A Decision Procedure for String to Code Point Conversion

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. In text encoding standards such as Unicode, text strings are sequences of *code points*, each of which can be represented as a natural number. We present a decision procedure for a concatenation-free theory of strings that includes length and a conversion function from strings to integer code points. Furthermore, we show how many common string operations, such as conversions between lowercase and uppercase, can be naturally encoded using this conversion function. We describe our implementation of this approach in the SMT solver CVC4, which contains a high-performance string subsolver, and show that the use of a native procedure for code points significantly improves its performance with respect to other state-of-the-art string solvers.

1 Introduction

String processing is an important part of many kinds of software. In particular, strings often serve as a common representation for the exchange of data at interfaces between different programs, between different programming languages, and between programs and users. At such interfaces, strings often represent values of types other than strings, and developers have to be careful to sanitize and parse those strings correctly. This is a challenging task, making the ability to automatically reason about such software and interfaces appealing. Applications of automated reasoning about strings include finding or proving the absence of SQL injections and XSS vulnerabilities in web applications [27,30,33], reasoning about access policies in cloud infrastructure [7], and generating database tables from SQL queries for unit testing [31]. To make this type of automated reasoning scalable, several approaches for reasoning natively about string constraints have been proposed [3,4,11,20,21].

To reason about complex string operations such as conversions between strings and numeric values, string solvers typically reduce these operations to operations in some basic fragment of the theory of strings which they support natively. The scalability of a string solver thus depends on the efficiency of the

This work was partially funded by Amazon Web Services.

N. Peltier and V. Sofronie-Stokkermans (Eds.): IJCAR 2020, LNAI 12166, pp. 218–237, 2020.
https://doi.org/10.1007/978-3-030-51074-9_13

reductions as well as the performance of the solver over the basic constraints. In such approaches, the set of operations in the basic fragment of strings has to be chosen carefully. If the set is too extensive, the implementation becomes complex and its performance as well as its correctness may suffer as a result. On the other hand, if the set is too restrictive, the reductions may become too verbose or only approximate, also leading to suboptimal performance. In current string solvers, basic constraints typically include only *word equations* (i.e., equalities between concatenations of variables and constants) and length constraints. Certain operations, however, such as conversions between strings and numeric values, cannot be represented efficiently in this fragment because the encoding requires reasoning by cases on the concrete characters that may occur in the string values assigned to a string variable.

In this work, we investigate extending the set of basic operators supported in a modern string solver to bridge the gap between character and integer domains. We assume a finite character domain of some cardinality n and, similarly to the Unicode standard, we assume a bijective mapping between its character set and the first n natural numbers which associates each character with a unique *code point*. We introduce then a new string operator, code, from characters strings to integers which can be used to encode the code point value of strings of length one and, more generally, reason about the code point of any character in a string. We propose an approach that involves extending a previous decision procedure with native support for this operator, obtaining a new decision procedure which avoids splitting on character values. Using the code operator, we can succinctly represent string operations including common string transducers, conversion between strings and integers, lexicographic ordering on strings, and regular expression membership constraints involving character ranges. We have implemented our proposed decision procedure in the state-of-the-art SMT solver CVC4 as an extension of its decision procedure for word equations by Liang et al. [21]. We have modified CVC4's reductions to take advantage of code. Using benchmarks generated by the concolic execution of Python code, we show that our technique provides significant benefits compared to doing case splitting on values.

To summarize, our contributions are as follows:

- We provide a decision procedure for a simple set of string operations containing length and a code point conversion function code, and prove its correctness. We describe how it can be combined with existing procedures for other string operators.
- We demonstrate how the code operator can be used in the reductions of several classes of useful string constraints.
- We implement and evaluate our approach in CVC4, showing that it leads to significant performance gains with respect to the state of the art.

In the following, we discuss related work. We then describe a fragment of the theory of strings in Sect. 2 that includes code. In Sect. 3, we provide a decision procedure for this fragment, prove its correctness, and describe how it can be integrated with existing decision procedures. Finally, we discuss applications

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

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

of reasoning about code points in Sect. 4 and evaluate our implementation in Sect. 5.

Related Work. The study of the decidability of different fragments of string constraints has a long history. We know that solvability of word equations over unbounded strings is decidable [23], whereas the addition of quantifiers makes the problem undecidable [25]. The boundary between decidable and undecidable fragments, however, remains unclear—a long-standing open question is whether word equations combined with equalities over string lengths are decidable [14]. Adding extended string operators such as $\mathsf{replace}$ [13] or conversions between strings and integers [18] leads to undecidability. Weakly chaining string constraints make up one decidable fragment. This fragment requires that the graph of relational constraints appearing in the constraints only contains limited types of cycles. It generalizes the straight-line fragment [22], which disallows equalities between initialized string variables, and the acyclic fragment [5], which disallows equalities involving multiple occurrences of a string variable and does not include transducers.

In practice, string solvers have to deal with undecidable fragments or fragments of unknown decidability, so current solvers for strings such as CVC4 [9], Z3 [16], Z3STR3 [11] and TRAU [3] implement efficient semi-decision procedures. In this work, we present a decision procedure that can be combined modularly with those procedures.

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, (quantified) formula, and free variable (see, e.g., [17]). We consider many-sorted signatures Σ that contain an (infix) logical symbol $\approx$ for equality—which has type $\sigma \times \sigma$ for all sorts σ in Σ and is always interpreted as the identity relation. 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. A Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. Given a (set of) terms S, we write $\mathcal{T}(S)$ to denote the set of subterms of S. By convention and unless otherwise stated, we use letters x, y, z to denote variables and s, t to denote terms.

We consider a theory T_{AS} of strings with length and code point functions, with a signature Σ_{AS} given in Fig. 1. We *fix a finite totally ordered set* $\mathcal{A}$ *of characters as our alphabet* and define T_{AS} as a set of Σ_{AS}-structures with universe $\mathcal{A}^*$ (the set of all words over $\mathcal{A}$) which differ only on the value they assign to variables. The signature includes the sorts Str and Int, interpreted as $\mathcal{A}^*$ and $\mathbb{Z}$, respectively. Figure 1 partitions the signature Σ_{AS} into the subsignatures Σ_{A} and Σ_{S}, as indicated. The first includes the usual symbols of linear integer arithmetic, interpreted as expected. We will write $t_1 \bowtie t_2$, with $\bowtie \in \{>, <, \leqslant\}$, as syntactic sugar for the equivalent inequality between t_1 and t_2 expressed using only $\geqslant$. The subsignature Σ_{S} includes: all the words of $\mathcal{A}^*$ (including the empty word ϵ) as constant symbols, or *string constants*, each interpreted as itself; a function symbol $\mathsf{len} : \mathsf{Str} \rightarrow \mathsf{Int}$, interpreted as the word length function; and a code point function whose semantics is defined as follows.

Definition 1. *Given alphabet $\mathcal{A}$ and its associated total order $<$, let $(c_0, \ldots, c_{n-1})$ be the enumeration of $\mathcal{A}$ induced by $<$ (with $c_i < c_{i+1}$ for all $i = 0 \ldots, n-2$). For each character c_i in the enumeration, we refer to i as its code point. The function symbol* $\mathsf{code} : \mathsf{Str} \rightarrow \mathsf{Int}$ *is interpreted in T_{AS} as the unique code point function* code *such that:*

1. *for all words $w \in \mathcal{A}^1$,* $\mathrm{code}(w)$ *is the code point of the (single) character of w, and*
2. *for all other words $w \in \mathcal{A}^*$,* $\mathrm{code}(w)$ *is -1.*

The code point function can be used in practice to reason about the *code point* values of Unicode strings.[1] We will see in Sect. 4 that this operator is very useful for encoding constraints that occur in applications. We stress, however, that the procedure presented in this paper is agnostic with respect to the concrete alphabet $\mathcal{A}$ and its character ordering.

Note that we do *not* consider string concatenation in the signature above. This omission is for the sake of modularity; also, procedures for word equations have been addressed in a number of recent works [4,21]. In practice, our procedure for string constraints involving code can be naturally combined with existing procedures for a signature that includes string concatenation, as we discuss in Sect. 3.1.

An *atomic term* is either a constant or a variable. A *string term* is either a constant or one that contains function symbols from Σ_{S} only. Notice that integer constants are string terms. A *string constraint* is a (dis)equality between string terms. An *arithmetic constraint* is an inequality or (dis)equality between linear combinations of atomic and/or string terms with integer sort. Notice that the equality $\mathsf{code}(x) \approx \mathsf{code}(y)$ with variables x and y is both a string constraint and an arithmetic constraint.

3 A Decision Procedure for String to Code Point Conversion

In this section, we introduce a decision procedure for a fragment of string constraints involving code but not containing string concatenation. In particular, we introduce a decision procedure for sets of (quantifier-free) Σ_{AS}-constraints

[1] For technical details on Unicode see [28].

for the signature introduced in Fig. 1. A key property of this procedure is that it is able to reason about terms of the form $\mathsf{code}(x)$ without having to do case splitting on concrete values for string x.

Following Liang et al. [21], we describe this procedure as a set of derivation rules that modify configurations of the form $\langle \mathsf{A}, \mathsf{S} \rangle$, where A is a set of arithmetic constraints, and S is a set of string constraints. At a high level, the procedure can be understood as a cooperation between two subsolvers, an *arithmetic subsolver* and a *string subsolver*, which handle these two sets respectively. Our procedure assumes the following preconditions on $\langle \mathsf{A}, \mathsf{S} \rangle$ and maintains them as an invariant for all derived configurations:

1. $\mathsf{A} \cup \mathsf{S}$ contains no terms of the form $\mathsf{len}(l)$ or $\mathsf{code}(l)$ for any string literal l.
2. For every string literal $l \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})$, the set S contains $x \approx l$ for some variable x.

The above restrictions come with no loss of generality since terms of the form $\mathsf{len}(l)$ and $\mathsf{code}(l)$ can be replaced by an equivalent (constant) integer, and fresh variables can be introduced as necessary for the second requirement.

We present the rules of the procedure in two parts, given in Figs. 2 and 3. The rules are given in *guarded assignment form*, where the top of the rule describes the conditions under which the rule can be applied, and the bottom of the rule either is unsat, or otherwise describes the resulting modifications to the components of our configuration. A rule may have multiple, alternative conclusions separated by $\|$. In the premises of the rules, we write $\mathsf{S} \models \varphi$ to denote that S entails formula φ in the empty theory. This can be checked using a standard algorithm for congruence closure, where string literals are treated as distinct values; thus $\mathsf{S} \models l_1 \not\approx l_2$ for any S, $l_1 \neq l_2$. Observe that, for $f \in \{\mathsf{len}, \mathsf{code}\}$, $\mathsf{S} \models f(x) \approx f(y)$ iff $\mathsf{S} \models x \approx y$.

An application of a rule is *redundant* if it has a conclusion where each component in the derived configuration is a subset of the corresponding component in the premise configuration. A configuration other than unsat is *saturated* if every possible application of a derivation rule to it is redundant. A *derivation tree* is a tree where each node is a configuration whose children, if any, are obtained by a non-redundant application of a rule of the calculus. A derivation tree is *closed* if all of its leaves are unsat. We show later that a closed derivation tree with root node $\langle \mathsf{A}, \mathsf{S} \rangle$ is a proof that $\mathsf{A} \cup \mathsf{S}$ is unsatisfiable in T_{AS}. In contrast, a derivation tree with root node $\langle \mathsf{A}, \mathsf{S} \rangle$ and a saturated leaf is a witness that $\mathsf{A} \cup \mathsf{S}$ is satisfiable in T_{AS}.

Figure 2 presents rules adapted from previous work [21,26] that model the interaction between the string and arithmetic subsolvers. First, either subsolver can report that the current set of constraints is unsatisfiable by the rules A-Conf or S-Conf. For the former, the entailment $\models_{\mathsf{LIA}}$ can be checked by a standard procedure for linear integer arithmetic. The rules A-Prop and S-Prop correspond to a form of Nelson-Oppen-style theory combination between the two subsolvers. In particular, each theory solver propagates entailed equalities between terms of type Int. The next two rules ensure that length constraints are satisfied. In particular, L-Intro ensures that the length of a term x is equal to the length of

$$\text{A-Conf}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\mathsf{unsat}} \qquad \text{A-Prop}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} s \approx t \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{S} := \mathsf{S}, s \approx t}$$

$$\text{S-Conf}\ \frac{\mathsf{S} \models \bot}{\mathsf{unsat}} \qquad \text{S-Prop}\ \frac{\mathsf{S} \models s \approx t \quad s,t : \mathsf{Int} \quad s,t \in \mathcal{T}(\mathsf{A} \cup \mathsf{S})}{\mathsf{A} := \mathsf{A}, s \approx t}$$

$$\text{L-Intro}\ \frac{\mathsf{S} \models x \approx l \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S},\ \mathsf{len}(x) \approx (\mathsf{len}(l))\downarrow} \qquad \text{L-Valid}\ \frac{x \in \mathcal{T}(\mathsf{A} \cup \mathsf{S}) \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \quad \mathsf{A} := \mathsf{A},\ \mathsf{len}(x) > 0}$$

$$\text{Card}\ \frac{\mathsf{S} \models \mathsf{len}(x_1) \approx \ldots \approx \mathsf{len}(x_n) \quad n > 1}{\|_{1 \leqslant i < j \leqslant n}\ \mathsf{S} := \mathsf{S}, x_i \approx x_j \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(x_1) > \lfloor \log_{|\mathcal{A}|}(n-1) \rfloor}$$

Fig. 2. Core derivation rules.

$$\text{C-Intro}\ \frac{\mathsf{S} \models x \approx l \quad l \in \mathcal{A}^1}{\mathsf{A} := \mathsf{A}, \mathsf{code}(x) \approx (\mathsf{code}(l))\downarrow} \qquad \text{C-Collapse}\ \frac{\mathsf{S} \models x \approx l \quad \mathsf{code}(x) \in \mathcal{T}(\mathsf{S})}{\mathsf{A} := \mathsf{A}, \mathsf{code}(x) \approx (\mathsf{code}(l))\downarrow}$$

$$\text{C-Valid}\ \frac{\mathsf{code}(x) \in \mathcal{T}(\mathsf{S})}{\mathsf{A} := \mathsf{A}, \mathsf{len}(x) \not\approx 1, \mathsf{code}(x) \approx -1 \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}(x) \approx 1, 0 \leqslant \mathsf{code}(x) < |\mathcal{A}|}$$

$$\text{C-Inj}\ \frac{\mathsf{code}(x), \mathsf{code}(y) \in \mathcal{T}(\mathsf{S}) \quad x, y \text{ distinct}}{\mathsf{A} := \mathsf{A}, \mathsf{code}(x) \approx -1 \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{code}(x) \not\approx \mathsf{code}(y) \quad \| \quad \mathsf{S} := \mathsf{S}, x \approx y}$$

Fig. 3. Code point derivation rules.

string literals x is equated to in S. We write $(\mathsf{len}(l))\downarrow$ to denote the constant integer corresponding to the result of evaluating the expression $\mathsf{len}(l)$. The rule L-Valid has two conclusions. It ensures that either x is the empty string or the value assigned to $\mathsf{len}(x)$ is positive. Finally, since our alphabet is finite, the rule Card is used to determine when a length constraint is implied due to the number of distinct terms of a given length. In particular, if there are n distinct variables $x_1, \ldots, x_n$ whose length is the same, then either x_i is equal to x_j for some $i \neq j$, or their length must be large enough so that they each can be assigned a unique string value. The lower bound on their length is determined by taking the floor of the logarithm of $n-1$ base the cardinality of the alphabet, where this expression denotes an integer constant.[2]

Figure 3 lists rules for reasoning about the code point function. In C-Intro, if a string variable x is equal to a string literal of length one, we add to S an equality between $\mathsf{code}(x)$ and the concrete value of the code point of l. An equality of this form is also added via the rule C-Collapse if a string term x is equated in S to a string literal and, in addition, $\mathsf{code}(x)$ occurs in S. Rule C-Valid splits on whether an instance of $\mathsf{code}(x)$ from S is equal to a valid code point. The left conclusion considers the case where the code point is -1, which means that x must have a length different from 1. The right conclusion considers the case where the code point is between 0 and $|\mathcal{A}| - 1$, meaning that x is a one-character string. Finally, rule C-Inj reflects the fact that code denotes an injective function over the domain of strings of length 1. More precisely, it captures the fact that

[2] In the degenerate case where the cardinality of the alphabet is one, we assume this branch is omitted from the conclusion since logarithm base one is undefined.

for any pair of string values l_x and l_y for x and y respectively, one of the following (non-necessarily disjoint) cases always holds: (i) l_x has a length different from 1, (ii) l_x has length 1 and differs from l_y, and has a different code point from that of l_y, or (iii) l_x and l_y are the same.

We now demonstrate the procedure with a few simple examples. Recall that we assume a fixed alphabet $\mathcal{A}$ and write c_n to denote a character from this alphabet whose code point is some n between 0 and the cardinality of $\mathcal{A}$ minus one.

Example 1. Let A_0 be $\{\mathsf{len}(x) > \mathsf{len}(y),\ \mathsf{code}(x) \approx \mathsf{code}(y),\ \mathsf{code}(x) \geqslant 0\}$ and let S_0 be $\varnothing$. We can generate the following closed derivation tree with root $\langle \mathsf{A}_0, \mathsf{S}_0 \rangle$. At each node, we list the new constraint that is added to the configuration at that node. All the leaf nodes are derived by A-Conf (not shown in the tree).

$$\dfrac{\dfrac{\langle \mathsf{A}, \mathsf{S}\rangle := \langle \mathsf{A}_0, \mathsf{S}_0\rangle}{\mathsf{code}(x) \approx \mathsf{code}(y) \in \mathsf{S}}\ \textsf{A-Prop}}{\dfrac{\mathsf{code}(x) \approx -1 \in \mathsf{A}}{\mathsf{unsat}} \quad \dfrac{\mathsf{code}(x) \not\approx \mathsf{code}(y) \in \mathsf{A}}{\mathsf{unsat}} \quad \dfrac{\dfrac{x \approx y \in \mathsf{S}}{\mathsf{len}(x) \approx \mathsf{len}(y) \in \mathsf{A}}\ \textsf{S-Prop}}{\mathsf{unsat}}}\ \textsf{C-Inj}$$

First, since $\mathsf{A} \models_{\mathsf{LIA}} \mathsf{code}(x) \approx \mathsf{code}(y)$, we apply A-Prop which adds the equality $\mathsf{code}(x) \approx \mathsf{code}(y)$ to S. Subsequently, since $\mathsf{code}(x), \mathsf{code}(y) \in \mathcal{T}(\mathsf{S})$, we apply C-Inj which considers three cases. The first two branches result in the arithmetic component of our configuration A being unsatisfiable, and thus unsat may be derived by A-Conf. In the third branch, we consider the case where x is equal to y. We have that S entails that $\mathsf{len}(x) \approx \mathsf{len}(y)$, and hence, by S-Prop, this equality is added to A. Since $\mathsf{len}(x) > \mathsf{len}(y)$ is already in A, we can derive unsat in this branch by A-Conf as well. Since there is a closed derivation tree with root $\langle \mathsf{A}_0, \mathsf{S}_0 \rangle$, $\mathsf{A}_0 \cup \mathsf{S}_0$ is unsatisfiable in T_{AS}. □

Example 2. Let A_0 be $\{97 \leqslant \mathsf{code}(x) \leqslant 106\}$ and let S_0 be $\{x \not\approx y,\ x \not\approx z,\ y \approx c_{97},\ z \approx c_{106}\}$. We may obtain a derivation tree with root $\langle \mathsf{A}_0, \mathsf{S}_0 \rangle$ and a saturated configuration $\langle \mathsf{A}, \mathsf{S} \rangle$ where A extends A_0 with the constraints:

$$\{\mathsf{code}(x) \not\approx \mathsf{code}(y),\ \mathsf{code}(x) \not\approx \mathsf{code}(z),\ \mathsf{code}(y) \approx 97,\ \mathsf{code}(z) \approx 106\}$$

The constraints $\mathsf{code}(x) \not\approx \mathsf{code}(y)$ and $\mathsf{code}(x) \not\approx \mathsf{code}(z)$ may be obtained by C-Inj, and $\mathsf{code}(y) \approx 97$ and $\mathsf{code}(z) \approx 106$ may be obtained by C-Intro. Since a saturated configuration exists in a derivation tree with root node $\langle \mathsf{A}_0, \mathsf{S}_0 \rangle$, we have that $\mathsf{A}_0 \cup \mathsf{S}_0$ is satisfiable in T_{AS}. As we show in Theorem 1 (below), a model for $\mathsf{A}_0 \cup \mathsf{S}_0$ can be obtained by constructing an arbitrary model for $\mathsf{A} \cup \mathsf{S}$. In particular, notice that due to our derived constraints, it must be the case that $\mathsf{code}(x)$ is assigned a value in the range $[98 \ldots 105]$. Indeed, a model $\mathcal{M}$ exists for $\mathsf{A}_0 \cup \mathsf{S}_0$, where $\mathcal{M}(x) = c_k, \mathcal{M}(y) = c_{97}$, and $\mathcal{M}(z) = c_{106}$, for any k in the range $[98 \ldots 105]$. Note that we do not explicitly case split on the value of x. Instead, as we later describe in Definition 2, our procedure assigns a value to x based on the value that the arithmetic subsolver gives to $\mathsf{code}(x)$. □

Example 3. Let A_0 be $\{48 \leqslant \mathsf{code}(x) < 58, \mathsf{len}(x) < 1\}$ and let S_0 be $\varnothing$. We may obtain the following closed derivation tree with root $\langle \mathsf{A}_0, \mathsf{S}_0 \rangle$.

$$\dfrac{\langle \mathsf{A}, \mathsf{S} \rangle := \langle \mathsf{A}_0, \mathsf{S}_0 \rangle}{\dfrac{x \approx \epsilon \in \mathsf{S}}{\dfrac{\mathsf{code}(x) \approx -1 \in \mathsf{A}}{\mathsf{unsat}}\ \textsf{A-Conf}}\ \textsf{C-Collapse} \quad \dfrac{\mathsf{len}(x) > 0 \in \mathsf{A}}{\mathsf{unsat}}\ \textsf{A-Conf}}\ \textsf{L-Valid}$$

Since x is a string term from $\mathcal{T}(\mathsf{A} \cup \mathsf{S})$, we apply L-Valid. The left branch considers the case where x is empty. Since $\mathsf{S} \models x \approx \epsilon$, we apply C-Collapse which adds $\mathsf{code}(x) \approx \mathsf{code}(\epsilon)\downarrow = -1$ to A. This makes A unsatisfiable, and we can derive unsat by A-Conf. In the right branch, we consider the case that $\mathsf{len}(x) > 0$, which results in a case where A is unsatisfiable since $\mathsf{len}(x) < 1 \in \mathsf{A}$. Thus, $\mathsf{A}_0 \cup \mathsf{S}_0$ is unsatisfiable in T_{AS}. □

Example 4. Let $\mathsf{A}_0 = \{0 \leqslant \mathsf{code}(x) < \mathsf{len}(x)\}$ and $\mathsf{S}_0 = \varnothing$. We may obtain a saturated configuration $\langle \mathsf{A}, \mathsf{S} \rangle$ where A extends A_0 with $\{\mathsf{len}(x) \approx 1, 0 \leqslant \mathsf{code}(x) < |\mathcal{A}|\}$. These constraints are obtained by considering the right branch of an application of C-Valid since $\mathsf{code}(x) \in \mathcal{T}(\mathsf{S})$ (after the trivial propagation $\mathsf{code}(x) \approx \mathsf{code}(x)$ by A-Prop). The only models for $\mathsf{A} \cup \mathsf{S}$ are those where $\mathsf{code}(x)$ is assigned the value for 0; hence the only models $\mathcal{M}$ for $\mathsf{A} \cup \mathsf{S}$ (and hence $\mathsf{A}_0 \cup \mathsf{S}_0$) are where $\mathcal{M}(x) = c_0$. □

We now discuss the formal properties of our calculus, proving that it is refutation-sound, model-sound, and terminating for any set of Σ_{AS}-constraints, and thus yields a decision procedure. We also show that, for any saturated configuration, it is possible to construct a model for the input constraints based on the procedure given in the following definition. In each step, we argue the well-formedness of this construction. In the subsequent theorem, we show that the constructed model indeed satisfies our input constraints.

Definition 2 (Model Construction). *Let $\langle \mathsf{A}, \mathsf{S} \rangle$ be a saturated configuration. Construct a model $\mathcal{M}$ for $\mathsf{A} \cup \mathsf{S}$ based on the following steps.*

1. *Let U be the set of terms of the form $\mathsf{len}(x)$ or $\mathsf{code}(x)$ that occur in A. Let $\mathcal{Z}$ be a model of A', where A' is the result of replacing in A each of its subterms $t \in U$ with a fresh integer variable u_t. Notice that $\mathcal{Z}$ exists, since A-Conf does not apply to our configuration, meaning that A (and hence A') is satisfiable in LIA.*
2. *Construct $\mathcal{M}$ by assigning values to the variables in $\mathsf{A} \cup \mathsf{S}$ in the following order. Below, let $\widehat{\mathsf{S}}$ denote the congruence closure of S.*[3]
   - *(a) For all integer variables x, set $\mathcal{M}(x) = \mathcal{Z}(x)$.*
   - *(b) For all string equivalence classes $e \in \widehat{\mathsf{S}}$ that contain a string constant l (including the case where $l = \epsilon$), set $\mathcal{M}(y) = l$ for all variables $y \in e$. Notice that l is unique since S-Conf does not apply to our configuration.*

[3] That is, the equivalence relation over $\mathcal{T}(\mathsf{S})$ such that s, t are in the same equivalence class if and only if $\mathsf{S} \models s \approx t$.

(c) *For all string equivalence classes* $e \in \widehat{\mathsf{S}}$*, such that* $\mathcal{Z}(u_{\mathsf{len}(z)}) = 1$ *and* $\mathsf{code}(z) \in \mathcal{T}(\mathsf{S})$ *for some* $z \in e$*, we let* $\mathcal{M}(y) = c_k$ *for each variable* $y \in e$*, where* $k = \mathcal{Z}(u_{\mathsf{code}(z)})$*. Since* C-Valid *cannot be applied to our configuration, it must be the case that* A' *contains the constraint* $0 \leqslant u_{\mathsf{code}(z)} < |\mathcal{A}|$*. Since* $\mathcal{Z}$ *satisfies* A'*, the value of* $\mathcal{Z}(u_{\mathsf{code}(z)})$ *is guaranteed to be a valid code point and thus* c_k *is indeed a character in* $\mathcal{A}$*.*

(d) *For all remaining unassigned string equivalence classes* $e \in \widehat{\mathsf{S}}$*, we have that* $\mathsf{len}(z) \in \mathcal{T}(\mathsf{A})$ *for all variables* $z \in e$*, since* L-Valid *cannot be applied to our configuration. We choose some* l *of length* $\mathcal{Z}(u_{\mathsf{len}(z)})$*, such that* l *is not already assigned to any other string variable in* $\mathcal{M}$*, and set* $\mathcal{M}(y) = l$ *for all variables* $y \in e$*. Since our configuration is saturated with respect to* Card*, we know that at least one such string literal exists: if the set of string literals of length* $\mathcal{Z}(u_{\mathsf{len}(z)})$ *were each in the range of* $\mathcal{M}$*, it would imply that there are* $|\mathcal{A}|^{\mathcal{Z}(u_{\mathsf{len}(z)})} + 1$ *distinct terms whose length is* $\mathcal{Z}(u_{\mathsf{len}(z)})$*, in which case* Card *would require* $\mathsf{len}(z)$ *to be greater than the value of* $\lfloor \log_{|\mathcal{A}|}(|\mathcal{A}|^{\mathcal{Z}(u_{\mathsf{len}(z)})} + 1 - 1) \rfloor = \mathcal{Z}(u_{\mathsf{len}(z)})$*. However, this is not the case since* A *is satisfiable in* LIA*.*

Theorem 1. *Let* $\mathsf{M} = \mathsf{A}_0 \cup \mathsf{S}_0$ *be a set of* Σ_{AS}*-constraints where* A_0 *are arithmetic constraints and* S_0 *are non-arithmetic constraints. The following statements hold.*

1. *There is a closed derivation tree with root* $\langle \mathsf{A}_0, \mathsf{S}_0 \rangle$ *only if* M *is unsatisfiable in* T_{AS}*.*
2. *There is a derivation tree with root* $\langle \mathsf{A}_0, \mathsf{S}_0 \rangle$ *containing a saturated configuration only if* M *is satisfiable in* T_{AS}*.*
3. *All derivation trees with root* $\langle \mathsf{A}_0, \mathsf{S}_0 \rangle$ *are finite.*

Proof. To show (1), assume there exists a model $\mathcal{M}$ of $\mathsf{A}_0 \cup \mathsf{S}_0$. It is straightforward to show that for every rule of the calculus, applying that rule to any node $\langle \mathsf{A}, \mathsf{S} \rangle$ results in a tree where at least one child $\langle \mathsf{A}', \mathsf{S}' \rangle$ is such that $\mathcal{M}$ also satisfies $\mathsf{A}' \cup \mathsf{S}'$. Thus, by induction on the size of the derivation tree, there exists at least one terminal node that is not closed. Thus, if there exists a closed derivation tree with root node $\langle \mathsf{A}_0, \mathsf{S}_0 \rangle$, then it must be the case that no model exists for $\mathsf{A}_0 \cup \mathsf{S}_0$, so M is unsatisfiable in T_{AS}.

To show (2), assume there exists a derivation tree with a saturated configuration $\langle \mathsf{A}, \mathsf{S} \rangle$. Let $\mathcal{M}$ be the model constructed based on the procedure in Definition 2. Below, we argue that $\mathcal{M}$ is a model for $\mathsf{A} \cup \mathsf{S}$, which is a superset of $\mathsf{A}_0 \cup \mathsf{S}_0$ and thus satisfies M. Let U and $\mathcal{Z}$ respectively be the set of terms and model as computed in Step 1 of Definition 2. Below, we show that $\mathcal{M}$ is a model for each constraint in $\mathsf{A} \cup \mathsf{S}$.

- To show $\mathcal{M}$ satisfies each constraint in A, we show $\mathcal{Z}(x) = \mathcal{M}(x \cdot \sigma)$ for all integer variables x, where σ is the substitution $\{u_t \mapsto t \mid u_t \in U\}$.
  - Consider the case $x = u_{\mathsf{len}(y)}$ for some y, that is, x is a variable introduced in Step 1 of Definition 2 for an application of a length term. If y was assigned a value in Step 2(b) of Definition 2, then $\mathcal{M}(y) = l$ for some l

such that $\mathsf{S} \models y \approx l$. Since L-Intro cannot be applied to our configuration, we have that $\mathsf{len}(y) \approx (\mathsf{len}(l))\downarrow \in \mathsf{A}$, and hence $\mathcal{Z}(u_{\mathsf{len}(y)}) = \mathcal{Z}(\mathsf{len}(l)\downarrow) = \mathcal{M}(\mathsf{len}(y))$. If y was assigned in Step 2(c) or 2(d), we have that $\mathcal{M}(y) = l$ for some l whose length is $\mathcal{Z}(u_{\mathsf{len}(y)})$, and hence $\mathcal{Z}(u_{\mathsf{len}(y)}) = \mathcal{M}(\mathsf{len}(y))$.

- Consider the case $x = u_{\mathsf{code}(y)}$ for some y. If y was assigned in Step 2(b) of Definition 2, then $\mathcal{M}(y) = l$ for some l such that $\mathsf{S} \models y \approx l$. Since C-Collapse cannot be applied to our configuration, we have that $\mathsf{code}(y) \approx (\mathsf{code}(l))\downarrow \in \mathsf{A}$ and hence $\mathcal{Z}(u_{\mathsf{code}(y)}) = \mathcal{M}(\mathsf{code}(l)\downarrow) = \mathcal{M}(\mathsf{code}(y))$. If y was assigned in Step 2(c) or 2(d), we have that $\mathcal{M}(y) = l$ for some l whose length is $\mathcal{Z}(u_{\mathsf{len}(y)})$. If it was assigned in Step 2(c), we have that $l = c_k$ for $k = \mathcal{Z}(u_{\mathsf{code}(y)})$ and hence $\mathcal{Z}(u_{\mathsf{code}(y)}) = \mathcal{M}(\mathsf{code}(y))$. If y was assigned in Step 2(d), we have that $\mathsf{code}(y) \in \mathcal{T}(\mathsf{S})$. Since y was not assigned in Step 2(c), it must be the case that $\mathcal{Z}(u_{\mathsf{len}(y)}) \neq 1$. Since C-Valid cannot be applied, and since $\mathsf{code}(y) \in \mathcal{T}(\mathsf{S})$, we have, by its left conclusion, that $\mathsf{len}(y) \not\approx 1$ and $\mathsf{code}(y) \approx -1$ are in A. Due to the former constraint, $\mathcal{Z}(u_{\mathsf{len}(y)}) \neq 1$ and the length of l is not one, and thus $\mathcal{M}(\mathsf{code}(y)) = -1$. Due to the latter constraint, $\mathcal{Z}(u_{\mathsf{code}(y)}) = -1$ as well.
- For all other $x : \mathsf{Int}$, we have that $\mathcal{Z}(x) = \mathcal{M}(x)$ by Step 2(a) of Definition 2.

In all cases above, we have shown that $\mathcal{Z}(x) = \mathcal{M}(t)$ where $t = x \cdot \sigma$. Since all free variables of A' are of integer type and since $\mathcal{Z}$ is a model for A', we have that $\mathcal{M}$ satisfies $\mathsf{A}' \cdot \sigma = \mathsf{A}$.

– To show $\mathcal{M}$ satisfies the equalities between terms of type Int in S, since our configuration is saturated with respect to S-Prop, equalities between integer terms are a subset of those in A, and since $\mathcal{M}$ satisfies A, it satisfies these equalities as well. Furthermore, S' does not contain disequalities between terms of type Int by construction.

– To show $\mathcal{M}$ satisfies the equalities between terms of type Str in S, notice that $s \approx t \in \mathsf{S}'$ implies that s and t reside in the same equivalence class of $e \in \widehat{\mathsf{S}}$. By construction of $\mathcal{M}$ every variable in e is assigned the same value and that value is the same value as the string literal in e if one exists. Thus $\mathcal{M}(s) = \mathcal{M}(t)$ for all terms s, t of type Str that reside in the same equivalence class, and thus $\mathcal{M}$ satisfies $s \approx t$.

– To show that $\mathcal{M}$ satisfies the disequalities $s \not\approx t$ between terms of string type in S, it suffices to show that distinct values are assigned to variables in each distinct equivalence class of $\widehat{\mathsf{S}}$. Moreover, by assumption of the configurations, each equivalence class of terms of type string has at least one variable in it. Let x and y be variables residing in two distinct equivalence classes of $\widehat{\mathsf{S}}$, and without loss of generality, assume y was assigned after x in the construction of $\mathcal{M}$. We show $\mathcal{M}(x) \neq \mathcal{M}(y)$ in the following. If y was assigned in Step 2(d) of Definition 2, then the statement holds since by construction, its value was chosen to be distinct from the value of string variables in previous equivalence classes, including the one containing x. If both x and y were assigned in Step 2(b), the statement holds since S-Conf does not apply. Otherwise, y must have been assigned in Step 2(c) to a string literal of length one. If x was assigned in Step 2(b) and $\mathsf{S} \models x \approx l$ for some string literal l not of length one, then x

and y are assigned different values trivially. Otherwise, x is assigned (either by Step 2(b) or Step 2(c)) to a string of length one. Moreover, $\mathsf{code}(z)$ is a term in S for some z such that $\mathsf{S} \models x \approx z$: if x was assigned in Step 2(b), since C-Intro cannot be applied we have $\mathsf{code}(x) \approx \mathsf{code}(l)\!\downarrow\, \in \mathsf{S}$; if x was assigned in Step 2(c) it holds by construction. Since C-Inj cannot be applied, either $\mathsf{code}(y) \approx -1 \in \mathsf{A}$, $\mathsf{code}(z) \not\approx \mathsf{code}(y) \in \mathsf{A}$, or $z \approx y \in \mathsf{S}$. The first case cannot hold since $\mathcal{M}$ satisfies A, and thus $\mathcal{M}(\mathsf{code}(y))$ is not equal to -1. In the second case, since $\mathcal{M}$ satisfies A, we have that $\mathcal{M}(\mathsf{code}(z)) \neq \mathcal{M}(\mathsf{code}(y))$, and hence, since code is injective over the domain of strings of length one, we have that $\mathcal{M}(z) \neq \mathcal{M}(y)$. Since $\mathcal{M}(z) = \mathcal{M}(x)$, it then follows that $\mathcal{M}(x) \neq \mathcal{M}(y)$. The third case cannot hold since z and y are in distinct equivalence classes. Thus, variables in distinct equivalence classes are assigned distinct values. All disequalities $s \not\approx t \in \mathsf{S}$ are such that s and t are in different equivalence classes since S-Conf cannot be applied. Thus, $\mathcal{M}$ satisfies $s \not\approx t$.

Thus, $\mathcal{M}$ satisfies all constraints in $\mathsf{A} \cup \mathsf{S}$ and the part (2) of the theorem holds.

To show (3), it is enough to show that only finitely many constraints can be generated by the rules of the calculus. Let T^* be the (finite) set of terms that includes $\mathcal{T}(\mathsf{A}_0 \cup \mathsf{S}_0) \cup \{\epsilon, -1\}$ and contains $\mathsf{len}(x)$ and $\mathsf{code}(x)$ for all variables $x \in \mathcal{T}(\mathsf{A}_0 \cup \mathsf{S}_0)$ of type Str, and $(\mathsf{len}(l))\!\downarrow$ and $(\mathsf{code}(l))\!\downarrow$ for all string literals $l \in \mathcal{T}(\mathsf{A}_0 \cup \mathsf{S}_0)$. Let A^* be the set containing A_0, equalities between terms from T^* of type Int, literals of the form $\mathsf{len}(x) > 0, 0 \leqslant \mathsf{code}(x) < |\mathcal{A}|$, $\mathsf{len}(x) \not\approx 1$ for all variables $x \in \mathcal{T}(\mathsf{A}_0 \cup \mathsf{S}_0)$ of type Str, and inequalities of the form $\mathsf{len}(x) > \log_{|\mathcal{A}|}(n-1)$ where n is any positive integer less than or equal to the number of terms of type Str in $\mathcal{T}(\mathsf{A}_0 \cup \mathsf{S}_0)$. Let S^* be the set containing S_0 and equalities between string terms from T^*. Notice that both A^* and S^* are finite. By definition of the rules of our calculus, and by induction on the size of the derivation tree, one can show that all derived configurations $\langle \mathsf{A}, \mathsf{S} \rangle$ are such that $\mathsf{A} \cup \mathsf{S}$ is a subset of $\mathsf{A}^* \cup \mathsf{S}^*$. Since no application of a derivation rule in a tree is redundant, each node in the derivation tree contains at least one more constraint from this set than its parent. Thus, the depth of any tree is bounded by the cardinality of $\mathsf{A}^* \cup \mathsf{S}^*$, and the statement holds. □

An immediate consequence of Theorem 1 is that any strategy for applying the derivation rules in Figs. 2 and 3 is a decision procedure for Σ_{AS}-constraints. We stress that, thanks to the constructiveness of the proof of Part 2, the procedure can also compute a satisfying assignment for the free variables of M when it halts with a saturated configuration.

3.1 Implementation in an SMT Solver

The procedure in this section can be integrated into the DPLL(T) solving architecture [24] used by modern SMT solvers such as CVC4. In the most basic version of this architecture, given an arbitrary quantifier-free Σ_{AS}-formula, an incremental propositional SAT solver first searches for a truth assignment for the literals of this formula that satisfies the formula at the propositional level. If none can

substr :	Str × Int × Int → Str	to_int :	Str → Int	to_lower :	Str → Str
≤:	Str × Str → Bool	from_int :	Int → Str	to_upper :	Str → Str

Fig. 4. A sample of the extended string functions.

be found, the input is declared unsatisfiable. Otherwise, the found assignment is given as a set of Σ_{AS}-literals to a theory solver that implements the calculus above. If the solver finds a saturated configuration, then the input is declared satisfiable. Otherwise, either a *conflict clause* or a *lemma* is asserted to the SAT solver in the form of additional T_{AS}-valid constraints and the process restarts with the extended set of formulas.

We have integrated the procedure in cvc4. cvc4's linear arithmetic subsolver acts as the arithmetic subsolver of our procedure and reports a conflict clause when the rule A-Conf is applied. Similarly, the string subsolver reports conflict clauses when S-Conf is applied. The rules A-Prop and S-Prop are implemented using the standard Nelson-Oppen theory combination mechanism. Rules with multiple conclusions are implemented via the splitting-on-demand paradigm [10], where the conclusions of the rule are sent as a disjunctive lemma to the SAT solver. The remaining rules are implemented using a solver whose core data structure implements congruence closure, where additional (dis)equalities are added to this structure based on the specific rules of the calculus.

We remark that the procedure presented in this section can be naturally combined with procedures for other kinds of string constraints. While the rules we presented had premises of the form $\mathsf{S} \models s \approx t$ denoting entailment in the empty theory, the procedure can be applied in the same manner for premises $\mathsf{S} \models_{T_S} s \approx t$ for any extension T_S of the core theory of strings. In practice, our theory solver interleaves reasoning about code points with reasoning about other string operators, e.g., string concatenation and regular expressions operators, via the procedure by Liang et al. [21].

The derivation rules of the calculus are applied with consideration to combinations with the other subsolvers of cvc4. For the rules in Fig. 2, we follow the strategy used by Liang et al., which applies L-Intro and L-Valid eagerly and Card only after a configuration is saturated with respect to all other rules. Moreover, since Card is very expensive, we split on equalities between string terms ($x_1, \ldots, x_n$ in the premise of this rule) if some x_i, x_j such that neither $x_i \approx x_j$ or $x_i \not\approx x_j$ is in our current set of assertions. Among the rules in Fig. 3, C-Valid and C-Collapse are applied eagerly, the former when a term $\mathsf{code}(x)$ is registered with the string subsolver, and the latter as soon as our congruence closure procedure puts that term in the same equivalence class as a string literal. Rules C-Intro and C-Inj are applied lazily, only after the arithmetic subsolver determines A is satisfiable in LIA and the string subsolver is finished computing the set of equalities that are entailed by S.

4 Applications

In this section, we describe how a number of common string functions can be implemented efficiently using reductions involving the code function. Previous work has focused on efficient techniques for handling *extended* string functions, which include operators like substring (substr) and string replace (replace), among others [26]. Here we consider the alphabet $\mathcal{A}$ to be the set of all Unicode characters and interpret code as mapping one-character strings to the character's Unicode code point.

A few commonly used extended functions are listed in Fig. 4. In the following, we say a string l is *numeric* if it is non-empty, all of its characters are in the range "0" ... "9", and it its has no leading zeroes, that is, it starts with "0" only if it has length 1.[4] At a high level, the semantics of the operators in Fig. 4 is the following. First, $\mathsf{substr}(x, n, m)$ is interpreted as the maximal substring of x starting at position n with length at most m, or the empty string if n is outside the interval $[0, |x| - 1]$ or m is negative; $\mathsf{to\_int}(x)$ is the non-negative integer represented by x in decimal notation if x is numeric, and is -1 otherwise; $\mathsf{from\_int}(n)$ is the result of converting the value of n to its decimal notation if n is non-negative, and is ϵ otherwise; $x \preceq y$ holds if x is equal to y or precedes it lexicographically, that is, in the lexicographic extension of the character ordering $<$ introduced in Definition 1; $\mathsf{to\_upper}(x)$ maps each lower case letter character from the Basic Latin Unicode block (code points 97 to 122) in x to its uppercase version and all the other characters to themselves. The inverse function $\mathsf{to\_lower}(x)$ is similar except that it maps upper case letters (code points 65 to 90) to their lower case version.

Note that our restriction to the Latin alphabet $\mathsf{to\_upper}(x)$ and to_lower is only for simplicity since case conversions for the entire Unicode alphabet depend on the locale and follow complex rules. However, our definition and reduction can be extended as needed depending on the application.

Generally speaking, current string solvers handle the additional functions above using lazy reductions to a core language of string constraints. We say ρ is a *reduction predicate* for an extended function f if ρ does not contain f and is equivalent to $\lambda \boldsymbol{x}, y.\, f(\boldsymbol{x}) \approx y$ where $\boldsymbol{x}, y$ consist of distinct variables. All applications of f can be eliminated from a quantifier-free formula φ by replacing their occurrences with fresh variables y and conjoining φ with the appropriate applications of the reduction predicate. Reduction predicates are chosen so that their dependencies are not circular (for instance, we do not use reduction predicates for two functions that each introduce applications of the other). In practice, reduction predicates often may contain universally quantified formulas over (finite) integer ranges, which can be handled via a finite model finding strategy that incrementally sets upper bounds on the lengths of strings [26]. These reductions often generate constraints that are both large and hard to reason about. Further-

[4] Treatment of leading zeroes is slightly different in the SMT-LIB theory of strings [29]; our implementation actually conforms to the SMT-LIB semantics. Here, we provide an alternative semantics for simplicity since it admits a simpler reduction.

more, the reduction of certain extended functions cannot be expressed concisely. For example, a reduction for the $\mathsf{to\_upper}(s)$ function naively requires splitting on 26 cases to ensure that "a" is converted to "A", "b" to "B", and so on, for each character in s. As part of this work, we have revisited these reductions and incorporated the use of code. The new reduction predicates are more concise and lead to significant performance gains in practice as we demonstrate in Sect. 5.

Conversions to Lower/Upper Case. The equality $\mathsf{to\_lower}(s) \approx r$ is equivalent to:

$$\mathsf{len}(r) \approx \mathsf{len}(s) \wedge \forall_{0 \leqslant i < \mathsf{len}(s)}. \mathsf{code}(r_i) \approx \mathsf{code}(s_i) - \mathsf{ite}(97 \leqslant \mathsf{code}(s_i) \leqslant 122, 32, 0)$$

where r_i is $\mathsf{substr}(r, i, 1)$, s_i is $\mathsf{substr}(s, i, 1)$ and ite is the if-then-else operator. Intuitively, the formula above states that the result of $\mathsf{to\_upper}(s)$ is a string r of the same length as s such that for all positions i in s, the character at that position has a code point that is 32 less than the character at the same position in s if that character is a lowercase character; otherwise it has the same code point. Similarly, the equality $\mathsf{to\_lower}(s) \approx r$ is equivalent to:

$$\mathsf{len}(r) \approx \mathsf{len}(s) \wedge \forall_{0 \leqslant i < \mathsf{len}(s)}. \mathsf{code}(r_i) \approx \mathsf{code}(s_i) + \mathsf{ite}(65 \leqslant \mathsf{code}(s_i) \leqslant 90, 32, 0)$$

More generally, the code operator allows us to concisely encode many common string transducers, which have been studied in a number of recent works [6,19,22].

String to Integer Conversion. The equality $\mathsf{to\_int}(s) \approx r$ is equivalent to:

$$(\neg\varphi_s^{\mathsf{is\_num}} \Rightarrow r \approx -1) \wedge (\varphi_s^{\mathsf{is\_num}} \Rightarrow (r \approx \mathsf{sti}_s(\mathsf{len}(s)) \wedge \varphi_s^{\mathsf{sti}}))$$

where sti_s is an (uninterpreted) function of type $\mathsf{Int} \rightarrow \mathsf{Int}$, $\varphi_s^{\mathsf{is\_num}}$ is:

$$s \not\approx \epsilon \wedge \forall_{0 \leqslant i < \mathsf{len}(s)}. \mathsf{ite}(\mathsf{len}(s) > 1 \wedge i \approx 0, 49, 48) \leqslant \mathsf{code}(s_i) \leqslant 57$$

s_i is $\mathsf{substr}(s, i, 1)$, and φ_s^{sti} is:

$$\mathsf{sti}_s(0) \approx 0 \wedge \forall_{0 \leqslant i < \mathsf{len}(s)}. \mathsf{sti}_s(i+1) \approx 10 * \mathsf{sti}_s(i) + \mathsf{code}(s_i) - 48$$

In the above reduction, the formula $\varphi_s^{\mathsf{is\_num}}$ states that s is numeric. It must be non-empty, and each of its characters must have a code point in the interval $[48, 57]$, which corresponds to the characters for digits "0" through "9". The term $\mathsf{ite}(\mathsf{len}(s) > 1 \wedge i \approx 0, 49, 48)$ insists that the code point of the first index of s be at least 49 to exclude the possibility that its first character is "0" if the string has length greater than 1.

For a numeric string s, the formula φ_s^{sti} ensures that for each non-zero position i in s, the value of $\mathsf{sti}_s(i)$ is the result of converting the first i characters in s to the integer it denotes. The definition of φ_s^{sti} first constrains that $\mathsf{sti}_s(0)$ is zero. Then, for each $i \geqslant 0$, the value of $\mathsf{sti}_s(i+1)$ is determined by shifting the previously considered characters to the left by a digits place $(10 * \mathsf{sti}_s(i))$ and adding the integer interpretation of the current character $(\mathsf{code}(s_i) - 48)$. In the end, the above formula ensures that the value of $\mathsf{sti}_s(\mathsf{len}(s))$ is equivalent to the overall value of $\mathsf{to\_int}(s)$, which is constrained to be equal to the result r in the above reduction.

Given these definitions, it is straightforward to define the opposite reduction from integers to strings. The equality $\mathsf{from\_int}(n) \approx r$ is equivalent to the following:

$$(n < 0 \Rightarrow r \approx \epsilon) \wedge (n \geqslant 0 \Rightarrow (\varphi_r^{\mathsf{is\_num}} \wedge n \approx \mathsf{sti}_r(\mathsf{len}(r)))) \wedge \varphi_r^{\mathsf{sti}}$$

By definition, $\mathsf{from\_int}$ maps negative integers to the empty string. For non-negative integers, the above reduction states that the result of converting integer n to a string is a string r that is a string representation of an integer (due to $\varphi_r^{\mathsf{is\_num}}$), and moreover is such that sti_r for this string results in n. We additionally insist that the formula constraining the semantics of this conversion (φ_r^{sti}) holds.

In practice, these reductions are implemented by introducing a fresh uninterpreted function of type $\mathsf{Int} \Rightarrow \mathsf{Int}$ to represent sti_s for each string s. The functions above are introduced during solving as needed for strings that occur as arguments to $\mathsf{to\_int}$ or those that represent the result of $\mathsf{from\_int}$ according to the above reduction.

Lexicographic Ordering. The (Boolean) equality $(x \preceq y) \approx r$ is equivalent to:

$$(x \approx y \Rightarrow r \approx \top) \wedge (x \not\approx y \Rightarrow \exists k.\, \varphi_{k,x,y}^{\mathsf{diff}} \wedge r \approx \mathsf{code}(x_k) < \mathsf{code}(y_k))$$

where x_k is $\mathsf{substr}(x, k, 1)$, y_k is $\mathsf{substr}(y, k, 1)$, and $\varphi_{k,x,y}^{\mathsf{diff}}$ is:

$$x_k \not\approx y_k \wedge \mathsf{substr}(x, 0, k) \approx \mathsf{substr}(y, 0, k)$$

Above, $\varphi_{k,x,y}^{\mathsf{diff}}$ states that x and y are different and k is the first position at which they differ. If x is a prefix of y or vice versa, then k is the length of the shorter of the two.

The reduction above considers two cases. First, if x and y are the same string, then $x \preceq y$ is trivially true. If x and y are different, then they must differ at some smallest position k. The value of r is equivalent to the comparison of $\mathsf{code}(x_k)$ and $\mathsf{code}(y_k)$. This definition correctly handles cases when k refers to the end position of x or y. If x is a strict prefix of y, k must be $\mathsf{len}(x)$, x_k is the empty string and hence $\mathsf{code}(x_k)$ must be -1. In this case, r must be true since y_k is non-empty and hence the value of $\mathsf{code}(y_k)$ is non-negative; indeed $x \preceq y$ is true when x is a prefix of y. Similarly, r must be false if y is a strict prefix of x; indeed $x \preceq y$ is false when y is a strict prefix of x.

Regular Expression Ranges. In practice, the theory of strings is often extended with memberships constraints of the form $x \in R$, where $\in$ is an infix binary predicate whose first argument is a string and whose second argument R is a regular expression denoting a sublanguage $\mathcal{L}(R)$ of $\mathcal{A}^*$. This constraint holds if x is a member of $\mathcal{L}(R)$.

The code operator can be used for regular expressions that occur often in applications. In particular, the constraint $x \in \mathsf{range}(c_m, c_n)$, where $m \leqslant n$ and c_i is is singleton string constant with code point i, states that x consists of one character whose code point is in the interval $[m, n]$. This is equivalent to $n \leqslant \mathsf{code}(x) \wedge \mathsf{code}(x) \leqslant m$. Our implementation of regular expressions in CVC4 utilizes this as a rewrite rule on membership constraints since it can eliminate the expensive computation of certain regular expression intersections. For example, consider the following equivalent formulas:

$$x \in \mathsf{range}(\texttt{"A"}, \texttt{"M"}) \wedge x \in \mathsf{range}(\texttt{"J"}, \texttt{"Z"}) \quad (1)$$

$$65 \leqslant \mathsf{code}(x) \leqslant 77 \wedge 74 \leqslant \mathsf{code}(x) \leqslant 90 \quad (2)$$

A naive approach to regular expression solving may compute the intersection of the two regular expressions above by explicitly splitting on characters in the ranges of (1). Our approach instead reasons about the arithmetic constraints in (2) and infers the constraint $74 \leqslant \mathsf{code}(x) \leqslant 77$ without expensive case splits. If the latter constraint persists in a saturated configuration, our procedure will then assign x a character in $\mathsf{range}(\texttt{"J"}, \texttt{"M"})$.

5 Evaluation

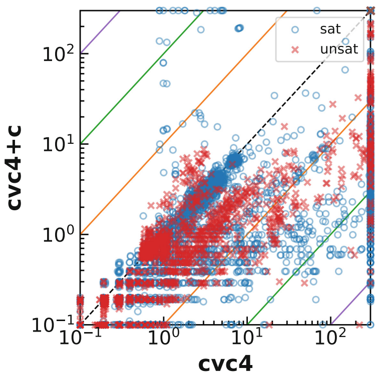
In this section, we evaluate whether our approach is practical and whether code can enable more efficient implementations of common string functions.[5] As outlined in Sect. 3.1, we have implemented our approach in CVC4, which has a state-of-the-art subsolver for the theory of strings with length and regular expressions. We evaluated it on 21,573 benchmarks [1] originating from the concolic execution of Python code involving `int()` using Py-Conbyte [8,32]. The benchmarks make extensive use of to_int, from_int and regular expression ranges. They are divided into four sets, one for each solver used to generate the benchmarks (CVC4, TRAU [3], Z3 [16], and Z3STR3).

We compare two configurations of CVC4 to show the impact of our approach: A configuration (**cvc4+c**) that uses the reductions from Sect. 4 and a configuration (**cvc4**) that disables all code-derivations and uses reductions without code. For regular expression ranges, **cvc4** disables the rewrite to inequalities involving code and uses its regular expression solver to process them. The reductions in **cvc4** use nested ite terms of the form $\mathsf{ite}(c = \texttt{"9"}, 9, \mathsf{ite}(c = \texttt{"8"}, 8, \ldots))$, i.e., do case splitting on the 10 concrete string values that correspond to valid digits, instead of the code operator but keep the reductions the same otherwise. As a point of reference, we also compare against Z3 version 4.8.7, another state-of-the-art string solver. We omit a comparison against Z3STR3 4.8.7 and Z3-TRAU 1.0 [2] (the new version of TRAU) because our experiments have shown that the current versions are unsound.[6]

5 The implementation, the benchmarks, and the results are available at https://cvc4.github.io/papers/ijcar2020-strings.

6 CVC4 and Z3STR3 disagreed on 498 benchmarks whereas CVC4 and Z3-TRAU disagreed on 9. In all instances, Z3STR3 and Z3-TRAU answered that the benchmark is unsatisfiable but accepted CVC4's model when we incorporated it as an additional constraint to the benchmark.

Benchmark Set		**cvc4+c**	**cvc4**	z3
py-conbyte_cvc4	sat	**1344**	1104	1187
	unsat	**8576**	8547	8482
	×	13	282	264
py-conbyte_trauc	sat	**1009**	929	697
	unsat	1424	1407	**1428**
	×	13	110	321
py-conbyte_z3seq	sat	**1354**	1126	1343
	unsat	**5864**	5797	5719
	×	35	330	191
py-conbyte_z3str	sat	**711**	652	692
	unsat	**1227**	1223	1223
	×	3	66	26
Total	sat	**4418**	3811	3919
	unsat	**17091**	16974	16852
	×	64	788	802



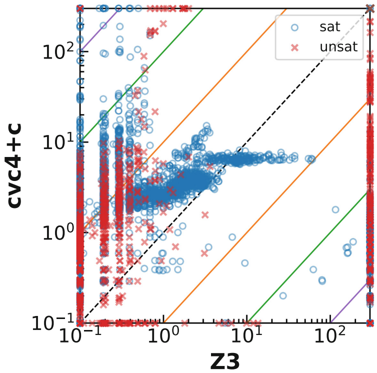


Fig. 5. Number of solved problems per benchmark set and scatter plots comparing the different solvers and configurations on a log-log scale. Best results are in bold. All benchmarks ran with a timeout of 300 s.

We ran our experiments on a cluster with Intel Xeon E5-2637 v4 CPUs running Ubuntu 16.04 and allocated one CPU core, 8 GB of RAM, and 300 s for each job.

Figure 5 summarizes the results of our experiments. The table lists the number of satisfiable and unsatisfiable answers as well as timeouts/memouts (×). z3 ran out of memory on a benchmark but had no other memouts. The figure shows two scatter plots comparing the performance of **cvc4+c** and **cvc4** and comparing **cvc4+c** and z3. Configuration **cvc4** solves more unsatisfiable benchmarks than z3 and fewer satisfiable ones, which suggests that **cvc4** is a reasonable baseline. Our new approach performs significantly better than both **cvc4** and z3. Compared to **cvc4**, configuration **cvc4+c** times out on an order of magnitude fewer benchmarks (64 versus 788) and also improves performance on commonly solved benchmarks, as the scatter plot indicates. While **cvc4** performs worse than z3 on satisfiable benchmarks, **cvc4+c** performs significantly better than both on those benchmarks. The scatter plot indicates that z3 manages to solve a subset of the benchmarks quickly. However, when z3 is not able to solve a benchmark quickly, it is unlikely that it solves it within our timeout. This results in **cvc4+c** having significantly fewer timeouts overall. The results indicate that our new approach is practical and capable of improving the performance of state-of-the-art solvers by enabling more efficient encodings.

6 Conclusion

We have presented a decision procedure for a fragment of strings that includes a string to code point conversion function. We have shown that models can be generated for satisfiable inputs, and that existing techniques for handling strings in SMT solvers can be extended with this procedure. Due to its use for encoding extended string functions, our implementation in CVC4 significantly improves on the state of the art for benchmarks involving string-to-integer conversions and regular expression ranges.

In future work, we plan to extend CVC4 to solve new constraints of interest to user applications. This includes instrumenting our string solver to be capable of generating proofs based on the procedure described in this paper. Further directions such as configuring the solver to generate interpolants for constraints in the theory of strings combined with linear arithmetic could also be explored. Finally we conjecture that efficient support for reasoning about string-to-code conversions can be leveraged for further extensions, such as handling user-defined string transducers.

References

1. str_int_benchmarks (2019). https://github.com/plfm-iis/str_int_benchmarks
2. z3-Trau (2020). https://github.com/guluchen/z3/releases/tag/z3-trau
3. Abdulla, P.A., et al.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Cohen and Vechev [15], pp. 602–617 (2017)
4. Abdulla, P.A., et al.: String constraints for verification. In: Biere and Bloem [12], pp. 150–166 (2014)
5. Abdulla, P.A., et al.: Norn: an SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 462–469. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_29
6. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janků, P.: Chain-free string constraints. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 277–293. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_16
7. Backes, J., et al.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, 30 October–2 November 2018, pp. 1–9. IEEE (2018)
8. Ball, T., Daniel, J.: Deconstructing dynamic symbolic execution. In: Irlbeck, M., Peled, D.A., Pretschner, A. (eds.) Dependable Software Systems Engineering, volume 40 of NATO Science for Peace and Security Series, D: Information and Communication Security, pp. 26–41. IOS Press (2015)

9. 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
10. Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on demand in SAT modulo theories. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 512–526. Springer, Heidelberg (2006). https://doi.org/10.1007/11916277_35
11. 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)
12. Biere, A., Bloem, R. (eds.): CAV 2014. LNCS, vol. 8559. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9
13. 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
14. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. Math. Log. Q. **34**(4), 337–342 (1988)
15. Cohen, A., Vechev, M.T. (eds.): Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017. ACM (2017)
16. 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
17. Enderton, H.B.: A mathematical Introduction to Logic, 2nd edn. Academic Press (2001)
18. Ganesh, V., Berzish, M.: Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. CoRR, abs/1605.09442 (2016)
19. Hu, Q., D'Antoni, L.: Automatic program inversion using symbolic transducers. In: Cohen and Vechev [15], pp. 376–389 (2017)
20. 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)
21. 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 and Bloem [12], pp. 646–662 (2014)
22. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 123–136. ACM (2016)
23. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **145**(2), 147–236 (1977)
24. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
25. Quine, W.V.O.: Concatenation as a basis for arithmetic. J. Symb. Log. **11**(4), 105–114 (1946)

26. 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., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 453–474. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_24
27. 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)
28. The Unicode Consortium. The Unicode Standard, Version 12.1.0 (2019). http://www.unicode.org/versions/Unicode12.1.0/
29. Tinelli, C., Barrett, C., Fontaine, P.: Unicode Strings (2020). http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml
30. Trinh, M., Chu, D., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. 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)
31. 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
32. Wu, W.-C.: Py-Conbyte (2019). https://github.com/spencerwuwu/py-conbyte
33. 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