets of rewrite rules. Manually discovering and implementing rewrite rules is challenging. In this work, we propose a framework that uses enumerative syntax-guided synthesis (SyGuS) to propose rewrite rules that are not implemented in a given SMT solver. We implement this framework in CVC4, a state-of-the-art SMT and SyGuS solver, and evaluate several use cases. We show that some SMT solvers miss rewriting opportunities, or worse, have bugs in their rewriters. We also show that a variation of our approach can be used to test the correctness of a rewriter. Finally, we show that rewrites discovered with this technique lead to significant improvements in CVC4 on both SMT and SyGuS problems over bit-vectors and strings.

1 Introduction

Developing state-of-the-art Satisfiability Modulo Theories (SMT) solvers is challenging. Implementing only basic decision procedures is usually not enough, since in practice, many problems can only be solved after they have been simplified to a form for which the SMT solver is effective. Typically, such simplifications are implemented as a set of *rewrite rules* applied by a solver component that we will call the *rewriter*. Depending on the theory, optimizing the rewriter can be as important as optimizing the decision procedure. Designing an effective and correct rewriter requires extensive domain knowledge and the analysis of many specific problem instances. New rewrite rules are often only introduced when a new problem requires a particular simplification.

SMT rewriters also have other applications such as accelerating enumerative syntax-guided synthesis (SyGuS). The SyGuS Interchangeable Format [6] uses a subset of the theories defined in SMT-LIB [11] to assign meaning to pre-defined function symbols. Because a term t is equivalent to its *rewritten form* $t{\downarrow}$, an enumerative SyGuS solver can limit its search space to rewritten terms only. This significantly prunes the search.

Rewriters are typically developed manually by SMT experts. Because this process is difficult and error-prone, automating parts of this process is extremely useful. In this paper, we propose a partially automated workflow that increases the productivity of SMT solver developers by systematically identifying rewriting opportunities that the solver misses. We leverage the common foundations of SyGuS and SMT solving to

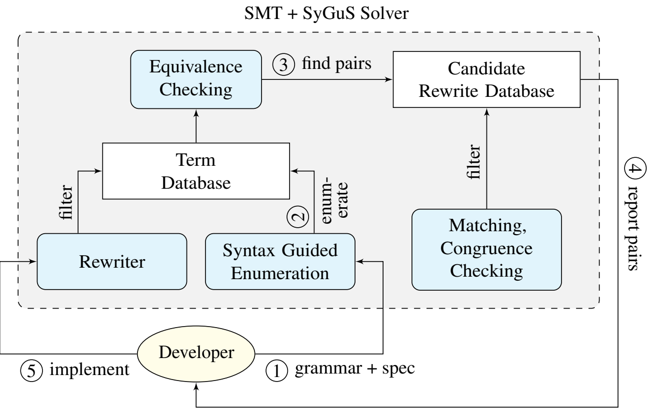

Fig. 1: Overview of our workflow for the development of rewriters.

guide developers in the analysis and implementation of the rewriter, independently of the theory under consideration.

Figure 1 shows an overview of our proposed workflow. In Step 1, the developer provides a *grammar* and a *specification* as inputs (see Section 5 for examples). This input describes the class of terms that the developer is interested in targeting in the rewriter. In Step 2, we use previous techniques (implemented in the SyGuS module of the SMT solver CVC4 [9]) to efficiently enumerate target terms into a *term database*. In Steps 3 and 4, we pair those terms together to form a *candidate rewrite database*. A subset of this database is then reported to the developer as a set of *unoriented* pairs $t_1 \approx s_1, \ldots, t_n \approx s_n$ with the following key properties:

1. Terms t_i and s_i were inferred to be equivalent based on some criteria in Step 3.
2. $t_i\!\downarrow$ is not equal to $s_i\!\downarrow$; i.e., the rewriter does not treat t_i and s_i as equivalent.

We can interpret these pairs as rewrites by picking an orientation ($t_i \leadsto s_i$ or $s_i \leadsto t_i$) and interpreting the free variables on the left-hand side as place-holders that match arbitrary subterms. For example, $x + 0 \approx x$ can be interpreted as a rewrite $x + 0 \leadsto x$ that rewrites matching terms such as $y \cdot z + 0$ to $y \cdot z$.

The set of pairs can be understood as a *to do* list of rewrites that are currently missing in the rewriter. In Step 5, based on this list, the developer can extend the rewriter to incorporate rules for these unhandled equivalences. We have found that our workflow is most effective when candidate rewrite rules act as hints to inspire developers, who, through creativity and careful engineering, are subsequently able to improve the rewriter. In our experience, this workflow leads to a positive feedback loop. As the developer improves the rewriter, term enumeration produces fewer redundant terms, thus yielding more complex terms on the next iteration. As a result, the candidate rewrite rules become more complex as well. Since our workflow is *partially* automated, there are no restrictions on the complexity of the rewrites implemented. As we illustrate in Section 5.1, some rewrites can only be defined with an expressive language, and fully automated approaches tend to restrict themselves to a relatively simple form of rewrites.

Contributions

- We present a novel theory-independent workflow for generating candidate rewrite rules, based on a user-provided grammar, that uses the solver under development as an oracle for determining the missing rules.
- We propose techniques for efficient equivalence checking between terms and for filtering to reduce redundancy in the candidate rewrite rules.
- We introduce several metrics for measuring coverage of the rewrite rules implemented by SMT solvers and demonstrate their impact in practice.
- We show that our workflow is highly effective both for discovering shortcomings in state-of-the-art SMT solvers, and as means of improving the performance of our own solver CVC4 on a wide range of SyGuS and SMT benchmarks.

We discuss our approach for choosing suitable grammars (Step 1) in Section 5.1. We describe Steps 2-4 of Figure 1 in Sections 2, 3, and 4 respectively. In Section 5, we report on our experience with the proposed workflow and discuss using the workflow to test other SMT solvers and to improve our confidence in our implementation. Finally, we evaluate the impact of the rewrites on solving performance. We discuss related work in Section 6 and future work in Section 7.

Preliminaries We use standard notions of typed (higher-order) logic, including formula, term, quantifier, etc. We recall a few definitions here. A *substitution* $\sigma = \{\bar{x} \mapsto \bar{t}\}$ is a finite map from variables to terms of the same type, and $s\sigma$ denotes the result of replacing all occurrences of $\bar{x}$ in s by $\bar{t}$. A *theory* T is a pair (Σ, I), where Σ is a signature (a set of types and function symbols), and I is a set of Σ-interpretations, called the *models* of T. We assume that Σ contains the equality relation $\approx$ for all types τ from Σ, interpreted as the identity relation. A formula is T-*satisfiable* (resp., T-*valid*) if it is satisfied by some (resp., all) models of T. Terms t_1 and t_2 are T-*equivalent*, written $t_1 \approx_T t_2$, if the formula $t_1 \approx t_2$ is T-valid.

2 Syntax-Guided Synthesis for Term Enumeration

Step 2 of the workflow in Figure 1 is to enumerate terms from a given grammar. For this task, we leverage previous work on enumerative approaches to the SyGuS problem [25].

Syntax-Guided Synthesis A *SyGuS problem* for an n-ary (first-order) function f in a background theory T consists of: *(i)* a set of *syntactic restrictions*, given by a grammar $\mathcal{R}$; and *(ii)* a set of *semantic restrictions*, a *specification*, given by a T-formula of the form $\exists f. \forall \bar{x}. \varphi[f, \bar{x}]$, where φ is typically a quantifier-free formula over the (second-order) variable f and the first-order variables $\bar{x} = (x_1, ..., x_n)$. A grammar $\mathcal{R}$ consists of an initial symbol s_0, a set S of *non-terminal symbols*, where $s_0 \in S$, and a set R of rules $s \to t$, with $s \in S$, and where t is a term built from symbols of S, free variables, and symbols in the signature of T—with the latter two acting as *terminal symbols*. The set of rules R defines a rewrite relation over terms s and t, denoted by $\to$. A term t is *generated* by $\mathcal{R}$ if $s_0 \to^* t$, where $\to^*$ is the reflexive-transitive closure of $\to$ and t contains no symbols from S. A *solution* for f is a closed lambda term $\lambda\bar{y}.\, t$ of the same type as f such that t is generated by $\mathcal{R}$ and $\forall \bar{x}. \varphi[\lambda\bar{y}.\, t, \bar{x}]$ is T-valid.

An *enumerative* SyGuS solver consists of a *candidate solution generator*, which produces a stream of terms t_i in the language generated by grammar $\mathcal{R}$, and a *verifier*, which, given a candidate solution t_i, checks whether $\forall \bar{x}.\ \varphi[\lambda \bar{y}.\ t_i, \bar{x}]$ is T-valid. The solver terminates as soon as it generates a candidate solution t_i that the verifier accepts. In practice, most state-of-the-art enumerative SyGuS solvers [33, 7] are implemented on top of an SMT solver with support for quantifier-free T-formulas, which can be used as a verifier in this approach. Our solver CVC4 acts as both the candidate solution generator and the verifier [26].

Generating Rewrite Rules With SyGuS In the context of our proposed workflow, an enumerative SyGuS solver can be used as a term generator in Step 2 from Figure 1. In particular, it can be used to produce *multiple* solutions to the SyGuS problem specified in Step 1, where each solution is added to the term database. Recall that free variables in the solutions are interpreted as place-holders for arbitrary subterms. Thus, the arguments of f determine the number and types of the place-holders in the generated rewrites. A SyGuS specification $\exists f. \forall \bar{x}. \varphi[f, \bar{x}]$ acts as a filtering mechanism to discard terms that should not be included in rewrite rules. If we do not wish to impose any semantic restrictions, the common case in our workflow, we can use the specification $\exists f.\ \top$, which causes the enumeration of all terms that meet the syntactic restrictions.

To achieve high performance, SyGuS solvers implement techniques that limit the enumeration of equivalent candidate solutions in order to prune the search space. The rationale for this is that if $\forall \bar{x}.\ \varphi[\lambda \bar{y}.\ t_1, \bar{x}]$ is not T-valid for a candidate t_1, then it is fruitless to consider any candidate t_2 if $t_1 \approx_T t_2$ since $\forall \bar{x}.\ \varphi[\lambda \bar{y}.\ t_2, \bar{x}]$ will not be T-valid either. CVC4 uses its own rewriter as an (incomplete but fast) T-equivalence oracle, where the syntactic equality of $t_1\!\downarrow$ and $t_2\!\downarrow$ implies, by the soundness of the rewriter, that $t_1 \approx_T t_2$ [27]. When the SyGuS solver of CVC4 discovers the equivalence of two terms t_1 and t_2, it generates constraints to ensure that subsequent solutions only include either t_1 or t_2 as a subterm. As a consequence, since CVC4 never generates two candidate solutions that are identical up to rewriting, a better rewriter leads to a more efficient term enumeration. This also ensures that the term database generated in Step 2 in Figure 1 contains no distinct terms s and t such that $s\!\downarrow = t\!\downarrow$, which, in turn, also ensures that no existing rewrites are considered as candidates.

3 Equivalence Checking Techniques for Rewrite Rules

In Step 3 in Figure 1, we are given a database of terms, which may contain free variables $\bar{y}$ corresponding to the arguments of the function to be synthesized. The term database, a set of terms D, is generated by syntax-guided enumeration. From this set, we generate a set of (unoriented) pairs of the form $t_1 \approx s_1, \ldots, t_n \approx s_n$ where for each $i = 1, \ldots, n$, terms t_i and s_i are in D and meet some criterion for equivalence. We call such pairs *candidate rewrite rules*.

Our techniques apply to any background theory with a distinguished set of *values* for each of its types, i.e., variable-free terms for denoting the elements of that type, e.g. (negated) numerals $(-)n$ for an integer type. We assume that the initial rewriter reduces any variable-free term to a value, e.g., $4 - 5$ to -1. These assumptions hold for the initial rewriter and the theories in our evaluation.

A naïve way to find candidate rewrite rules is to consider each pair of distinct terms $s[\bar{y}], t[\bar{y}] \in D$, check the satisfiability of $\exists \bar{y}.\, t \not\approx s$, and, if unsatisfiable, include $t \approx s$ as a candidate. However, this can be inefficient and may even be infeasible for some theories.[3] To mitigate these issues, we have developed techniques based on evaluating terms on a set of *sample points*, i.e. tuples of values $\bar{c}$ of the same type as $\bar{y}$ above. We describe those techniques below.

We compute an equivalence relation E over the terms in our (evolving) term database D. Two terms t_i, t_j are related in E if for every sample point $\bar{c}$ they have the same evaluation, i.e. $(t_i\{\bar{y} \mapsto \bar{c}\})\!\downarrow = (t_j\{\bar{y} \mapsto \bar{c}\})\!\downarrow$. While equivalence in E does not entail T-equivalence, terms disequivalent in E are guaranteed to be T-disequivalent. To see how E evolves, let $\{r_1, \ldots, r_n\} \subseteq D$ be a set of *representative* terms from the equivalence relation E containing exactly one term from each equivalence class. For each new term t added to D, we either *(i)* determine t is equivalent to some r_i, output $t \approx r_i$ as a candidate rewrite rule, and add t to the equivalence class of r_i; or *(ii)* determine t is not equivalent to any of $r_1, \ldots, r_n$, i.e. for each r_i there is at least one sample point on which the evaluations differ, and add $\{t\}$ as an equivalence class. Thus, each equivalence class $\{t_1, \ldots, t_n\}$ of E is such that, for each $i = 1, \ldots, n$, a pair of the form $t_i \approx t_j$ has been generated for some $j \neq i$. In other words, E is the transitive closure of the set of pairs generated so far.

To optimize the evaluation of terms on sample points we rely on a *lazy evaluation trie*. This data structure maintains a list $P = [\bar{c}_1, \ldots, \bar{c}_n]$ of sample points, all of the same type as $\bar{y}$. It indexes a term t by its *evaluation sequence* on the set of sample points P, i.e., term t is indexed by a sequence of the form $[(t\{\bar{y} \mapsto \bar{c}_1\})\!\downarrow, \ldots, (t\{\bar{y} \mapsto \bar{c}_n\})\!\downarrow]$. Due to our assumptions, each term in this list is a value. For example, if $\bar{y} = (y_1, y_2)$ and $P = [(0, 1), (3, 2), (5, 5)]$, then the term $y_1 + 1$ is indexed by the list $[1, 4, 6]$. When a new term t is added to D, it is evaluated on each of the points in P. If the resulting sequence is already in the trie, t is added to the equivalence class of the term indexed by that sequence. If not, the new sequence is added to the trie and t becomes a singleton equivalence class. This guarantees that each representative from E is indexed to a different location in this trie. The technique can be made more efficient by performing certain evaluations *lazily*. In particular, it is sufficient to only use a *prefix* of the above sequence, provided that the prefix suffices to show that t is distinct from all other terms in the trie. For the previous example, if $y_1 + 1$ and $y_1 + y_2$ were the only two terms in the trie, they would be indexed by $[1, 4]$ and $[1, 5]$ respectively, since the second sample point $(3, 2)$ shows their disequality. We now discuss our different equivalence checking criteria by the way the sample points P are constructed.

Random Sampling A naïve method for constructing P is to choose n points at random. For that, we have implemented a random value generator for each type we are interested in. For Booleans and fixed-width bit-vectors, it simply returns a uniformly random value in the (finite) range. For integers, we first pick a sign and then iteratively concatenate digits $0 - 9$, with a fixed probability to terminate after each iteration. For rationals, we pick a fraction c_1/c_2 with integer c_1 and non-zero integer c_2 chosen at random. For strings, we concatenate a random number of characters over a fixed alphabet that includes all characters occurring in a rule of $\mathcal{R}$ and dummy character(s) for ensuring that the

[3] E.g. for checks in the theory of strings with length whose decidability is unknown [17].

cardinality of this alphabet is at least two. The latter is needed because, for instance, if the alphabet contained only one letter, it would be impossible to generate witnesses that disprove equivalences such as $\texttt{contains}(\texttt{"A"} \texttt{ ++ } x \texttt{ ++ } \texttt{"A"}, \texttt{"AA"}) \approx \texttt{true}$ where $\texttt{++}$ denotes string concatenation and $\texttt{contains}(t, s)$ is true if s is a substring of t.

Grammar-Based Sampling While random sampling is easy to implement, it is not effective at generating points that witness the disequivalence of certain term pairs. Thus, we have developed an alternative method that constructs points based on the user-provided grammar. In this method, each sample point in P (of arity n) is generated by choosing random points $\bar{c}_1, \ldots, \bar{c}_n$, random terms $t_1, \ldots, t_n$ generated by the input grammar $\mathcal{R}$, and then computing the result of $((t_1\{\bar{y} \mapsto \bar{c}_1\})\!\downarrow, \ldots, (t_n\{\bar{y} \mapsto \bar{c}_n\})\!\downarrow)$. The intuition is that sample points of this form are biased towards interesting values. In particular, they are likely to include non-trivial combinations of the user-provided constants that occur in the input grammar. For example, if the grammar contains `++`, `"A"`, and `"B"`, grammar-based sampling may return samples such as `"BA"` or `"AAB"`.

Exact Checking with Model-based Sampling In contrast to the previous methods, this method makes two terms equivalent only if they are T-equivalent. It is based on satisfiability checking and *dynamic, model-based* generation of a set of sample points P, which is initially empty. When a term t is generated, we check if it evaluates to the same values on all sample points in P as any previously generated term s. If so, we separately check the T-satisfiability of $t \not\approx s$. If $t \not\approx s$ is unsatisfiable, we put t and s in the same equivalence class. Otherwise, $t \not\approx s$ is satisfied by some model $\mathcal{M}$, and we add $\mathcal{M}(\bar{y})$ as a new sample point to P, guaranteeing that s and t evaluate differently on the updated set. The new sample point remains in P for use as other terms are generated. For example, $x + 1 \not\approx x$ is satisfied by $x = 1$, so $x = 1$ is added to P.

4 Filtering Techniques for Rewrite Rules

For developers, it is desirable for the workflow in Figure 1 to identify *useful* rewrites. For instance, it is desirable for the set of candidate rewrite rules to omit trivial consequences of other rules in the set. A rewrite rule $t \approx s$ is *redundant* with respect to a set $\{t_1 \approx s_1, \ldots, t_n \approx s_n\}$ of equations with free variables from $\bar{y}$ if $\forall \bar{y}.\, (t_1 \approx s_1 \wedge \ldots \wedge t_n \approx s_n)$ entails $\forall \bar{y}.\, t \approx s$ in the theory of equality. Redundant rules are not useful to the user, since they typically provide no new information. For example, if the framework generates $x \approx x + 0$ as a candidate rewrite rule, then it should not also generate the redundant rule $x \cdot y \approx (x + 0) \cdot y$, which is entailed by the former equality. Checking this entailment with a solver, however, is expensive, since it involves first-order quantification. Instead, we use several incomplete but sound and efficient filtering techniques. These techniques significantly reduce the number of rewrite rules printed (as we show empirically in Section 5.2).

Filtering Based on Matching One simple way to detect whether a rewrite rule is redundant is to check if it is an instance of a previously generated rule. For example, $y_1 + 1 \approx 1 + y_1$ is redundant with respect to any set that includes $y_1 + y_2 \approx y_2 + y_1$. Our implementation caches in a database all representative terms generated as a result of our equivalence checking. For each new candidate rewrite rule $t \approx s$, we query this

database for a set of *matches* of the form $t_1\sigma_1, \ldots, t_n\sigma_n$ where for each $i = 1, \ldots, n$, we have that t_i is a previous term added to the structure, and $t_i\sigma_i = t$. If for any such i, we have that $t_i \approx s_i$ was a previously generated candidate rewrite rule and $s_i\sigma_i = s$, we discard $t \approx s$.

Filtering Based on Variable Ordering This technique discards rewrite rules whose variables are not in a given canonical order. For example, assume a grammar $\mathcal{R}$ with free variables y_1, y_2 ordered as $y_1 \prec y_2$. Furthermore, assume that y_1 and y_2 are indistinguishable in $\mathcal{R}$ in the sense that if $\mathcal{R}$ has a rule $s \to t$, then it also has rules $s \to t\{y_1 \mapsto y_2\}$ and $s \to t\{y_2 \mapsto y_1\}$. In this case, we can discard candidate rewrite rules $t_1 \approx t_2$ where y_2 appears before y_1 in (say, left-to-right, depth-first) traversals of both t_1 and t_2. For example, we can pre-emptively discard the rewrite $y_2 + 0 \approx y_2$, without needing to apply the above filtering based on matching, by reasoning that we will eventually generate $y_1 + 0 \approx y_1$.

Filtering Based on Congruence Another inexpensive way to discover that a rewrite rule is redundant is to verify that it can be deduced from previous ones by congruence. For example, for any function f, it is easy to check that $f(y_1 + 0) \approx f(y_1)$ is redundant with respect to a set of rules containing $y_1 + 0 \approx y_1$. Our implementation maintains a data structure representing the congruence closure $\mathcal{C}(S)$ of the current set S of rewrite rules, i.e., the smallest superset of S closed under entailment in the theory of equality. Then, it discards any new candidate rewrite rule if it is already in $\mathcal{C}(S)$.

5 Evaluation

We now discuss our experience with the proposed workflow, evaluate different configurations, and show their impact on benchmarks. We ran experiments on a cluster equipped with Intel E5-2637 v4 CPUs running Ubuntu 16.04. All jobs used one core and 8 GB RAM unless otherwise indicated.

5.1 Experience

We implemented our framework for enumerating rewrite rules in CVC4, a state-of-the-art SMT solver. We used four grammars (Section 5.2), over strings (using the semantics of the latest draft of the SMT-LIB standard [32]), bit-vectors, and Booleans, as inputs to our framework. The grammar determines the operators and the number and types of different subterms that can be matched by the generated rewrites. To generate rewrites to accelerate the search for a particular SyGuS problem, we can simply use the grammar provided by the problem. To generate more general rewrites, it is helpful to divide the problem into rewrites for function symbols and rewrites for predicate symbols, as each of these formulations will use a different return type for f, the function to synthesize. We typically picked a small number of arguments for f, using types that we are interested in. In practice, we found two arguments to be sufficient to generate a large number of interesting rewrites for complex types. Rewrites with more variables on the left side are rarer than rewrites with fewer because they are more general. Thus, increasing the number of arguments is often not helpful and results in extra overhead.

```
(synth-fun f ((x String)
 (y String) (z Int)) String (
 (Start String (x y "A" "B" ""
  (str.++ Start Start)
  (str.replace Start Start Start)
  (str.at Start ie) (int.to.str ie)
  (str.substr Start ie ie)))
 (ie Int (0 1 z (+ ie ie) (- ie ie)
  (str.len Start) (str.to.int Start)
  (str.indexof Start Start ie)))))
```

```
(synth-fun f ((s (BitVec 4))
  (t (BitVec 4))) (BitVec 4) (
 (Start (BitVec 4) (
  s t #x0
  (bvneg  Start) (bvnot  Start)
  (bvadd  Start Start) (bvmul Start Start)
  (bvand  Start Start) (bvor  Start Start)
  (bvlshr Start Start)
  (bvshl  kStart Start)))))
```

Fig. 2: Examples of grammars used in our workflow: strterm (left), bvterm_4 (right).

Guided by the candidates generated using these grammars, we wrote approximately $5{,}300$ lines of source code to implement the new rewrite rules. We refer to this implementation as the *extended rewriter* (ext), which can be optionally enabled. The rewrites implemented in ext are a superset of the rewrites in the default rewriter (std). Our implementation is public [3]. We implemented approximately 80 classes of string rewrites with a focus on eliminating expensive operators, e.g. $\texttt{contains}(\texttt{replace}(x, y, z), z) \rightsquigarrow \texttt{contains}(x, y) \vee \texttt{contains}(x, z)$, where $\texttt{replace}(x, y, z)$ denotes the string obtained by replacing the first occurrence (if any) of the string y in x by z. A less intuitive example is $\texttt{replace}(x, \texttt{replace}(x, y, x), x) \rightsquigarrow x$. To see why this holds, assume that the inner replace returns a string other than x (otherwise it holds trivially). In that case, the returned string must be longer than x, so the outer replacement does nothing. We implemented roughly 30 classes of bit-vector rewrites, including $x + 1 \rightsquigarrow -\sim x$, $x - (x \mathbin{\&} y) \rightsquigarrow x \mathbin{\&} \sim y$, and $x \mathbin{\&} \sim x \rightsquigarrow 0$ where $\sim$ and $\&$ are respectively bitwise negation and conjunction. Note that our workflow suggests bit-vector rewrites for fixed bit-widths, so the developer has to establish which rewrites hold *for all* bit-widths. For Booleans, we implemented several classes of rewrite rules for negation normal form, commutative argument sorting, equality chain normalization, and constraint propagation.

By design, our framework does not generalize the candidate rewrite rules. We found instead that the framework naturally suggests candidates that can be seen as instances of the same generalized rule. This allows the developer to devise rules over conditions that are not easily expressible as logical formulas. For example, consider the candidates: $x \texttt{ ++ } \texttt{"A"} \approx \texttt{"AA"} \rightsquigarrow x \approx \texttt{"A"}$, $x \texttt{ ++ } \texttt{"A"} \approx \texttt{"A"} \rightsquigarrow x \approx \texttt{""}$, and $\texttt{"BBB"} \approx x \texttt{ ++ } \texttt{"B"} \texttt{ ++ } y \rightsquigarrow \texttt{"BB"} \approx x \texttt{ ++ } y$. These suggest that if one side of an equality is just a repetition of a single character, one can drop any number of occurrences of it from both sides. The condition and conclusion of such rules are more easily understood operationally than as logical formulas.

5.2 Evaluating Internal Metrics of our Workflow

We now address the following questions about the effectiveness of our workflow:

- How does the number of unique terms scale with the number of grammar terms?
- How do rewriters affect term redundancy and enumeration performance?
- What is the accuracy and performance of different equivalence checks?
- How many candidate rewrites do our filtering techniques eliminate?

Grammar	Size	Terms	T-Unique	none Red. %	none Time [s]	std Red. %	std Time [s]	ext Red. %	ext Time [s]	Z3 Red. %
strterm	1	218	86	60.6%	0.18	17.3%	0.09	0.0%	0.09	21.1%*
	2	24587	4204	82.9%	22.64	49.4%	8.78	20.0%	3.57	52.5%*
strpred	1	104	31	70.2%	0.13	34.0%	0.13	0.0%	0.07	32.6%*
	2	8726	1057	87.9%	9.32	66.5%	7.66	26.2%	2.16	68.1%*
	3	1100144	≥53671	—	t/o	≤82.5%	1154.02	≤57.0%	376.61	—
bvterm$_4$	1	63	22	65.1%	0.16	8.3%	0.12	0.0%	0.13	29.0%
	2	2343	288	87.7%	1.03	22.0%	0.24	0.7%	0.14	58.3%
	3	110583	4744	95.7%	89.84	39.3%	11.03	9.9%	3.55	76.9%
	4	5865303	84048	—	t/o	—	t/o	23.9%	242.15	—
bvterm$_{32}$	1	63	22	65.1%	0.09	8.3%	0.05	0.0%	0.05	29.0%
	2	2343	290	87.6%	4.53	21.4%	1.45	0.0%	0.85	58.0%
	3	110583	4925	95.5%	462.47	37.0%	85.62	6.5%	51.79	—
	4	5865303	≥84229†	—	t/o	—	t/o	≤23.8%	1955.97	—
crci	1	4	3	25.0%	0.11	25.0%	0.13	0.0%	0.11	0.0%
	2	32	12	62.5%	0.11	52.0%	0.12	0.0%	0.12	33.3%
	3	276	44	84.1%	0.15	74.4%	0.13	0.0%	0.12	62.1%
	4	2656	176	93.4%	0.38	87.5%	0.28	0.0%	0.13	81.1%
	5	17920	228	98.7%	2.11	96.9%	1.05	0.0%	0.15	93.1%
	6	107632	348	99.7%	15.97	99.0%	6.33	36.7%	0.24	97.8%
	7	596128	396	99.9%	112.71	99.8%	31.62	68.3%	0.43	99.3%
	8	2902432	396	—	t/o	99.9%	124.28	71.4%	0.45	—

Table 1: Impact of different rewriters on term redundancy using grammar equivalence checking. * may be inaccurate because Z3's rewriter is incorrect for strings (see Section 5.3 for details). † estimate is based on the unique terms for bvterm$_4$.

We consider four grammars: strterm and strpred for the theory of strings, bvterm for the theory of bit-vectors, and crci for Booleans. To show the impact of bit-width, we consider 4-bit (bvterm$_4$) and a 32-bit (bvterm$_{32}$) variant of bvterm. Figure 2 shows strterm and bvterm$_4$ in SyGuS syntax. A SyGuS problem specifies a function `f` with a set of parameters (e.g. two strings `x` and `y` and an integer `z` for strterm) and a return type (e.g. a string for strterm) to synthesize. The initial symbol of the grammar is `Start` and each rule of the grammar is given by a symbol (e.g. `ie` in strterm) and a list of terms (e.g. the constants `0`, `1` and the functions `len`, `str.to.int`, `indexof` for `ie`). We omit semantic constraints because we want to enumerate all terms for a given grammar. The number of terms is infinite, so our evaluation restricts the search for candidate rewrites by limiting the *size* of the enumerated terms, using a 24h timeout and a 32 GB RAM limit. The size of a term refers to the number of non-nullary symbols in it. Our implementation provides three rewriter settings: none disables rewriting; and std and ext are as defined in Section 5.1. For equivalence checking, we implemented the three methods from Section 3: random sampling (random); grammar-based sampling (grammar); and exact equivalence checking (exact).

T-Unique Solutions In Table 1, we show the number of terms (Terms) and the number of unique terms modulo T-equivalence (T-Unique) at different sizes for each grammar. We established the number of unique terms for the Boolean and bvterm$_4$ grammars using the exact equivalence checking technique. For bvterm$_{32}$ and the string grammars, some equivalence checks are challenging despite their small size, as we discuss in Section 5.3.

For terms of size 2 from the string grammars and terms of size 3 from the bvterm_{32} grammar, we resolved the challenging equivalence checks in a semi-automated fashion in some cases and manually in others. For larger term sizes, we approximated the number of unique terms using grammar with $10{,}000$ samples.[4] Our equivalence checks do not produce false negatives, i.e. they will not declare two terms to be different when they are actually equivalent. Thus, grammar can be used to compute a lower bound on the actual number of unique terms. For all grammars, the number of terms grows rapidly with increasing size while the number of unique terms grows much slower. Thus, enumerating terms without rewriting is increasingly inefficient as terms grow in size, indicating the utility of aggressive rewriting in these domains. The number of unique terms differs between the 4-bit and the 32-bit versions of bvterm at size 2, showing that some rewrite rules are valid for smaller bit-widths only.

Rewriter Comparison To measure the impact of different rewriters, we measured the redundancy of our rewriter configurations at different sizes for each grammar, and the wall-clock time to enumerate and check all the solutions. We define the *redundancy* of a rewriter for a set of terms S to be $(n-u)/n$, where n is the cardinality of $\{t\!\downarrow \mid t \in S\}$ and u is the number of T-unique terms in S. We used grammar with $1{,}000$ samples for the equivalence checks. As a point of reference, we include the redundancy of Z3's `simplify` command by counting the number of unique terms after applying the command to terms generated by none. Table 1 summarizes the results. With none, the redundancy is very high at larger sizes, whereas std keeps the redundancy much lower, except for crci. This is because std only performs basic rewriting for Boolean terms as CVC4 relies on a SAT solver for Boolean reasoning. Overall, std is competitive with Z3's rewriter, indicating that it is a decent representative of a state-of-the-art rewriter. As expected, ext fares much better in all cases, lowering the percentage of redundant terms by over 95% in the case of crci at size 5. This has a significant effect on the time it takes to enumerate all solutions: ext consistently and significantly outperforms std, in some cases by almost two orders of magnitude (crci at size 7), especially at larger sizes. Compared to none, both std and ext perform much better.

Equivalence Check Comparison To compare the different equivalence checks, we measured their error with respect to the set of terms enumerated by the ext rewriter, and the wall-clock time to enumerate all the solutions. For a set of terms S, we define the error of an equivalence check as $(u-n)/u$, where n is number of equivalence classes of S induced by the check and u is the number of T-unique terms in S, where $u \geq n$. For both random and grammar, we used $1{,}000$ samples. We summarize the results in Table 2. For crci and bvterm_4, $1{,}000$ samples are enough to cover all possible inputs, so there is no error in those cases and we do not report grammar. While sampling performs similarly to exact for crci and bvterm_4, for bvterm_{32}, exact is more precise and slightly faster for size 2. At sizes 3 and 4, exact ran out of memory. For strterm, we found that grammar was much more effective than random, having an error that was around 2.7 times smaller for term size 2. Similarly, grammar-based sampling was more effective on strpred with an error that was 1.5 times smaller for term size 2. Recall that

[4] For a better estimate for bvterm_{32}, we approximate the number as $u_{4,n} + u_{32,n-1} - u_{4,n-1}$ where $u_{m,n}$ is the number of unique terms for bit-width m and term size n.

		no − eqc	random		grammar		exact		Rewrites	Confidence
Grammar	Size	Time	Error	Time	Error	Time	Error	Time	Filtered	Overhead
strterm	1	0.04	0.0%	0.03	0.0%	0.09	0.0%	0.13	0.0%	111.1%
	2	0.60	2.5%	3.75	0.9%	3.57	—	t/o	63.8%	2.0%
strpred	1	0.03	0.0%	0.03	0.0%	0.07	0.0%	0.12	0.0%	85.7%
	2	0.49	6.8%	2.09	4.4%	2.16	—	t/o	59.8%	6.9%
	3	59.54	≤16.1%	380.23	≤13.6%	376.61	—	t/o	66.5%	0.6%
bvterm_4	1	0.02	0.0%	0.04			0.0%	0.02	0.0%	92.3%
	2	0.03	0.0%	0.06			0.0%	0.12	50.0%	85.7%
	3	0.35	0.0%	3.56			0.0%	2.10	45.0%	3.1%
	4	9.71	0.0%	266.09			0.0%	215.93	60.8%	2.3%
bvterm_{32}	1	0.01	27.3%	0.04	0.0%	0.05	0.0%	0.12	0.0%	80.0%
	2	0.03	62.8%	1.54	15.9%	0.85	0.0%	0.47	0.0%	16.5%
	3	0.35	79.9%	69.40	40.9%	51.79	—	t/o	57.8%	7.5%
	4	9.06	≤87.3%	2502.11	≤57.3%	1955.97	—	t/o	69.7%	2.9%
crci	1	0.02	0.0%	0.03			0.0%	0.02	0.0%	163.6%
	2	0.02	0.0%	0.05			0.0%	0.02	0.0%	150.0%
	3	0.04	0.0%	0.03			0.0%	0.03	0.0%	191.7%
	4	0.05	0.0%	0.06			0.0%	0.05	0.0%	146.2%
	5	0.08	0.0%	0.08			0.0%	0.12	0.0%	113.3%
	6	0.10	0.0%	0.17			0.0%	0.43	0.0%	87.5%
	7	0.22	0.0%	0.37			0.0%	1.58	0.0%	48.8%
	8	0.21	0.0%	0.38			0.0%	1.70	0.0%	44.4%

Table 2: Comparison of different equivalence checks, the number of candidates filtered and the overhead of checking rewrites for soundness (Section 5.4), using the ext rewriter. For bvterm_4 and crci random and grammar, are the same.

we determined the numbers of unique terms at this size manually and grammar is good enough to discard a majority of spurious rewrites at those small sizes. As expected, exact gets stuck and times out for strterm and strpred at sizes 2 and 3.

Impact of Filtering Table 2 also lists how many candidate rewrites the filtering techniques in Section 4 eliminate. We used ext and exact if available and grammar otherwise. Filtering eliminates up to 69.7% of the rules, which significantly lowers the burden on the developer when analyzing the proposed rewrite rules.

5.3 Evaluating SMT Solvers for Equivalence Checking

In this section, we demonstrate the use of our workflow to generate small queries that correspond to checking the equivalence of two enumerated terms to evaluate other SMT solvers. The motivation for this is twofold. First, we are interested in how other SMT solvers perform as equivalence checkers in our workflow. Second, we are interested in finding queries that uncover issues in SMT solvers or serve as small but challenging benchmarks. In contrast to random testing for SMT solvers [14, 23, 13], our approach can be seen as a form of *exhaustive* testing, where all relevant queries up to a given term size are considered.

First, we logged the candidate rewrites generated by our workflow up to a fixed size using a basic rewriter that only evaluates operators with constant arguments. We

Grammar	Result	CVC4ext	CVC4std	Z3		Z3STR3		BOOLECTOR
				Solved	#w	Solved	#w	
strterm (1045)	unsat	1030 (666)	991 (254)	888 (348)		953	3	
	sat	10	9	5	93	4	28	
	unsolved	5	45	59		57		
strpred (835)	unsat	807 (569)	775 (297)	716 (287)		779		
	sat	13	13	6	32	11	17	
	unsolved	15	47	81		28		
bvterm_{32} (1575)	unsat	1484 (1271)	1406 (641)	1426 (743)				1399 (766)
	sat	85	85	89				89
	unsolved	6	84	60				87

Table 3: Results for equivalence checking with a $300s$ timeout. The number of benchmarks for each grammar is below its name. The number of responses x (y) indicates that the solver solved x benchmarks, of which it solved y by simplification only. Incorrect responses from solvers are given in the columns #w.

considered the grammars strterm, strpred, and bvterm_{32}, for which equivalence checking is challenging. We considered a size of bound 2 for the string grammars and 3 for bvterm_{32}. Then, we used grammar-based sampling with $1{,}000$ samples to compute candidate rewrite rules, outputting for each candidate $t \approx s$ the (quantifier-free) satisfiability query $t \not\approx s$. Finally, we tested CVC4 with the ext and std rewriters as well as state-of-the-art SMT solvers for string and bit-vector domains. Specifically, we tested Z3 [20] 4.8.1 for all grammars, Z3STR3 [12] for the string grammars and BOOLECTOR [22] for the bit-vector grammar. This set of solvers includes the winners of the QF_BV (BOOLECTOR) and BV (CVC4) divisions of SMT-COMP 2018 [1].

Table 3 summarizes the results. For each grammar and solver, we give the number of unsatisfiable, satisfiable, and unsolved responses. We additionally provide the number of unsatisfiable responses for which a solver did not require any SAT decisions, i.e., it solved the benchmark with rewriting only.

For benchmarks from the string grammars, we found that Z3 and Z3STR3 generated 125 and 45 incorrect "sat" responses respectively.[5] For 122 and 44 of these cases respectively, CVC4ext answered "unsat". For the other 3 cases, CVC4ext produced a different model that was accepted by all solvers. Additionally, we found that Z3STR3 gave 3 confirmable incorrect "unsat" responses.[6] We filed these cases as bug reports and the developers confirmed that reasons for the incorrect responses include the existence of unsound rewrites. As expected, CVC4ext significantly outperforms the other solvers because the ext rewriter is highly specialized for this domain. CVC4ext solves a total of $1{,}235$ instances using rewriting alone, which is 684 instances more than CVC4std. Even on instances that CVC4ext's rewriter does not solve directly, it aids solving. This is illustrated by CVC4ext solving 96.8% whereas CVC4std solving only 92.2% of the string benchmarks that neither of them solved by rewriting alone.

[5] The solver answered "sat", but produced a model that did not satisfy the constraints.
[6] The solver answered "unsat", but accepted a model generated by CVC4ext.

For bvterm_{32}, we found no incorrect answers. Again, CVC4ext outperforms the other solvers due to the fact that its rewriter was trained for this grammar. Among the other solvers, BOOLECTOR solved the most benchmarks using simplification alone. Surprisingly, it solved 27 fewer unsatisfiable benchmarks than Z3, the second best performer. This can be primarily attributed to the fact that BOOLECTOR currently does not rewrite $x \cdot -y$ and $-(x \cdot y)$ to the same term. As a result, it could not prove the disequality $x \cdot -y \not\approx -(x \cdot y)$ within $300s$. Two variants of $(s \cdot s) \gg (s \ll s) \not\approx s \cdot (s \gg (s \ll s))$ were challenging for all solvers, which confirms that our workflow can be used to find small, challenging queries.

5.4 Improving Confidence in the Rewriter

The soundness of rewriters is of utmost importance because an unsound rewriter often implies that the overall SMT solver is unsound. A rewriter is unsound if there exists a pair of T-disequivalent terms t and s such that $t\downarrow = s\downarrow$. To accommodate the rapid development of rewrite rules in our workflow in Figure 1, CVC4 supports an optional mode that attempts to detect unsoundness in its rewriter. When this mode is enabled, for each term t enumerated in Step 2, we use grammar to test the equivalence of t and $t\downarrow$. In particular, this technique discovers points where t and $t\downarrow$ have different values. This functionality has been critical for discovering subtle bugs in the implementation of new rules. It even caught *previously existing* bugs in CVC4's rewriter. For instance, an older version of CVC4 implemented $\mathtt{replace}(x, x, y) \rightsquigarrow y$ which was incompatible with the (now outdated) semantics of `replace` (if the second argument was empty, the operation would leave the first argument unchanged). Table 2, shows the overhead of running these checks for each grammar and term size, where grammar-based sampling is used both for computing the rewrite rules and for checking the soundness of the ext rewriter. For example, adding checks that ensure that no unsound rewrites are produced for terms up to size 3 in the bvterm_{32} grammar has a 7.5% overhead. As term size increases and the enumeration rate decreases, the relative overhead tends to becomes smaller. Overall, this option leads to a noticeable, but not prohibitively large, slowdown in the workflow.

5.5 Impact of Rewrites on Solving

Finally, we evaluate the impact of the extended rewrites on CVC4's solving performance on SyGuS and SMT problems. Figure 3 summarizes the results.

Impact on SyGuS Problems The Boolean and the string problems in our evaluation are from SyGuS-COMP 2018 [2]. Their grammars are similar to the ones we used in our workflow. We distinguish between two types of bit-vector (BV) benchmarks: programming-by-examples (PBE) benchmarks where the constraints are input-output pairs and benchmarks with arbitrary non-PBE constraints. We make this distinction because for PBE problems, CVC4 can rely on input examples to decide term equivalence (the function to synthesize can return arbitrary values for the domain outside of the input examples). For the non-PBE benchmarks, we use the benchmarks from the general track of SyGuS-COMP, from work on synthesizing invertibility conditions (IC) for bit-vector

Benchmark Set	#	std	ext	S%
Strings	108	73	**88**	1.81×
BV (non-PBE)	361	236	**238**	1.02×
▷ IC	160	**131**	130	0.76×
▷ CegisT	79	**41**	**41**	1.09×
▷ General	122	64	**67**	1.73×
BV (PBE)	803	**774**	773	1.11×
Boolean	214	153	**159**	1.83×

Logic	Result	std	ext	S%
QF_SLIA (25421)	unsat	3803	**3823**	0.46×
	sat	20887	**20950**	1.87×
BV (5751)	unsat	4969	**4974**	1.12×
	sat	536	**542**	1.41×

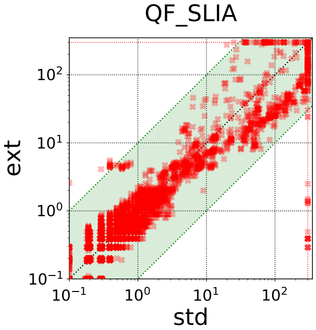

Fig. 3: Impact of ext on SyGuS (top left) and SMT (bottom left and right) solving. They ran with a $3600s$ and $300s$ timeouts respectively. Best results are in **bold**. The scatter plot is logarithmic. "S%" is the speedup of ext over std on commonly solved problems.

operators [24], and invariant synthesis [4]. The PBE bit-vector benchmarks [19, 5] are from the 2013 ICFP programming competition. Their grammars vary significantly.

The top table in Figure 3 lists the number of solved instances within a $3600s$ timeout for ext and std and the speedup of ext over std on commonly solved benchmarks. With ext, CVC4 solves more benchmarks on all benchmark sets except IC and the PBE bit-vector problems. We experienced a significant speedup on the commonly solved instances from the Boolean benchmarks, the SyGuS-COMP general track bit-vector benchmarks, and the string benchmarks. The IC grammars are focused on bit-vector comparisons, which our grammar bvterm lacks. Thus, the new rewrites do not significantly reduce the redundancy for that set. On the PBE bit-vector problems, ext is slightly faster overall and solves a problem that std does not solve but times out on two hard problems that std solves, which is likely due to the rewrites affecting CVC4's search heuristic.

Impact on SMT Problems We evaluated the impact of ext on quantifier-free string benchmarks (QF_SLIA) from the symbolic execution of Python programs [28] and found that ext has a significant positive impact. While ext is faster than std on commonly solved satisfiable benchmarks, it is slower on unsatisfiable ones due to three outliers. The scatter plot in Figure 3 shows that for benchmarks that take more than one second, there is a trend towards shorter solving times with ext.

For benchmarks in the quantified bit-vector logic (BV) of SMT-LIB [10], the extended rewriter improves the overall performance by solving 11 additional instances as shown in the bottom table in Figure 3.

For quantifier-free bit-vector (QF_BV) problems, naively using all new bit-vector rules from ext resulted in fewer solved instances. This is due to the fact that ext performs aggressive rewriting to eliminate as much redundancy as possible, which is helpful in enumerative SyGuS. For QF_BV solving, however, it is important to consider the effects of a rewrite at the word-level *and* the bit-level. Rewrites at the word-level can destroy common structures between different terms at the bit-level, which may increase the size of the formula at the bit-level while decreasing it at the word level. Rewrites that eliminate

redundancy at the cost of introducing expensive operators (e.g., multipliers) may also harm performance. We evaluated each bit-vector rewrite in ext w.r.t. solving QF_BV problems. We generalized one family involving bitwise operators over concatenations that e.g. rewrites $x \mathbin{\&} (0 \circ y)$ to $0 \circ (x[m-1:0] \mathbin{\&} y)$, where x has bit-width n and y has bit-width m for some $m < n$. These resulted in a net gain of 41 solved instances over $40{,}102$ QF_BV benchmarks with $1{,}089$ timeouts, a 3.8% improvement in the success rate of CVC4 on this set. This suggests that developing a library of rewrite rules and selectively enabling some can be beneficial in this domain.

6 Related Work

A number of techniques have been proposed for automatically generating rewrite rules for bit-vectors. SWAPPER [31] is an automatic formula simplifier generator based on machine learning and program synthesis techniques. In the context of symbolic execution, Romanoe et al. [29] propose an approach that learns rewrite rules to simplify expressions before sending them to an SMT solver. In contrast, our approach targets the SMT solver developer and is not limited to rewrites expressible in a restricted language. A related approach was explored by Hansen [18], which generates all the terms that fit a grammar and finds equivalent pairs that can the be used by the developer to implement new rules. In contrast to our work, the candidate rules are not filtered, and the grammar is hard-coded and only considers bit-vector operations. Nadel [21] proposed generating bit-vector rewrite rules in SMT solvers *at runtime* for a given problem. Syntax-guided synthesis was used by Niemetz et al. [24] to synthesize conditions that characterize when bit-vector constraints have solutions. Rewrite rules in SMT solvers—especially the ones for the theories of bit-vectors and Booleans—bear similarities with local optimizations in compilers [8, 15]. Finally, caching counterexamples as we do in our exact equivalence check is similar to techniques used in symbolic execution engines, e.g. KLEE [16], and superoptimizers, e.g. STOKE [30].

7 Conclusion

We have presented a syntax-guided paradigm for developing rewriters for SMT solvers. In ongoing work, we are exploring an automated analysis of how likely particular rewrites are to help solving constraints, which we found was the most tedious aspect of our current workflow. Furthermore, we plan to adapt existing techniques for automatically constructing grammars from a set of problems to use them as inputs to our approach. In the shorter term, we plan to use the existing framework to identify useful rewrite rules for emerging SMT domains, notably the theory of floating-point arithmetic, for which developing a rewriter is notoriously difficult.

Acknowledgements This material is based upon work partially supported by the National Science Foundation (Award No. 1656926), the Office of Naval Research (Contract No. 68335-17-C-0558), and DARPA (N66001-18-C-4012, FA8650-18-2-7854 and FA8650-18-2-7861).

References

1. SMT-COMP 2018. http://smtcomp.sourceforge.net/2018/ (2018)
2. SyGuS-COMP 2018. http://sygus.seas.upenn.edu/SyGuS-COMP2018.html (2018)
3. CVC4 sat2019 branch. https://github.com/4tXJ7f/CVC4/tree/sat2019 (2019)
4. Abate, A., David, C., Kesseli, P., Kroening, D., Polgreen, E.: Counterexample guided inductive synthesis modulo theories. In: Chockler, H., Weissenbacher, G. (eds.) 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. Lecture Notes in Computer Science, vol. 10981, pp. 270–288. Springer (2018), https://doi.org/10.1007/978-3-319-96145-3_15
5. Akiba, T., Imajo, K., Iwami, H., Iwata, Y., Kataoka, T., Takahashi, N., Moskal, M., Swamy, N.: Calibrating research in program synthesis using 72,000 hours of programmer time. MSR, Redmond, WA, USA, Tech. Rep (2013)
6. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8. IEEE (2013), http://ieeexplore.ieee.org/document/6679385/
7. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 319–336 (2017), https://doi.org/10.1007/978-3-662-54577-5_18
8. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. In: Shen, J.P., Martonosi, M. (eds.) Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006. pp. 394–403. ACM (2006), http://doi.acm.org/10.1145/1168857.1168906
9. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011), https://doi.org/10.1007/978-3-642-22110-1_14
10. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
11. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
12. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 55–59 (2017), https://doi.org/10.23919/FMCAD.2017.8102241
13. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V.: Stringfuzz: A fuzzer for string solvers. In: 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 II. pp. 45–51 (2018), https://doi.org/10.1007/978-3-319-96142-2_6

14. Brummayer, R., Biere, A.: Fuzzing and Delta-Debugging SMT Solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT'09). p. 5. ACM (2009)
15. Buchwald, S.: Optgen: A generator for local optimizations. In: Franke, B. (ed.) Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9031, pp. 171–189. Springer (2015), https://doi.org/10.1007/978-3-662-46663-6_9
16. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
17. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.C.: Word equations with length constraints: What's decidable? In: Biere, A., Nahir, A., Vos, T.E.J. (eds.) Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers. Lecture Notes in Computer Science, vol. 7857, pp. 209–226. Springer (2012), https://doi.org/10.1007/978-3-642-39611-3_21
18. Hansen, T.: A constraint solver and its application to machine code test generation. Ph.D. thesis, University of Melbourne, Australia (2012), http://hdl.handle.net/11343/37952
19. Jr., H.S.W.: Hacker's Delight, Second Edition. Pearson Education (2013), http://www.hackersdelight.org/
20. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. pp. 337–340 (2008), https://doi.org/10.1007/978-3-540-78800-3_24
21. Nadel, A.: Bit-vector rewriting with automatic rule generation. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. pp. 663–679 (2014), https://doi.org/10.1007/978-3-319-08867-9_44
22. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. Journal on Satisfiability, Boolean Modeling and Computation **9**, 53–58 (2014 (published 2015))
23. Niemetz, A., Preiner, M., Biere, A.: Model-Based API Testing for SMT Solvers. In: Brain, M., Hadarean, L. (eds.) Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT 2017), affiliated with the 29th International Conference on Computer Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017. p. 10 pages (2017)
24. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) 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 II. Lecture Notes in Computer Science, vol. 10982, pp. 236–255. Springer (2018), https://doi.org/10.1007/978-3-319-96142-2_16
25. Reynolds, A., Barbosa, H., , Nötzil, A., Barrett, C., Tinelli, C.: CVC4Sy: Smart and fast term enumeration for syntax-guided synthesis. In: Dilig, I., Tasiran, S. (eds.) Computer Aided Verification (CAV) - 31st International Conference. (Accepted for publication). Lecture Notes in Computer Science, Springer (2019)

26. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.W.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9207, pp. 198–216. Springer (2015), https://doi.org/10.1007/978-3-319-21668-3_12
27. Reynolds, A., Tinelli, C.: Sygus techniques in the core of an SMT solver. In: Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017. pp. 81–96 (2017), https://doi.org/10.4204/EPTCS.260.8
28. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 453–474. Springer (2017), https://doi.org/10.1007/978-3-319-63390-9_24
29. Romano, A., Engler, D.R.: Expression reduction from programs in a symbolic binary executor. In: Bartocci, E., Ramakrishnan, C.R. (eds.) Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7976, pp. 301–319. Springer (2013), https://doi.org/10.1007/978-3-642-39176-7_19
30. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Sarkar, V., Bodík, R. (eds.) Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013. pp. 305–316. ACM (2013), http://doi.acm.org/10.1145/2451116.2451150
31. Singh, R., Solar-Lezama, A.: SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016. pp. 185–192. IEEE (2016), https://doi.org/10.1109/FMCAD.2016.7886678
32. Tinelli, C., Barrett, C., Fontaine, P.: Unicode Strings (Draft 1.0). http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml (2018)
33. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: specifying protocols with concolic snippets. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 287–296 (2013), http://doi.acm.org/10.1145/2462156.2462174

A Appendix

A.1 Grammars

In the following, we show the crci and strpred (extension of strterm where `s` corresponds to `Start` in strterm) grammars.

Listing 1.1: crci

```
(synth-fun f ((x Bool) (y Bool)
  (z Bool) (w Bool)) Bool (
 (Start Bool ((and d1 d1) (not d1)
  (or d1 d1) (xor d1 d1)))
 (d1 Bool (x (and d2 d2) (not d2)
  (or d2 d2) (xor d2 d2)))
 (d2 Bool (w (and d3 d3) (not d3)
  (or d3 d3) (xor d3 d3)))
 (d3 Bool (y (and d4 d4) (not d4)
  (or d4 d4) (xor d4 d4)))
 (d4 Bool (z))))
```

Listing 1.2: strpred

```
(synth-fun f ((x String)
  (y String) (z Int))
  Bool (
 (Start Bool (
  true false
  (not Start)
  (= s s)
  (str.prefixof s s)
  (str.suffixof s s)
  (str.contains s s)))
 ...))
```

A.2 Example Output

We provide sample output of CVC4 in its rewrite rule enumeration mode. For each grammar, we give the results of CVC4 with no rewriter (none), its standard rewriter (std), and the rewriter we generated as a result of this work so far (ext). In each case, the output is a list of unoriented pairs indicating possible rewrites. All variables $x, y, z, w, s, t, \ldots$ should be interpreted as universal, in that the rewrite holds for all values of $x, y, z, w, s, t, \ldots$. Intuitively, the output when using none corresponds to a list of rewrites over terms in the grammar if the developer was writing a rewriter from scratch; the output when using std (resp. ext) corresponds to the simplest rewrites, measured in term size, over the terms from the grammar that the given rewriter currently can not infer.

A.3 strterm

Listing 1.3: none

```
(= (str.at "A" 1) "")
(= (str.at "B" 0) "B")
(= (str.at "B" 1) "")
(= (str.at "" 0) "")
(= (str.at "" 1) "")
(= (str.at "" z) "")
(= (str.substr x 0 0) "")
(= (str.substr x 0 1) (str.at x 0))
(= (str.substr x 1 0) "")
(= (str.substr x 1 1) (str.at x 1))
```

Listing 1.4: std

```
(= (str.at "" z) "")
(= (str.substr "A" 1 z) "")
(= (str.substr "A" z z) "")
(= (str.substr "B" 1 z) "")
(= (str.substr "B" z z) "")
(= (str.substr "" 0 z) "")
(= (str.substr "" 1 z) "")
(= (str.substr "" z z) "")
(= (str.replace x x y) y)
(= (str.replace y y "A") (str.replace x x "A"))
```

Listing 1.5: ext

```
(= (int.to.str (str.indexof "B" "" z))
   (int.to.str (str.indexof "A" "" z)))
(= (str.at x (str.indexof x "" 1)) (str.at x 1))
(= (str.at x (str.indexof x "" z)) (str.at x z))
(= (str.at x (str.indexof "A" x 1)) "")
(= (str.at x (str.indexof "B" "" z))
   (str.at x (str.indexof "A" "" z)))
(= (str.at x (str.indexof "" x 0)) "")
(= (str.at x (str.indexof "" x z)) "")
(= (str.at "A" (- 0 z)) (str.at "A" z))
(= (str.at "A" (- z 1)) (str.at "A" (- 1 z)))
(= (str.at "A" (+ z z)) (str.at "A" z))
```

A.4 strpred

Listing 1.6: none

```
(= (= x x) true)
(= (str.prefixof x x) true)
(= (str.suffixof x x) true)
(= (str.contains x x) true)
(= (str.suffixof x "A") (str.prefixof x "A"))
(= (str.suffixof x "B") (str.prefixof x "B"))
(= (str.prefixof x "") (= x ""))
(= (str.suffixof x "") (= x ""))
(= (str.contains x "") true)
(= (= y x) (= x y))
```

Listing 1.7: std

```
(= (str.suffixof x "A") (str.prefixof x "A"))
(= (str.suffixof x "B") (str.prefixof x "B"))
(= (str.prefixof x "") (= x ""))
(= (str.suffixof x "") (= x ""))
(= (str.contains x "") true)
(= (str.contains "A" x) (str.prefixof x "A"))
(= (str.contains "B" x) (str.prefixof x "B"))
(= (str.prefixof "" x) true)
(= (str.suffixof x (int.to.str 0))
   (str.prefixof x (int.to.str 0)))
(= (str.suffixof x (int.to.str 1))
   (str.prefixof x (int.to.str 1)))
```

Listing 1.8: ext

```
(= (= x (str.at x 1)) (= x ""))
(= (str.suffixof x (str.at y 0))
   (str.prefixof x (str.at y 0)))
(= (str.suffixof x (str.at y 1))
   (str.prefixof x (str.at y 1)))
(= (str.suffixof x (str.at y z))
   (str.prefixof x (str.at y z)))
(= (= x (str.substr x 1 z)) (= x ""))
(= (= x (str.substr x z z)) (= x ""))
(= (str.suffixof x (str.substr "A" 0 z))
   (str.prefixof x (str.substr "A" 0 z)))
(= (str.suffixof x (str.substr "B" 0 z))
   (str.prefixof x (str.substr "B" 0 z)))
(= (str.prefixof x (str.++ "A" x))
   (str.suffixof x (str.++ x "A")))
(= (str.suffixof x (str.++ "A" "A"))
   (str.prefixof x (str.++ "A" "A")))
```

A.5 bvterm$_4$

Listing 1.9: none

```
(= (bvand s s) s)
(= (bvor s s) s)
(= (bvadd s #b0000) s)
(= (bvmul s #b0000) (bvlshr s s))
(= (bvand s #b0000) (bvlshr s s))
(= (bvlshr s #b0000) s)
(= (bvor s #b0000) s)
(= (bvshl s #b0000) s)
(= (bvadd t s) (bvadd s t))
(= (bvmul t s) (bvmul s t))
```

Listing 1.10: std

```
(= (bvlshr t t) (bvlshr s s))
(= (bvneg (bvlshr s s)) (bvlshr s s))
(= (bvadd s (bvnot s)) (bvnot (bvlshr s s)))
(= (bvadd s (bvnot #b0000)) (bvnot (bvneg s)))
(= (bvadd s (bvlshr s s)) s)
(= (bvmul s (bvlshr s s)) (bvlshr s s))
(= (bvand s (bvlshr s s)) (bvlshr s s))
(= (bvlshr s (bvlshr s s)) s)
(= (bvor s (bvlshr s s)) s)
(= (bvshl s (bvlshr s s)) s)
```

Listing 1.11: ext

```
(= (bvshl (bvshl s s) s) (bvshl (bvadd s s) s))
(= (bvlshr s (bvnot (bvneg s))) (bvlshr (bvadd s s) s))
(= (bvadd s (bvnot (bvadd s s))) (bvnot s))
(= (bvshl s (bvneg (bvmul s s)))
   (bvlshr s (bvneg (bvmul s s))))
(= (bvshl s (bvnot (bvmul s s)))
   (bvlshr s (bvnot (bvmul s s))))
(= (bvlshr s (bvneg (bvshl s s))) (bvlshr s (bvshl s s)))
(= (bvshl s (bvneg (bvshl s s))) (bvlshr s (bvshl s s)))
(= (bvshl s (bvnot (bvshl s s)))
   (bvlshr s (bvnot (bvmul s s))))
(= (bvshl s (bvneg (bvlshr t s)))
   (bvlshr s (bvneg (bvlshr t s))))
(= (bvlshr s (bvnot (bvlshr t s)))
   (bvlshr s (bvnot (bvmul s s))))
```

A.6 crci

Listing 1.12: none

```
(= (or x x) (and x x))
(= (not (or w w)) (not (and w w)))
(= (and x (or w w)) (and x (and w w)))
(= (or x (or w w)) (or x (and w w)))
(= (xor x (or w w)) (xor x (and w w)))
(= (and x (xor w w)) (xor x x))
(= (or x (xor w w)) (and x x))
(= (xor x (xor w w)) (and x x))
(= (and (not w) x) (and x (not w)))
(= (or (not w) x) (or x (not w)))
```

Listing 1.13: std

```
(= (or x x) (and x x))
(= (not (or w w)) (not (and w w)))
(= (and x (or w w)) (and x (and w w)))
(= (or x (or w w)) (or x (and w w)))
(= (xor x (or w w)) (xor x (and w w)))
(= (or x (xor w w)) (and x x))
(= (and (not w) x) (and x (not w)))
(= (or (not w) x) (or x (not w)))
(= (xor (not w) x) (xor x (not w)))
(= (or (and w w) x) (or x (and w w)))
```

Listing 1.14: ext

```
(= (xor (and w (not y)) (not (or y (not z))))
   (and (not (and y y)) (xor w (not (not z)))))
(= (xor (and w (not y)) (not (or y (and z z))))
   (and (not (and y y)) (xor w (not (and z z)))))
(= (xor (and w (not y)) (and w (not (not z))))
   (and (and w w) (not (xor y (and z z)))))
(= (xor (and w (not y)) (or w (not (not z))))
   (xor (not (and y y)) (or w (xor y (not z)))))
(= (xor (and w (not y)) (and w (not (and z z))))
   (and (and w w) (not (xor y (not z)))))
(= (xor (and w (not y)) (or w (not (and z z))))
   (xor (not (and y y)) (or w (xor y (and z z)))))
(= (xor (and w (not y)) (and w (and y (not z))))
   (and (and w w) (not (and y (and z z)))))
```

```
(= (xor (and w (not y)) (or w (and y (not z))))
   (and (not (not y)) (or w (not (and z z)))))
(= (or (and w (not y)) (xor w (and y (not z))))
   (xor (not w) (not (and y (not z)))))
(= (xor (and w (not y)) (xor w (and y (not z))))
   (and (not (not y)) (xor w (not (and z z)))))
```

A.7 Challenging Equivalence Checks

In this section, we list the rewrites from the section on "Evaluating SMT Solvers for Verifying Rewrites" that we were not able to prove automatically and that we either verified manually or in a semi-automated fashion. We list them as interesting challenges for tool developers.

Listing 1.15: strterm

```
(= (str.at (int.to.str z) z)
                    (int.to.str (str.indexof x x z)))
(= (str.replace (str.replace x y x) x y)
                    (str.replace x (str.replace x y x) y))
```

Listing 1.16: strpred

```
(= (str.prefixof x (str.replace y x y))
                    (str.prefixof x y))
(= (str.suffixof x (str.replace y x y))
                    (str.suffixof x y))
(= (str.suffixof "A" (str.replace x "A" ""))
   (str.suffixof "A" (str.replace x "A" "B")))
(= (str.contains "A" (str.replace x "A" ""))
   (str.prefixof x (str.++ "A" "A")))
(= (str.contains "B" (str.replace x "A" ""))
   (str.contains "A" (str.replace x "B" "")))
(= (str.suffixof "B" (str.replace x "B" ""))
   (str.suffixof "B" (str.replace x "B" "A")))
(= (str.contains "B" (str.replace x "B" ""))
   (str.prefixof x (str.++ "B" "B")))
(= (str.prefixof (str.++ "A" "A") x)
   (str.prefixof "A" (str.replace x "A" "")))
(= (str.prefixof (str.++ "B" "B") x)
   (str.prefixof "B" (str.replace x "B" "")))
(= (str.suffixof (str.replace x "A" "") x)
   (str.prefixof x (str.replace x "A" x)))
(= (str.suffixof (str.replace x "B" "") x)
   (str.prefixof x (str.replace x "B" x)))
```

Listing 1.17: bvterm

```
(= (bvlshr (bvmul s s) (bvshl s s))
                    (bvmul s (bvlshr s (bvshl s s))))
(= (bvlshr (bvmul s s) (bvshl t s))
                    (bvmul s (bvlshr s (bvshl t s))))
```

We solved all but the first two equivalences in strpred using a different configuration of CVC4. We verified the bvterm rewrites semi-automatically by breaking the problem down into smaller chunks and solving those chunks with an SMT solver. We checked the remaining four equivalences in strterm and strpred using pencil-and-paper proofs.

A.8 Scatter Plots Showing Impact of Rewrites on SyGuS Solving

The following scatter plots show the difference between ext and std on SyGuS solving in more detail. They were generated from the same data as the table in Figure 3.