High-Level Abstractions for Simplifying Extended String Constraints in SMT

Andrew Reynolds[1], Andres Nötzli[2(✉)], Clark Barrett[2], and Cesare Tinelli[1]

[1] Department of Computer Science, The University of Iowa, Iowa City, USA
[2] Department of Computer Science, Stanford University, Stanford, USA
noetzli@cs.stanford.edu

Abstract. Satisfiability Modulo Theories (SMT) solvers with support for the theory of strings have recently emerged as powerful tools for reasoning about string-manipulating programs. However, due to the complex semantics of *extended string functions*, it is challenging to develop scalable solvers for the string constraints produced by program analysis tools. We identify several classes of simplification techniques that are critical for the efficient processing of string constraints in SMT solvers. These techniques can reduce the size and complexity of input constraints by reasoning about arithmetic entailment, multisets, and string containment relationships over input terms. We provide experimental evidence that implementing them results in significant improvements over the performance of state-of-the-art SMT solvers for extended string constraints.

1 Introduction

Most programming languages support strings natively and a considerable number of programs perform some form of string manipulation. Automated reasoning about string-manipulating programs for verification and test case generation purposes is then highly relevant for these languages and programs. Applications to security, such as finding SQL injection and XSS vulnerabilities in web applications [16,18,23] or proving their absence, are of critical importance. String constraints have also been used to generate relational database tables from SQL queries for unit testing purposes [21]. These applications require modeling all of the string operations that appear in real programs. This is challenging since some of those operations are complex and often realized by iterative applications of simpler operations. Additionally, since strings in many programming languages have variable length, reasoning accurately about them cannot be done by a reduction to bounded types such as bit-vectors, and requires instead the development of solvers for *unbounded* strings. To make this type of reasoning more scalable, the use of dedicated theory solvers natively supporting common string operations has been proposed [5,9]. Some string solvers are fully integrated within

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11562, pp. 23–42, 2019.
https://doi.org/10.1007/978-3-030-25543-5_2

Satisfiability Modulo Theories (SMT) solvers [4,12]; some are built (externally) on top of such solvers [9,16,19]; and others are independent of SMT solvers [23].

A major challenge in developing solvers for unbounded string constraints is the complex semantics of *extended string functions* beyond the basic operations of string concatenation and equality. Extended functions include replace, which replaces a string in another string, and indexof, which returns the position of a string in another string. Another challenge is that constraints using extended functions are often combined with constraints over other theories, e.g. integer constraints over string lengths or applications of indexof, which requires the involvement of solvers for those theories. Current string solvers address these challenges by reducing constraints with extended string functions to typically more verbose constraints over basic functions. As with every reduction, some of the higher level structure of the problem may be lost, with negative repercussions on the performance and scalability.

To address this issue, we have developed new techniques that reason about constraints with extended string operators before they are reduced to simpler ones. This analysis of complex terms can often eliminate the need for expensive reductions. The techniques are based on reasoning about relationships over strings with high-level abstractions, such as their arithmetic relationships (e.g., reasoning about their length), their string containment relationships, and their relationships as multisets of characters. We have implemented these techniques in CVC4, an SMT solver with native support for string reasoning. An experimental evaluation with benchmarks from various applications shows that our new techniques allows CVC4 to significantly outperform other state-of-the-art solvers that target extended string constraints.

Our main contributions are:

- A novel procedure for proving entailments over arithmetic predicates built from the theory of strings and linear integer arithmetic.
- Extensions of this technique for showing containment relationships between strings.
- A novel simplification technique based on abstracting strings as multisets.
- Experimental evidence that the simplification techniques provide significant performance improvements over current state-of-the-art solvers.

In the remainder of this section, we discuss related work. In Sect. 2, we provide some background on the theory of strings and how solvers reduce extended functions. In Sects. 3, 4 and 5, we describe, respectively, our arithmetic-based, containment-based, and multiset-based simplification techniques. Section 6 describes our implementation of those techniques, and Sect. 7 presents our evaluation.

Related Work. Various approaches to solving constraints over extended string functions have been proposed. Saxena et al. [16] showed that constraints from the symbolic execution of JavaScript code contain a significant number of extended string functions, which underlines their importance. Their approach translates string constraints to bit-vector constraints, similar to other approaches based on

bounded strings such as HAMPI [9]. Bjørner et al. [5] proposed native support for extended string operators in string solvers for scaling symbolic execution of .NET code. They reduce extended string functions to basic ones after getting bounds for string lengths from an integer solver. They also showed that constraints involving unbounded strings and replace are undecidable. PASS [11] reduces string constraints over extended functions to arrays. Z3-str and its successors [4,24,25] reduce extended string functions to basic functions eagerly during preprocessing. S3 [18] reduces recursive functions such as replace incrementally by splitting and unfolding. Its successor S3P [19] refines this reduction by pruning the resulting subproblems for better performance. CVC4 [3] reduces constraints with extended functions lazily and leverages context-dependent simplifications to simplify the reductions [15]. TRAU [1] reduces certain extended functions, such as replace, to context-free membership constraints. OSTRICH [7] implements a decision procedure for a subset of constraints that include extended string functions. The simplification techniques presented in this paper are agnostic to the underlying solving procedure, so they can be combined with all of these approaches.

2 Preliminaries

We work in the context of many-sorted first-order logic with equality and assume the reader is familiar with the notions of signature, term, literal, formula, and formal interpretation of formulas. We review a few relevant definitions in the following. A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, the *models* of T. We assume Σ contains the equality predicate $\approx$, interpreted as the identity relation, and the predicates $\top$ (for true) and $\bot$ (for false). A Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. We write $\models_T \varphi$ to denote that the Σ-formula φ is T-*valid*, i.e., is satisfied in every model of T. Two Σ-terms t_1 and t_2 are *equivalent in* T if $\models_T t_1 \approx t_2$.

We consider an extended theory T_{S} of strings and length equations, whose signature Σ_{S} is given in Fig. 1 and whose models differ only on how they interpret variables.[1] We assume a fixed finite alphabet $\mathcal{A}$ of characters which includes the digits $\{0, \ldots, 9\}$. The signature includes the sorts Bool, Int, and Str denoting the Booleans, the integers ($\mathbb{Z}$), and Kleene closure of $\mathcal{A}$ ($\mathcal{A}^*$), respectively. The top half of Fig. 1 includes the usual symbols of *linear* integer arithmetic, interpreted as expected, a *string literal* l for each word/string of $\mathcal{A}^*$, a variadic function symbol con, interpreted as word concatenation, and a function symbol len, interpreted as the word length function. We write ϵ for the empty word and abbreviate $\mathsf{len}(s)$ as $|s|$. We use words over the characters a, b, and c, as in abca, as concrete examples of string literals.

We refer to the function symbols in the bottom half of the figure as *extended functions* and refer to terms containing them as *extended terms*. A *position* in

[1] Our implementation supports a larger set of symbols, but for brevity, we only show the subset of the symbols used throughout this paper.

$n : \mathsf{Int}$ for all $n \in \mathbb{N}$ $\quad + : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ $\quad - : \mathsf{Int} \to \mathsf{Int}$ $\quad \geqslant : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$
$l : \mathsf{Str}$ for all $l \in \mathcal{A}^*$ $\quad \mathsf{con} : \mathsf{Str} \times \cdots \times \mathsf{Str} \to \mathsf{Str}$ $\quad \mathsf{len} : \mathsf{Str} \to \mathsf{Int}$

$\mathsf{substr} : \mathsf{Str} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{Str}$ $\quad \mathsf{contains} : \mathsf{Str} \times \mathsf{Str} \to \mathsf{Bool}$
$\mathsf{indexof} : \mathsf{Str} \times \mathsf{Str} \times \mathsf{Int} \to \mathsf{Int}$ $\quad \mathsf{replace} : \mathsf{Str} \times \mathsf{Str} \times \mathsf{Str} \to \mathsf{Str}$
$\mathsf{str.to.int} : \mathsf{Str} \to \mathsf{Int}$ $\quad \mathsf{int.to.str} : \mathsf{Int} \to \mathsf{Str}$

Fig. 1. Functions in signature Σ_{S}. Str and Int denote strings and integers respectively.

a string $l \in \mathcal{A}^*$ is a non-negative integer n smaller than the length of l that identifies the $(n+1)^{th}$ character of l—with 0 identifying the first character, 1 the second, and so on. For all models $\mathcal{I}$ of T_{S}, all $l, l_1, l_2 \in \mathcal{A}^*$, and $n, m \in \mathbb{Z}$, $\mathsf{substr}^{\mathcal{I}}(l, n, m)$ (the interpretation of substr in $\mathcal{I}$ applied to l, n, m) is the longest substring of l starting at position n with length at most m, or ϵ if n is an invalid position or m is not positive; $\mathsf{contains}^{\mathcal{I}}(l_1, l_2)$ is true if and only if l_2 is a substring of l_1, with ϵ being a substring of every string; $\mathsf{indexof}^{\mathcal{I}}(l_1, l_2, n)$ is the position of the first occurrence of l_2 in l_1 at or after position n, n if l_2 is empty and $0 \leqslant n \leqslant |l_1|$, and -1 if n is an invalid position, or if no such occurrence exists; $\mathsf{replace}^{\mathcal{I}}(l, l_1, l_2)$ is the result of replacing the first occurrence of l_1 in l by l_2, l if l does not contain l_1, or the result of prepending l_2 to l if l_1 is empty; $\mathsf{str.to.int}^{\mathcal{I}}(l)$ is the non-negative integer represented by l in decimal notation or -1 if the string contains non-digit characters; $\mathsf{int.to.str}^{\mathcal{I}}(n)$ is the result of converting n to the corresponding string in decimal notation if n is non-negative, or ϵ otherwise. We write $\mathsf{substr}(t, u)$ as shorthand for the term $\mathsf{substr}(t, u, |t|)$, i.e. the suffix of t starting at position u.

Note that the semantics for replace and indexof correspond to the semantics in the current draft of the SMT-LIB standard for the theory of strings [17]; they are slightly different from the ones described in previous work [4,15,20].

2.1 Solving Extended String Constraints (with Simplification)

Various efficient solvers have been designed for the satisfiability problem for quantifier-free T_{S}-constraints, including CVC4 [3], S3# [20] and Z3STR3 [4]. In this section, we give an overview of how these solvers process extended functions in practice.

Generally speaking, constraints involving extended functions are converted to basic ones through a series of reductions performed in an incremental fashion by the solver. Operators whose reduction requires universal quantification are dealt with by guessing upper bounds on the lengths of input strings or by lazily adding constraints that block models that do not satisfy extended string constraints.

Example 1. To determine the satisfiability of $\neg\mathsf{contains}(t, s)$, the application of contains is reduced to constraints that ensure that s is not a substring of t at any position. Assuming we have a fixed upper bound n on the length of t, the above constraint is equivalent to the finite conjunction $\mathsf{substr}(t, 0, |s|) \not\approx s \wedge \cdots \wedge \mathsf{substr}(t, n, |s|) \not\approx s$. Each application of substr is then eliminated by

introducing an equality that constrains a fresh variable x_i to have the semantics of that substring. Thus, reducing the formula above results in

$$\bigwedge_{i=0}^{n} |t| \geqslant i + |s| \Rightarrow (x_i \not\approx s \wedge t \approx \mathsf{con}(x_i^{pre}, x_i, x_i^{post}) \wedge |x_i^{pre}| \approx i \wedge |x_i| \approx |s| \ )$$

where $x_i, x_i^{pre}, x_i^{post}$ are fresh string variables.[2] The above conjunction involves only string concatenation, string length, and equality, and thus can be handled by a string solver with support for word equations with length constraints.

The reduction in Example 1 introduces $5 \cdot n$ theory literals over basic string functions and $3 \cdot n$ string variables. A full reduction accounting for all corner cases of substr is even more complex and thus more expensive to process, even for small values of n. These performance challenges can be addressed by aggressive simplifications that *eliminate* extended functions using high-level reasoning, as shown in the next example.

Example 2. Consider an instance of the previous example where $s = \mathsf{con}(\mathsf{a}, x)$ and $t = \mathsf{con}(\mathsf{b}, \mathsf{substr}(x, 0, n))$. A full reduction of $\neg\mathsf{contains}(t, s)$ that eliminates all applications of substr, including those in t, introduces $5 \cdot n + 5$ new theory literals and $3 \cdot n + 3$ string variables. However, based on the semantics of contains it is easy to see that $\neg\mathsf{contains}(t, s)$ is T_{S}-valid: if t were to contain s, then s would have to occur in the portion of t after its first character b, since the first character of s is a. However, $\mathsf{con}(\mathsf{a}, x)$ cannot be contained in $\mathsf{substr}(x, 0, n)$, since the length of the former is at least $|x| + 1$, while the length of the latter is at most $|x|$. A solver which recognizes that $\neg\mathsf{contains}(t, s)$ can be simplified to $\top$ in this case can avoid the reduction altogether.

We advocate for aggressive simplification techniques to improve the performance of string solvers for extended functions. In the next sections, we describe several classes of such techniques that can be applied to inputs as a preprocessing step or during solving as part of a context-dependent solving strategy [15]. We present them as sets R of rewrite rules of the form $t \rightarrow_R s$, where s is a (simplified) term equivalent to t in T_{S}. We assume a deterministic application strategy for these rules, such that each term t rewrites to a unique *simplified form*, denoted by $t{\downarrow}$, which is irreducible by the rules. We split our simplifications into four categories, presented in Figs. 4, 6, 7 and 8.[3]

3 Arithmetic-Based String Simplification

To simplify string terms, it is useful to establish relationships between quantities such as the lengths of strings. For example, $\mathsf{contains}(t, s)$ can be simplified to $\bot$

[2] This formula is a simplified form of the general reduction. The general reduction also expresses that i is a valid position in t and that the third argument of substr is non-negative [15].

[3] Some specialized rules have been omitted for space reasons.

for a particular s and t if it can be inferred that $|s|$ is strictly greater than $|t|$. This section defines an inference system for such arithmetic relationships and the simplifications that it enables.

We are interested in proving the T_{S}-validity of formulas of the form $u \geqslant 0$, where u is a Σ_{S}-term of integer type. We describe an inference system as a set of rules for deriving judgments of the form $\vdash u \geqslant 0$ and a specific rule application strategy we have implemented. The inference system is *sound* in the sense that $\models_{T_{\mathsf{S}}} u \geqslant 0$ whenever $\vdash u \geqslant 0$ is derivable in it. It is, however, *incomplete* as it may fail to derive $\vdash u \geqslant 0$ in some cases when $\models_{T_{\mathsf{S}}} u \geqslant 0$. This incompleteness is by design, since proving the T_{S}-validity of inequalities is generally expensive due to the NP-hardness of linear integer arithmetic. Without loss of generality, we require that the term u be in a simplified form, where terms of the form $|l|$ with l a string literal of n characters are rewritten to n, terms of the form $|\mathsf{con}(t_1, \ldots, t_n)|$ are rewritten to $|t_1| + \cdots + |t_n|$, and like monomials in arithmetic terms are combined in the usual way (e.g., $2 \cdot |x| + |x|$ is rewritten to $3 \cdot |x|$).

Definition 1 (Polynomial Form). *An arithmetic term u is in polynomial form if $u = m_1 \cdot u_1 + \ldots m_n \cdot u_n + m$, where $m_1, \ldots, m_n$ are non-zero integer constants, m is an integer constant, and each $u_1, \ldots, u_n$ is a unique term and one of the following:*

1. *an integer variable,*
2. *an application of length to a string variable, e.g. $|x|$,*
3. *an application of length to an extended function, e.g. $|\mathsf{substr}(t, v, w)|$, or*
4. *an application of an extended function of integer type, e.g. $\mathsf{indexof}(t, s, v)$.*

Given u in polynomial form, our inference system uses a set of over- and under-approximations for showing that $u \geqslant 0$ holds in all models of T_{S}. We define two auxiliary rewrite systems, denoted $\rightarrow_O$ and $\rightarrow_U$. If u rewrites to v (in zero or more steps) in $\rightarrow_O$, written $u \rightarrow_O^* v$, we say that v is *an over-approximation* of u. We can prove in that case that $\models_{T_{\mathsf{S}}} v \geqslant u$. Dually, if u rewrites to v in $\rightarrow_U$, written $u \rightarrow_U^* v$, we say that v is *an under-approximation* of u and can prove that $\models_{T_{\mathsf{S}}} u \geqslant v$. Based on these definitions, the core of our inference system can be summarized by the single inference rule schema provided in Fig. 2 together with the conditional rewrite systems $\rightarrow_O$ and $\rightarrow_U$ which are defined inductively in terms of the inference system and each other.

A majority of the rewrite rules have side conditions requiring the derivability of certain judgments in the same inference system. To improve their readability we take some liberties with the notation and write $\vdash u_1 \geqslant u_2$, say, instead of $\vdash u_1 - u_2 \geqslant 0$. For example, $|\mathsf{substr}(t, v, w)|$ is under-approximated by w if it can be inferred that the interval from v to $v + w$ is a valid range of positions in string t, which is expressed by the side conditions $\vdash v \geqslant 0$ and $\vdash |t| \geqslant v + w$. Note that some arithmetic terms, such as $|\mathsf{substr}(t, v, w)|$, can be approximated in *multiple* ways—hence the need for a strategy for choosing the best approximation for arithmetic string terms, described later. The rules for polynomials are written modulo associativity of $+$ and state that a monomial $m \cdot v$ in them can be over- or under-approximated based on the sign of the coefficient m. For simplicity,

$$\frac{u \to_U^* n \quad n \geqslant 0}{\vdash u \geqslant 0} \quad \text{where}$$

$$
\begin{array}{rcll}
|t| & \to_U & 0 & \\
|\mathsf{substr}(t,v,w)| & \to_U & \begin{cases} w & \text{if } \vdash v \geqslant 0 \text{ and } \vdash |t| \geqslant v + w \\ |t| - v & \text{if } \vdash v \geqslant 0 \text{ and } \vdash v + w \geqslant |t| \end{cases} & \\
|\mathsf{replace}(t,s,r)| & \to_U & \begin{cases} |t| & \text{if } \vdash |r| \geqslant |s| \text{ or } \vdash |r| \geqslant |t| \\ |t| - |s| & \end{cases} & \\
|\mathsf{int.to.str}(v)| & \to_U & 1 & \text{if } \vdash v \geqslant 0 \\
\mathsf{indexof}(t,s,v) & \to_U & -1 & \\
\mathsf{str.to.int}(t) & \to_U & -1 & \\
m \cdot v + u' & \to_U & m \cdot w + u' & \text{if } v \to_U w \text{ and } m > 0 \text{ or } v \to_O w \text{ and } m < 0
\end{array}
$$

$$
\begin{array}{rcll}
|\mathsf{substr}(t,v,w)| & \to_O & w & \text{if } \vdash w \geqslant 0 \\
|\mathsf{substr}(t,v,w)| & \to_O & \begin{cases} |t| - v & \text{if } \vdash |t| \geqslant v \\ |t| & \end{cases} & \\
|\mathsf{replace}(t,s,r)| & \to_O & \begin{cases} |t| & \text{if } \vdash |s| \geqslant |r| \\ |t| + |r| & \end{cases} & \\
|\mathsf{int.to.str}(v)| & \to_O & \begin{cases} v & \text{if } \vdash v > 0 \\ v + 1 & \text{if } \vdash v \geqslant 0 \end{cases} & \\
\mathsf{indexof}(t,s,v) & \to_O & \begin{cases} |t| - |s| & \text{if } \vdash |t| \geqslant |s| \\ |t| & \end{cases} & \\
m \cdot v + u' & \to_O & m \cdot w + u' & \text{if } v \to_O w \text{ and } m > 0 \text{ or } v \to_U w \text{ and } m < 0
\end{array}
$$

Fig. 2. Rules for arithmetic entailment based on under- and over-approximations computed for arithmetic terms containing extended string operators. We write t, s, r to denote string terms, u, u', v, w to denote integer terms and m, n to denote integer constants.

we silently assume in the figure that basic arithmetic simplifications are applied after each rewrite step to put the right-hand side in polynomial form.

Example 3. Let u be $|\mathsf{replace}(x, \mathsf{aa}, \mathsf{b})|$. Because $\vdash |\mathsf{aa}| \geqslant |\mathsf{b}|$, the first case of the over-approximation rule for replace applies, and we get that $u \to_O |x|$. This reflects that the result of replacing the first occurrence, if any, of aa in x with b is no longer than x.

Example 4. Let u be the same as in the previous example and let v be $-1 \cdot u + 2 \cdot |x|$. Since $u \to_O |x|$ and the coefficient of u in v is negative, we have that $v \to_U -1 \cdot |x| + 2 \cdot |x|$, which simplifies to $|x|$; moreover, $|x| \to_U 0$. Thus, $v \to_U^* 0$ and so $\vdash v \geqslant 0$. In other words, we can use the approximations to show that u is at most $2 \cdot |x|$.

3.1 A Strategy for Approximation

The rewrite systems $\to_O$ and $\to_U$ allow for many possible derivations. Thus, it is important to devise a strategy that is efficient and succeeds often in practice. We use a greedy rule application strategy that favors rule applications leading to the cancellation of monomials. For example, consider the term $|x| - |\mathsf{substr}(y, 0, |x|)|$,

and observe that the subtrahend can be over-approximated either by $|y|$ or by $|x|$. However, proving the T_{S}-validity of $|x| - |\mathsf{substr}(y, 0, |x|)| \geqslant 0$ with the former over-approximation is impossible since $|x| - |y| \geqslant 0$ does not hold in all models of T_{S}. In contrast, the latter approximation produces $|x| - |x| \geqslant 0$ which is trivially T_{S}-valid.

STR-ARITH-APPROX(u), where $u = u_x + u_\ell + u_s + m$ and:
- $u_x = m_1^y \cdot y_1 + \ldots + m_n^y \cdot y_n$,
- $u_\ell = m_1^\ell \cdot |x_1| + \ldots + m_p^\ell \cdot |x_p|$,
- $u_s = m_1^v \cdot v_1 + \ldots + m_q^v \cdot v_q$.

for variables $x_1, \ldots, x_p, y_1, \ldots, y_n$ and extended terms $v_1, \ldots v_q$:

1. If $q > 0$, choose a v_i and v_i^a that maximize the following criteria (in descending order), where $u' = (u[m_i^v \cdot v_i \mapsto m_i^v \cdot v_i^a])\downarrow$:
   (a) (Soundness) $v_i \to_U v_i^a$ if $m_i^v > 0$ and $v_i \to_O v_i^a$ if $m_i^v < 0$;
   (b) (Avoids new terms) Minimizes the size of $\mathsf{negcoeff}(u') \backslash \mathsf{negcoeff}(u)$;
   (c) (Cancels existing terms) Maximizes the size of $\mathsf{negcoeff}(u) \backslash \mathsf{negcoeff}(u')$.
   Return $u \to_U u'$.
2. If $p > 0$ and $m_j^\ell > 0$ for some j, return $u \to_U (u[m_j^\ell \cdot |x_j| \mapsto 0])\downarrow$.

Fig. 3. A greedy strategy for showing arithmetic entailments in the theory T_{S}. We write $\mathsf{negcoeff}(u)$ to denote the set of terms whose coefficient is negative in u.

Recall that, given an arithmetic inequality $u \geqslant 0$, our goal is to find a reduction $u \to_U^* n$ where n is a non-negative constant. Our strategy for choosing which rule of $\to_U$ to apply to u is given in Fig. 3. We decompose u into three parts: the portion u_x consisting of a sum of integer variables, the portion u_ℓ consisting of a sum of lengths of string variables, and the remaining portion u_s which is a sum of monomials involving extended terms $v_1, \ldots, v_q$ as defined in Definition 1.

Since there are multiple choices for how terms in u_s are approximated, the strategy focuses primarily on this portion. In particular, we apply an approximation for one of the terms v_i, under-approximating or over-approximating depending on the sign of its coefficient, and replace the monomial in t by its corresponding approximation. The choice of v_i and v_i^a is based on maximizing the likelihood that the overall derivation will produce a non-negative constant.

For a term u in polynomial form, let $\mathsf{negcoeff}(u)$ be a set of integer terms whose coefficient is negative in u, e.g. $\mathsf{negcoeff}(y_1 + -1 \cdot y_2) = \{y_2\}$. Terms in this set can be seen as *obligations* for proving entailments in our derivations since if $y_2 \in \mathsf{negcoeff}(u)$, it must be the case that our derivation applies a rule that introduces a term with a positive coefficient for y_2. In Fig. 3, we say that our choice of $v_i \to_U v_i^a$ *avoids new terms* if it does not have the effect of adding any new terms to $\mathsf{negcoeff}(u)$, and *cancels existing terms* if it has the effect of removing terms from this set. If the portion u_s is empty, we apply the rule $|x_j| \to_U 0$ if there exists a monomial $m_j^\ell \cdot |x_j|$ where m_j^ℓ is positive. This rule is applied with lowest priority because these monomials may help to cancel negative terms introduced by the other steps.

Step 1 depends on knowing the set of possible one-step approximations $v_i \rightarrow_U v_i^a$ and $v_i \rightarrow_O v_i^a$ for terms from u. These are determined using the rules of Fig. 2. Whenever applicable, we break ties between rewrites in Step 1 by considering a fixed arbitrary ordering over extended terms.

Example 5. Let u be $1 + |t_1| + |t_2| - |x_1|$, where t_1 is $\mathsf{substr}(x_2, 1, |x_2| + |x_4|)$ and t_2 is $\mathsf{replace}(x_1, x_2, x_3)$. Step 1 of Str-Arith-Approx considers the possible approximations $|t_1| \rightarrow_U |x_2| - 1$ and $|t_2| \rightarrow_U |x_1| - |x_2|$. Note that under-approximations are needed because the coefficients of $|t_1|$ and $|t_2|$ are positive. The first approximation is an instance of the third rule in Fig. 2, noting that both $\vdash 1 \geqslant 0$ and $\vdash 1 + |x_2| + |x_4| \geqslant |x_2|$ are derivable by a *basic* strategy that, wherever applicable, under-approximates string length terms as zero. Our strategy chooses the first approximation since it introduces no new negative coefficient terms, thus obtaining: $u \rightarrow_U |x_2| + |t_2| - |x_1|$. We now choose the approximation $|t_2| \rightarrow_U |x_1| - |x_2|$, noting that it introduces no new negative coefficient terms and cancels an existing one, $|x_1|$. After arithmetic simplification, we have derived $u \rightarrow_U^* 0$, and hence $\vdash u \geqslant 0$.

One can show that our strategy is sound, terminating, and deterministic. This means that applying Str-Arith-Approx to completion produces a unique rewrite chain of the form $t \rightarrow_U u_1 \rightarrow_U \ldots \rightarrow_U u_n$ for a finite n, where each step is an application of one of the rewrite rules from Fig. 2.

3.2 Simplification Rules with Arithmetic Side Conditions

We use the inference system from the previous section for simplifications of string terms with arithmetic side conditions. Figure 4 summarizes those simplifications.

The first rule rewrites a string equality to $\bot$ if one of the two sides can be inferred to be strictly longer than the other. In the second rule, if one side of an equality, $\mathsf{con}(s, r, q)$, is such that the sum of lengths of s and q alone can be shown to be greater than or equal to the length of the other side, then r must be empty. The third rule recognizes that string containment reduces to string

$$
\begin{array}{rll}
t \approx s \rightarrow & \bot & \text{if } \vdash |t| \geqslant |s| + 1 \\
t \approx \mathsf{con}(s, r, q) \rightarrow & t \approx \mathsf{con}(s, q) \wedge r \approx \epsilon & \text{if } \vdash |s| + |q| \geqslant |t| \\
\mathsf{contains}(t, s) \rightarrow & t \approx s & \text{if } \vdash |s| \geqslant |t| \\
\mathsf{substr}(t, v, w) \rightarrow & \epsilon & \text{if } \vdash 0 > v \vee v \geqslant |t| \vee 0 \geqslant w \\
\mathsf{substr}(\mathsf{con}(t, s), v, w) \rightarrow & \mathsf{substr}(s, v - |t|, w) & \text{if } \vdash v \geqslant |t| \\
\mathsf{substr}(\mathsf{con}(s, t), v, w) \rightarrow & \mathsf{substr}(s, v, w) & \text{if } \vdash |s| \geqslant v + w \\
\mathsf{substr}(\mathsf{con}(t, s), 0, w) \rightarrow & \mathsf{con}(t, \mathsf{substr}(s, 0, w - |t|)) & \text{if } \vdash w \geqslant |t| \\
\mathsf{indexof}(t, s, v) \rightarrow & \mathsf{ite}(\mathsf{substr}(t, v) \approx s, v, -1) & \text{if } \vdash v + |s| \geqslant |t|
\end{array}
$$

Fig. 4. String simplification rules. Letters t, s, r, q denote string terms; v, w denote integer terms.

equality when it can be inferred that string s is at least as long as the string t that must contain it. The next rule captures the fact that substring simplifies to the empty string if it can be inferred that its position v is not within bounds, or its length w is not positive. In the figure, we write that rule with a disjunctive side condition; this is a shorthand to denote that we can pick any disjunct and show that it holds assuming the negation of the other disjuncts. We can use those assumptions to perform substitutions to simplify the derivation. Concretely, to show $\vdash u_1 \geqslant u_2 \vee \ldots \vee u \not\approx u'$ it is sufficient to infer $\vdash (u_1 \geqslant u_2)[u \mapsto u']$. We demonstrate this with an example.

Example 6. Consider the term $\mathsf{substr}(t, |t| + w, w)$. Our rules may simplify this term to ϵ by inferring that its start position $(|t| + w)$ is not within the bounds of t if we assume that its size (w) is positive. In detail, assume that $w > 0$ (the negation of the last disjunct in the side condition of the fourth rule), which is equivalent to $w \approx |x| + 1$ where x is a fresh string variable and $|x|$ denotes an unknown non-negative quantity. It is sufficient to derive the formula obtained by replacing all occurrences of w by $|x| + 1$ in the disjunct $|t| + w \geqslant |t|$ to show that the start position of our term is out of bounds. After simplification, we obtain $|x| + 1 \geqslant 0$, which is trivial to derive.

The next two rules in Fig. 4 apply if we can infer respectively that the start position of the substring comes strictly after a prefix t or that the end position of the substring comes strictly before a suffix t of the first argument string. In either case, t can be dropped.

Example 7. Let t be $\mathsf{substr}(\mathsf{con}(x_1, \mathsf{replace}(x_2, x_3, x_4)), 0, w)$, where w is $|x_1| - |x_2|$. We have that $t \rightarrow \mathsf{substr}(x_1, 0, w)$, noting that $\vdash |x_1| \geqslant 0 + |x_1| - |x_2|$. In other words, only the first component x_1 of the string concatenation is relevant to the substring since its end point must occur before the end of x_1.

The final rule for substr shows that a prefix of a substring can be pulled upwards if the start position is zero and we can infer that the substring is guaranteed to include at least a prefix string t. Finally, if we can infer that the last position of s in t starting from position v is at or beyond the end of t, then the indexof term can be rewritten as an if-then-else (ite) term that checks whether s is a suffix of t.

4 Containment-Based String Simplification

This section provides an overview of simplifications that are based on reasoning about the containment relationship between strings. We describe an inference system for deriving when one string is definitely contained or not contained in another. Following the notation from the last section, we write $\vdash t \ni s$ to denote the judgment of our inference system, denoting that string t contains string s in all models of T_{S}. Conversely, we write $\vdash t \not\ni s$ to denote string t does not contain string s. We write $\vdash t \ni^p s$ (resp., $\vdash t \ni^s s$) to denote the judgment indicating that s must be a prefix (resp., suffix) of t.

$$\frac{l_1 \text{ contains } l_2}{\vdash \mathsf{con}(l_1, t) \ni l_2} \qquad \frac{\vdash s \ni r}{\vdash \mathsf{con}(t, s) \ni r} \qquad \frac{\vdash t \ni^s r \quad \vdash s \ni^p q}{\vdash \mathsf{con}(t, s) \ni \mathsf{con}(r, q)} \qquad \frac{}{\vdash t \ni \mathsf{substr}(t, v, w)}$$

$$\frac{l_1 \text{ does not contain } l_2}{\vdash l_1 \notin \mathsf{con}(l_2, t)} \qquad \frac{\vdash r \notin t}{\vdash r \notin \mathsf{con}(s, t)} \qquad \frac{\vdash l_1 \setminus l_2 \notin t}{\vdash l_1 \notin \mathsf{con}(l_2, t)}$$

$$\frac{l_2 \text{ is a prefix of } l_1}{\vdash \mathsf{con}(l_1, t) \ni^p l_2} \qquad \frac{\vdash s \ni^p r}{\vdash \mathsf{con}(t, s) \ni^p \mathsf{con}(t, r)} \qquad \frac{}{\vdash t \ni^p t} \qquad \frac{\vdash v \leqslant 0}{\vdash t \ni^p \mathsf{substr}(t, v, w)}$$

$$\frac{l_2 \text{ is a suffix of } l_1}{\vdash \mathsf{con}(t, l_1) \ni^s l_2} \qquad \frac{\vdash s \ni^s r}{\vdash \mathsf{con}(s, t) \ni^s \mathsf{con}(r, t)} \qquad \frac{}{\vdash t \ni^s t} \qquad \frac{\vdash v + w \geqslant |t|}{\vdash t \ni^s \mathsf{substr}(t, v, w)}$$

Fig. 5. Inferences for string containment $\ni$, is-prefix $\ni^p$ and is-suffix $\ni^s$.

Rules for inferring judgments of these forms are given in Fig. 5. Like our rules for arithmetic, these rules are solely based on the syntactic structure of terms, so inferences in this system can be computed statically. Both the assumptions and conclusions of the rules assume associativity of string concatenation with identity element ϵ, that is, $\mathsf{con}(t, s)$ may refer to a term of the form $\mathsf{con}(\mathsf{con}(t_1, t_2), s) = \mathsf{con}(t_1, t_2, s)$ or alternatively to $\mathsf{con}(\epsilon, s) = s$. Most of the rules are straightforward. The inference system has special rules for substring terms $\mathsf{substr}(t, v, w)$, using arithmetic entailments from Sect. 3 to show prefix and suffix relationships with the base string t. For negative containment, the rules of the inference system together can show a (possibly non-constant) string cannot occur in a constant string by reasoning that its characters cannot appear in order in that string. We write $l_1 \setminus l_2$ to denote the empty string if l_1 does not contain l_2, or the result of removing the smallest prefix of l_1 that contains l_2 from l_1 otherwise.

Example 8. Let t be abcab and let s be $\mathsf{con}(\mathsf{b}, x, \mathsf{a}, y, \mathsf{c})$. String s is not contained in t for any value of x, y. We derive $\vdash t \notin s$ using two applications of the rightmost rule for negative containment in Fig. 5, noting $\mathsf{abcab} \setminus \mathsf{b} = \mathsf{cab}$, $\mathsf{cab} \setminus \mathsf{a} = \mathsf{b}$, and b does not contain c. In other words, the containment does not hold since the characters b, a and c cannot be found in order in the constant abcad.

4.1 Simplification Rules Based on String Containment

Figure 6 gives rules for simplifying extended function terms based on the aforementioned judgments pertaining to string containment. First, equalities can be rewritten to false and applications of $\mathsf{contains}$ can be rewritten to a constant based on the appropriate judgment of our inference system. Applications of $\mathsf{indexof}$ can be simplified to -1 if it can be shown that the second argument does not appear in the suffix of the first argument starting at the position given by the third argument. The next two rules reason about cases where the second argument s definitely occurs in the first argument starting from position v. In this case, if we additionally know that s occurs within (beyond) a prefix t of

$$
\begin{array}{rll}
t \approx s \rightarrow & \bot & \text{if } \vdash t \not\ni s \\
\mathsf{contains}(t,s) \rightarrow & \bot & \text{if } \vdash t \not\ni s \\
\mathsf{contains}(t,s) \rightarrow & \top & \text{if } \vdash t \ni s \\
\mathsf{indexof}(t,s,v) \rightarrow & -1 & \text{if } \vdash \mathsf{substr}(t,v) \not\ni s \\
\mathsf{indexof}(\mathsf{con}(t,r),s,v) \rightarrow & \mathsf{indexof}(t,s,v) & \text{if } \vdash \mathsf{substr}(t,v) \ni s \\
\mathsf{indexof}(\mathsf{con}(t,r),s,v) \rightarrow & \mathsf{indexof}(r,s,v-|t|)+|t| & \text{if } \vdash \mathsf{substr}(\mathsf{con}(t,r),v) \ni s \text{ and} \\
 & & \quad \vdash v \geqslant |t| \\
\mathsf{indexof}(t,s,v) \rightarrow & v & \text{if } \vdash \mathsf{substr}(t,v) \ni^{p} s \text{ and } \vdash v < |t| \\
\mathsf{replace}(t,s,r) \rightarrow & t & \text{if } \vdash t \not\ni s \\
\mathsf{replace}(\mathsf{con}(t,q),s,r) \rightarrow & \mathsf{con}(\mathsf{replace}(t,s,r),q) & \text{if } \vdash t \ni s \\
\mathsf{replace}(t,s,r) \rightarrow & \mathsf{con}(r,\mathsf{substr}(t,|s|)) & \text{if } \vdash t \ni^{p} s
\end{array}
$$

Fig. 6. Simplification rules based on string containment.

the first argument, then the suffix r (prefix t) can be dropped, where the start position and the return value of the result are modified accordingly. If we know s is a prefix of the first argument at position v, then the result is v if indeed v is in the bounds of t. Notice that the latter condition is necessary to handle the case where s is the empty string. The three rules for replace are analogous. First, the replace rewrites to the first argument if we know it does not contain the second argument s. If we know s is definitely contained in a prefix of the first argument, then we can pull the remainder of that string upwards. Finally, if we know s is a prefix of the first argument, then we can replace that prefix with r while concatenating the remainder. We use the term $\mathsf{substr}(t,|s|)$ to denote the remainder after the replacement for the sake of brevity, although this term typically does not involve extended functions after simplification, e.g. $\mathsf{replace}(\mathsf{con}(x,y),x,z) \rightarrow \mathsf{con}(z,y)$ noting that $(\mathsf{substr}(\mathsf{con}(x,y),|x|))\downarrow = y$, or $\mathsf{replace}(\mathsf{ab},\mathsf{a},x) \rightarrow \mathsf{con}(x,\mathsf{b})$ noting that $(\mathsf{substr}(\mathsf{ab},|\mathsf{a}|))\downarrow = \mathsf{b}$.

4.2 Simplifications Based on Equivalence of String Containment

We further refine our approach based on inferring when one containment is *equivalent* to another one. For example, $\mathsf{con}(\mathsf{a},x)$ is contained in $\mathsf{con}(\mathsf{b},y)$ if and only if $\mathsf{con}(\mathsf{a},x)$ is contained in y alone. We introduce simplifications for such equivalences by reasoning about the maximal overlap between two strings.

We adapt and extend the notation given in previous work [15]. Given string literals l_1 and l_2, the *sufficient left overlap* of l_1 and l_2, written $l_1 \sqcup_l l_2$, is the largest suffix of l_1 that is a prefix of l_2 or has l_2 as a prefix. For example, we have $\mathsf{abc} \sqcup_l \mathsf{cd} = \mathsf{c}$, $\mathsf{abc} \sqcup_l \mathsf{b} = \mathsf{bc}$, and $\mathsf{abc} \sqcup_l \mathsf{ba} = \epsilon$. We extend this definition to arbitrary strings s such that $l_1 \sqcup_l s$ is equivalent to $l_1 \sqcup_l l_2$ for the largest constant prefix l_2 of s, where notice that l_2 is the empty string if s does not have a constant prefix. For example, we have $\mathsf{abc} \sqcup_l \mathsf{con}(\mathsf{cde},y) = \mathsf{c}$, $\mathsf{abc} \sqcup_l \mathsf{con}(\mathsf{b},y) = \mathsf{bc}$, and $\mathsf{abc} \sqcup_l \mathsf{con}(\mathsf{a},y) = \mathsf{abc}$. We define the dual operator *sufficient right overlap*, written $l_1 \sqcup_r l_2$, which is the largest prefix of l_1 that is a suffix of l_2 or has l_2 as a suffix, e.g. $\mathsf{abc} \sqcup_r \mathsf{b} = \mathsf{ab}$, and extend this to arbitrary strings in an analogous way. The sufficient left (resp., right) overlap operator can be used to determine

how much of a constant string prefix l_1 (resp., suffix) can be safely removed from a string without impacting whether it contains another string.

$$
\begin{array}{rll}
\mathsf{contains}(\mathsf{con}(t,l),s) \rightarrow & \mathsf{contains}(\mathsf{con}(t, l \sqcup_r s), s) & \\
\mathsf{contains}(\mathsf{con}(l,t),s) \rightarrow & \mathsf{contains}(\mathsf{con}(l \sqcup_l s, t), s) & \\
\mathsf{indexof}(\mathsf{con}(t,l),s,v) \rightarrow & \mathsf{indexof}(\mathsf{con}(t, l \sqcup_r s), s, v) & \\
\mathsf{indexof}(\mathsf{con}(l,t),s,v) \rightarrow & \mathsf{indexof}(\mathsf{con}(l_2,t),s,v-|l_1|) & \text{if } l = l_1 \cdot l_2 \text{ and } l_2 = l \sqcup_l s \\
 & +|l_1| & \vdash \mathsf{substr}(\mathsf{con}(l,t),v) \ni s \\
\mathsf{replace}(\mathsf{con}(t,l),s,r) \rightarrow & \mathsf{con}(\mathsf{replace}(\mathsf{con}(t,l_1),s,r),l_2) & \text{if } l = l_1 \cdot l_2 \text{ and } l_1 = l \sqcup_r s \\
\mathsf{replace}(\mathsf{con}(l,t),s,r) \rightarrow & \mathsf{con}(l_1,\mathsf{replace}(\mathsf{con}(l_2,t),s,r)) & \text{if } l = l_1 \cdot l_2 \text{ and } l_2 = l \sqcup_l s
\end{array}
$$

Fig. 7. Simplification rules based on equivalence of string containment. We write l, l_1, l_2 to denote string literals, v, w to denote integer terms and t, s to denote string terms.

The rules in Fig. 7 simplify extended terms by considering string overlaps. The first two rules drop parts of string literals from the suffix or prefix of their first arguments. The two rules for indexof are similar: a suffix of the first argument can be dropped if it does not contribute to whether it contains the second argument. A prefix of an indexof term can be dropped if it does not contribute to containment, but only in the case where we know the second argument is definitely contained in the first argument. This is to guard against the case where the entire indexof term returns -1. The rules for replace are similar to those for contains, except that the suffix (resp., prefix) of the first argument is pulled upwards instead of being dropped.

5 Multiset-Based String Simplification

Next, we introduce simplifications based on reasoning about strings as multisets, i.e. collections of unordered characters. Such reasoning is sufficient for showing that equalities like $\mathsf{con}(\mathsf{a}, x) \approx \mathsf{con}(x, \mathsf{b})$ are equivalent to $\bot$, since the left side of the equality contains exactly one more occurrence of character a than the right-hand side. Similar to arithmetic reasoning from Sect. 3, we use approximations when reasoning about strings as multisets. We define the *multiset abstraction* of t, written $\mathcal{M}_t$, as the multiset $\{t_1, \ldots, t_n\}$ where t is equivalent to $\mathsf{con}(t_1, \ldots, t_n)$ and all constants in this set are characters. For example, $\mathcal{M}_{\mathsf{con}(\mathsf{aba},x)} = \{\mathsf{a}, \mathsf{a}, \mathsf{b}, x\}$. We define a rewrite system $\rightarrow_O^{\mathcal{M}}$ over strings where a rewritten string over-approximates the original string in the following sense: if $t \rightarrow_O^{\mathcal{M}} s$, then for all models of T_S and any character c, the number of occurrences of c in the strings in $\mathcal{M}_s$ is greater than or equal to the number of occurrences in the strings in $\mathcal{M}_t$.

Figure 8 lists the rules for the rewrite system $\rightarrow_O^{\mathcal{M}}$ and the simplifications based on multiset reasoning. Given a predicate $\mathsf{contains}(t, s)$, if over-approximating t with respect to the rules of $\rightarrow_O^{\mathcal{M}}$ results in a string r, and it can be determined that s contains strictly more occurrences of some character

c than r, then it cannot be the case that s is contained in t. To establish this, we check whether the multiset difference of $\mathcal{M}_s$ and $\mathcal{M}_r$ contains c, and conversely the difference of $\mathcal{M}_r$ and $\mathcal{M}_s$ contains only character constants which are distinct from c. In the second rule, if one side of an equality can be determined to contain *only* a character c, then one occurrence of that character can be dropped from both sides of the equality, since the relative position of that character does not matter. The three rules for $\rightarrow_O^{\mathcal{M}}$ state that the multiset abstraction of a term of the form $\mathsf{substr}(t, v, w)$ can be over-approximated as the entire string t; a term $\mathsf{replace}(t, s, r)$ can be over-approximated as a string having both t and r; and over-approximation can be applied to the children of con terms.

$$\begin{aligned}
\mathsf{contains}(t, s) &\rightarrow \bot && \text{if } t \rightarrow_O^{\mathcal{M}} {}^*r,\\
& && \mathcal{M}_s \backslash \mathcal{M}_r = \{c, s_1, \ldots, s_n\} \text{ and}\\
& && \mathcal{M}_r \backslash \mathcal{M}_s = \{c_1, \ldots, c_m\}\\
\mathsf{con}(t, c, s) \approx \mathsf{con}(q, c, r) &\rightarrow \mathsf{con}(t, s) \approx \mathsf{con}(q, r) && \text{if } \mathcal{M}_{\mathsf{con}(t,c,s)} \rightarrow_O^{\mathcal{M}} {}^*p \text{ and}\\
& && \mathcal{M}_p = \{c, \ldots, c\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{substr}(t, v, w) &\rightarrow_O^{\mathcal{M}} t\\
\text{where } \mathsf{replace}(t, s, r) &\rightarrow_O^{\mathcal{M}} \mathsf{con}(t, r)\\
\mathsf{con}(t, s, r) &\rightarrow_O^{\mathcal{M}} \mathsf{con}(t, q, r) \text{ if } s \rightarrow_O^{\mathcal{M}} q
\end{aligned}$$

Fig. 8. Simplification rules based on multiset reasoning. We write $c, c_1, \ldots$ to denote characters, v, w to denote integer terms, and t, s, r, q, p to denote string terms.

Example 9. We have that $\mathsf{con}(\mathsf{aaa}, \mathsf{substr}(x, y_1, y_2)) \approx \mathsf{con}(x, \mathsf{b}) \rightarrow \bot$ by noting that $\mathsf{con}(\mathsf{aaa}, \mathsf{substr}(x, y_1, y_2)) \rightarrow_O^{\mathcal{M}} {}^*\mathsf{con}(\mathsf{aaa}, x)$, $\mathcal{M}_{\mathsf{con}(\mathsf{aaa},x)} = \{\mathsf{a}, \mathsf{a}, \mathsf{a}, x\}$ and $\mathcal{M}_{\mathsf{con}(x,\mathsf{b})} = \{\mathsf{b}, x\}$. The difference of the latter with the former is $\{\mathsf{b}\}$, and the former with the latter is $\{\mathsf{a}, \mathsf{a}, \mathsf{a}\}$. Thus, the right side of the equality contains at least one more occurrence of b than the left side; hence, the equality is equivalent to false.

6 Implementation

We implemented the above simplification rules and others in the DPLL-based SMT solver CVC4, which implements a theory solver for a basic fragment of word equations with length, several other theory solvers, and reduction techniques for extended string functions as described in Sect. 2.1. Our simplification rules are run in a *preprocessing* pass as well as an *inprocessing* pass during solving. For the latter, we use a context-dependent simplification strategy that infers when an extended string constraint, e.g., $\neg\mathsf{contains}(t, s)$, simplifies to $\bot$ based on other assertions, e.g., $s \approx \epsilon$. Our simplification techniques do not affect the core procedure for the theory of strings, nor the compatibility of the string solver with other theories. In total, our implementation is about 3,500 lines of C++ code. We cache the results of the simplifications and the approximation-based arithmetic entailments to amortize their costs.

Additional Simplification Rules. The simplification rules in this paper are a subset of the rules in the implementation. We omit other uncategorized rules for lack of space. Many of these apply to specific term patterns, such as cases where two nested applications of substr can be combined; cases where an application of replace can be eliminated by case splitting; and other cases like $\mathsf{con}(t,t) \approx \mathsf{a} \to \bot$. An example of such rules is $\mathsf{contains}(\mathsf{replace}(t,w_1,w_2),w_3) \to \mathsf{contains}(t,w_3)$ if w_3 does not overlap with either w_1 or w_2, because the replace does not change whether t contains w_3 or not. Another class of rules only applies to strings of length one because they cannot span multiple components of a concatenations, e.g. $\mathsf{contains}(\mathsf{con}(t,s),c) \to \mathsf{contains}(t,c) \vee \mathsf{contains}(s,c)$ where c is a character. Finally, there are rewrites that benefit from multiple techniques presented in this paper. For example, we have a rewrite that splits string equations into multiple smaller equations if it can determine that prefixes must have the same length: $\mathsf{con}(\mathsf{a},t,s) \approx \mathsf{con}(t,\mathsf{b},r) \to \mathsf{con}(\mathsf{a},t) \approx \mathsf{con}(t,\mathsf{b}) \wedge s \approx r \to \bot$.

Validating Simplification Rules. The correctness of our simplification techniques is critical to the soundness of the overall solver. Due to the sophistication and breadth of those techniques, it is challenging to formally verify our implementation. As a pragmatic alternative, we periodically test our implementation using a testing infrastructure we developed for this purpose. We found this to be critical in our development process. Our testing infrastructure allows the developer to specify a context-free grammar in the syntax-guided synthesis format [2]. We generate all terms t in this grammar up to a fixed size and test the equivalence of t and its simplified form $t{\downarrow}$ on a set of randomly generated points. The most recent run of this system on two grammars (one for extended string terms and another for string predicates) up to a term size of three, validated 319,867 simplifications of string terms and 188,428 simplifications of string predicates on 1,000 sample points. This run took 924 s for string terms and 971 s for the string predicates using the same hardware as in Sect. 7.

7 Evaluation

We evaluate the impact of each simplification technique as implemented in CVC4 on three benchmark sets that use extended string operators: CMU, a dataset obtained from symbolic execution of Python code [15]; TermEq, a benchmark set consisting of the verification of term equivalences over strings [14]; and Slog, a benchmark set extracted from vulnerability testing of web applications [22]. The Slog set uses the replace function extensively but does not contain other extended functions. We also evaluate the impact on Aplas, a set of handcrafted benchmarks involving looping word equations [10] (string equalities whose left and right sides have variables in common).

We compare CVC4 with Z3 commit `9cb1a0f` [8],[4] a state-of-the-art string solver. Additionally, we compare against Ostrich on the Slog benchmarks but not other sets because it does not support some functions such as contains and

[4] `9cb1a0f` is newer than the current release 4.8.4 and includes several fixes for critical issues.

indexof. We omit a comparison with Z3STR3 4.8.4 because we found multiple issues in its latest release including wrong answers, which we have reported to the authors. We also omit a comparison with S3# due to differing semantics. We compare four configurations of CVC4: **all**, which enables all optimizations; **-arith**, which disables arithmetic-based simplification techniques (discussed in Sect. 3); **-contain**, which disables containment-based simplification techniques (discussed in Sect. 4); and **-msets**, which disables multiset-based simplification techniques (discussed in Sect. 5). Additionally, to test the applicability of our techniques to other solvers, we test the effect of our simplifications on Z3 by using CVC4 to generate simplified benchmarks and then running Z3 on those benchmarks. We generate a set of simplified benchmarks that are simplified with CVC4 with ($Z3_f$) and without ($Z3_b$) the simplification techniques presented in this paper.

Table 1. Number of solved problems per benchmark set. Best results are in **bold**. Gray cells indicate benchmark sets not supported by a solver. "R%" indicates the reduction of extended string functions during preprocessing. All benchmarks ran with a timeout of 600 s.

Set		**all**	**-arith**	**-contain**	**-msets**	Z3	$Z3_b$	$Z3_f$	OSTRICH	R%
CMU	sat	**5703**	5535	**5703**	**5703**	2343	3923	3943		32%
	unsat	**65**	29	**65**	**65**	50	58	61		
	×	154	358	154	154	3529	1941	1918		
TERMEQ	sat	**10**	**10**	**10**	**10**	4	5	5		68%
	unsat	51	37	28	51	35	40	**60**		
	×	19	33	42	19	41	35	15		
SLOG	sat	1302	1302	1302	1302	1133	1225	1225	**1304**	27%
	unsat	**2082**	**2082**	**2082**	**2082**	2080	2080	2080	**2082**	
	×	7	7	7	7	178	86	86	5	
APLAS	sat	**135**	**135**	**135**	**135**	9	51	46		n/a
	unsat	**292**	**292**	171	171	94	129	**292**		
	×	159	159	280	280	483	406	248		
Total	sat	**7150**	6982	**7150**	**7150**	3489	5204	5219	1304	
	unsat	2490	2440	2346	2369	2259	2307	**2493**	2082	
	×	339	557	483	460	4231	2468	2267	5	

We ran all benchmarks on a cluster equipped with Intel E5-2637 v4 CPUs running Ubuntu 16.04 and dedicated one core, 8 GB RAM, and 600 s for each job. Table 1 summarizes the number of solved instances for each configuration and the baseline solvers grouped by benchmark sets. We remark that the average reduction of extended string functions (with all simplification techniques enabled) shown in column "R%" is significant on all benchmark sets. The scatter plots in Fig. 9 detail the effects of disabling each family of simplifications. They distinguish between satisfiable and unsatisfiable instances. To emphasize

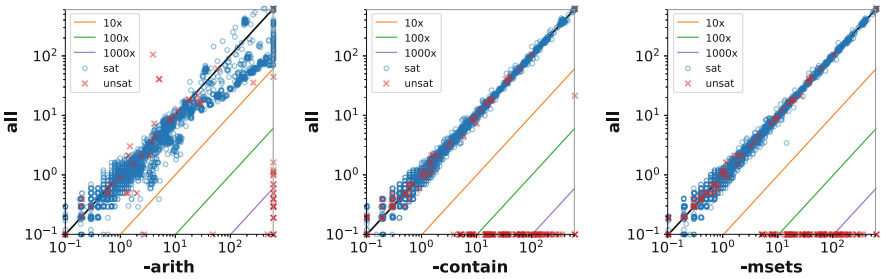

Fig. 9. Scatter plots showing the impact of disabling simplification techniques in CVC4 on both satisfiable and unsatisfiable benchmarks. All benchmarks ran with a timeout of 600 s.

non-trivial benchmarks, we omit the benchmarks that are solved in less than a second by all solvers.

The arithmetic-based simplification techniques have the most significant performance impact on the symbolic execution benchmarks CMU. The number of solved benchmarks is significantly lower when disabling those techniques. The scatter plot shows that for longer running satisfiable queries there is a large portion of the benchmarks that are solved up to an order of magnitude faster with the simplifications. These improvements in runtime on the CMU set are particularly compelling because they come from a symbolic execution application, which involves a large number of queries with a short timeout. The improvements are more pronounced for unsatisfiable benchmarks, where our results show that simplifications often give the solver the ability to derive a refutation in a matter of seconds, something that is infeasible with configurations without these techniques. The APLAS set contains no extended string operators and hence our arithmetic-based simplification techniques have little impact on this set.

In contrast, both containment and multiset-based rewrites have a high impact on the APLAS set, as **-contain** and **-msets** both solve 121 fewer benchmarks. Additionally, **-contain** has a high impact on the TERMEQ set, where the simplifications enable the best configuration to solve 61 out of 80 benchmarks. Since these techniques apply most frequently to looping word equations, they are less important for the CMU set, which does not have such equations. The containment-based and multiset-based techniques primarily help on unsatisfiable benchmarks, as shown in the scatter plots. On TERMEQ benchmarks, it tends to be easier to find counterexamples, i.e. to solve the satisfiable ones, so there is more to gain on unsatisfiable benchmarks.

On SLOG, OSTRICH solves two more instances than CVC4 but CVC4 is over 50 times faster on commonly solved instances while supporting a richer set of string operators. On all benchmark sets, CVC4 solves at least as many benchmarks as Z3 and CVC4 has 12× fewer timeouts than Z3. On the simplified benchmarks, Z3 performs significantly better. On the CMU and the APLAS benchmarks, $Z3_b$ outperforms Z3 by a large margin. Additionally simplifying the benchmarks with

the techniques presented in this paper improves performance further on most benchmark sets and allows Z3_f to solve the most unsatisfiable benchmarks overall. These results indicate that Z3 could benefit from additional simplifications, and they underscore the importance of curating and publishing simplification techniques in order to improve the state-of-the-art.

8 Conclusion

We have presented a set of aggressive simplification techniques for reasoning about extended string constraints. Our results suggest that such techniques are key to advancing the state of the art in SMT string solving. Arithmetic-based simplifications lead to significant speedups in benchmarks from a symbolic execution application, while containment and multiset-based simplifications improve the performance on problems consisting of difficult term equivalences and looping word equations. Our approach is not limited to CVC4 and can be adapted to other solvers.

Given the encouraging results for each of the simplification techniques in our evaluation, we plan to extend them to other types of abstraction and make them context-aware. The latter extension involves taking into account other assertions when checking whether a side condition of a rule is fulfilled.

Acknowledgements. This work was partially supported by the National Science Foundation under award 1656926, the Defense Advanced Research Projects Agency under award FA8650-18-2-7854, and Amazon Web Services.

References

1. Abdulla, P.A., et al.: TRAU: SMT solver for string constraints. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, 30 October–2 November 2018, pp. 1–5. IEEE (2018)
2. Alur, R., et al.: Syntax-guided synthesis. In: Irlbeck, M., Peled, D.A., Pretschner, A. (eds.) Dependable Software Systems Engineering. NATO Sciencefor Peace and Security Series, D: Information and Communication Security, vol. 40, pp. 1–25. IOS Press (2015)
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 55–59. IEEE (2017)
5. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_27

6. Chaudhuri, S., Farzan, A. (eds.): Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9779. Springer, Switzerland (2016). https://doi.org/10.1007/978-3-319-41528-4
7. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. PACMPL **3**(POPL), 49:1–49:30 (2019)
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for word equations over strings, regular expressions, and context-free grammars. ACM Trans. Softw. Eng. Methodol. **21**(4), 25:1–25:28 (2012)
10. Le, Q.L., He, M.: A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In: Ryu, S. (ed.) APLAS 2018. LNCS, vol. 11275, pp. 350–372. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02768-1_19
11. Li, G., Ghosh, I.: PASS: string solving with parameterized array and interval automaton. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 15–31. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03077-7_2
12. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 646–662. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_43
13. 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. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-63387-9
14. Reynolds, A., et al.: Rewrites for SMT solvers using syntax-guided enumeration. SMT (2018)
15. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar and Kuncak [13], pp. 453–474
16. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for Javascript. In: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16–19 May 2010, Berleley/Oakland, California, USA, pp. 513–528. IEEE Computer Society (2010)
17. Tinelli, C., Barrett, C., Fontaine, P.: Unicode Strings (Draft 1.0) (2018). http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml
18. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in webapplications. In: Ahn, G., Yung, M., Li, N. (eds.) Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014, pp. 1232–1243. ACM (2014)
19. Trinh, M.T., Chu, D.H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Chaudhuri and Farzan [6], pp. 218–240
20. Trinh, M.T., Chu, D.H., Jaffar, J.: Model counting for recursively-defined strings. In: Majumdar and Kuncak [13], pp. 399–418
21. Veanes, M., Tillmann, N., de Halleux, J.: Qex: symbolic SQL query explorer. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 425–446. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_24

22. Wang, H.E., Tsai, T.L., Lin, C.H., Yu, F., Jiang, J.H.R.: String analysis via automata manipulation with logic circuit representation. In: Chaudhuri and Farzan [6], pp. 241–260
23. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: an automata-based string analysis tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_13
24. Zheng, Y.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Form. Methods Syst. Des. **50**(2–3), 249–288 (2017)
25. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a z3-based string solver for web application analysis. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, Saint Petersburg, Russian Federation, 18–26 August 2013, pp. 114–124. ACM (2013)