A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type

Abstract. Strings are widely used in programs, especially in web applications. Integer data type occurs naturally in string-manipulating programs, and is frequently used to refer to lengths of, or positions in, strings. Analysis and testing of string-manipulating programs can be formulated as the path feasibility problem: given a symbolic execution path, does there exist an assignment to the inputs that yields a concrete execution that realizes this path? Such a problem can naturally be reformulated as a string constraint solving problem. Although state-of-the-art string constraint solvers usually provide support for both string and integer data types, they mainly resort to heuristics without completeness guarantees.
In this paper, we propose a decision procedure for a class of string-manipulating programs which includes not only a wide range of complex string operations such as concatenation, replaceAll, reverse, and finite transducers, but also those involving the integer data-type such as length, indexof, and substring. To the best of our knowledge, this represents one of the most expressive string constraint languages that is currently known to be decidable. Our decision procedure is based on a variant of cost register automata (Alur et al. LICS 2013). We provide an implementation of our tool OSTRICH+. We evaluate the performance of OSTRICH+ on a wide range of existing and new benchmarks. The experimental results show that OSTRICH+ is the first string decision procedure capable of tackling finite transducers and integer constraints, whilst its performance is comparable with the state-of-the-art string constraint solvers.

1 Introduction

String-manipulating programs are notoriously subtle, and their potential bugs may bring severe security consequences. A typical example is cross-site scripting (XSS), which is among the OWASP Top 10 Application Security Risks [38]. Integer data type occurs naturally and extensively in string-manipulating programs. An effective and increasingly popular method for identifying bugs, including XSS, is symbolic execution [16]. In a nutshell, this technique analyses static paths through the program being considered. Such a path can be viewed as a constraint φ over appropriate data domains, and symbolic execution tools demand fast constraint solvers to check the satisfiability of φ. Such constraint solvers need to support all native data-type operations occurring in a program.

Typically, mainstream programming languages provide, apart from standard string functions (e.g., concatenation, replace and replaceAll), functions such as length, indexOf, and substring, which can convert strings to integers and vice versa. These functions are indeed heavily utilised in practice; for instance, it was reported [34] that length, indexOf, substring, and variants thereof, comprise over 80% of string function occurrences in 18 popular JavaScript applications, notably outnumbering concatenation. The introduction of integers exacerbates the intricacy of string-manipulating programs, and poses new theoretical and practical challenges in solver development.

When combining strings and integers, decidability can easily be lost; for instance, the string theory with concatenation and letter counting functions is undecidable [13, 20]. It is still a major open problem whether the string theory with concatenation (arguably the simplest string operation) and length function (arguably the most common string-number function) is decidable [23]. One promising approach to retain decidability is to enforce a syntactic restriction to the constraints including solved forms [23], acyclicity [11, 8, 9], and straight-line fragment [29, 18, 19, 24]. On the one hand, such a restriction has led to decidability of string constraint solving with complex string constraints (not only concatenation, but also transducers), and integer constraints (letter-counting, length constraints, IndexOf, etc.), e.g., see [29]. On the other hand, there is a lot of evidence (e.g. from benchmarking) that sufficiently many practical string constraints could satisfy such syntactic restrictions.

Approaches to building practical string solvers could essentially be classified into two. Firstly, one could support as many constraints as possible, but primarily resort to heuristics not offering any completeness/termination guarantee. This is understandable since, as mentioned above, the problem involving both string and integer data types is in general undecidable. Many solvers belong to this category, e.g., CVC4 [28], Z3-str3 [12], S3 [36, 37], Trau [6] (or its variants Trau+ [9] and Z3-Trau [14]), ABC [15], Slent [41] (cf. related work). However, completeness guarantees are valuable since it is well-known that the performance of heuristics can be difficult to predict. The second approach is to develop solvers for decidable fragments supporting both strings and integers (e.g. the fragments of [23, 11, 8, 9, 29, 18, 19, 24]). Some solvers belong to this category including Norn [8], SLOTH [24], and OSTRICH [19]. The aforementioned decidability result of [29] unfortunately does not immediately lead to an implementable decision procedure. The fragment *without* complex string operations (e.g. replaceAll and transducers, but length constraints) can be handled quite well by Norn. The fragment *without* length constraints (but replaceAll and transducers) can be handled effectively by OSTRICH and SLOTH. In fact, most existing solvers that belong to the first category do not support complex string operations like replaceAll and transducers as well. This motivates the following challenging and timely problem: *provide an effective decision procedure that supports string and integer data type, as well as complex string operations, with a completeness guarantee*.

Contribution. The main contribution of this paper is a decision procedure for an expressive class of string constraints involving the integer data-type, which includes not only concatenation, replaceAll, reverse, finite transducers, and regular constraints, but also length, indexOf and substring. The decision procedure utilises a variant of cost-register automata introduced by Alur et al. [10], which we call *cost-enriched finite automata* (CEFA). The crux of the decision procedure is to compute the backward images of CEFAs under string functions, in the same flavour as [19] for string constraints *without* the integer data type. Such an approach is able to treat a wide range of string functions in a generic, and yet simple, way. To the best of our knowledge, the class of string constraints considered in this paper is currently the most expressive string theory involving the integer data type known to enjoy a decision procedure.

Perhaps more importantly, our decision procedure admits an efficient implementation OSTRICH+, which is based on the recent OSTRICH solver [19]. We have performed experiments on a wide range of benchmark suites, including the well-known KALUZA and PYEX, to evaluate the performance of OSTRICH+. The results show that 1) OSTRICH+ so far is the only string constraint solver capable of dealing with finite transducers and integer constraints, and 2) its performance is comparable with the best state-of-the-art string constraint solvers (e.g. CVC4 and Z3-Trau) which are short of completeness guarantees.

Related work. The discussion will mainly focus on (1) theoretical results in terms of decidability; and (2) practical (but generally incomplete) string solvers. We restrict our attention to the results and solvers for string constraints involving the integer data type.

Theoretical results. Recall that the decidability of the string theory of concatenation and linear arithmetic in length functions is a long-standing open problem. Ganesh et al. [23] showed that if the word equations are in a solved form, then the satisfiability of this string theory becomes decidable. On the other hand, Ganesh and Berzish proved that the satisfiability problem becomes undecidable if the string theory is extended with the string-number conversion [22]. Moreover, Lin and Majumdar, as well as Loc and He, have investigated decidable fragments of quadratic word equations, linear arithmetic with the length function, and regular constraints [30, 27].

Recently, researchers have also explored the decidability of string theories with more complex string operations and the integer data type. Lin and Barcelo showed that the straight-line string logic with concatenation, finite transducers, and regular constraints, is decidable, even when extended with length and letter-counting functions, as well as indexOf and charAt functions [29]. This fragment is subsumed by our language. Moreover, the decision procedure in [29] is based on a "global" construction of automata, which is rather difficult to implement (as far as we know, there is no implementation of the procedure therein); in contrast, our decision procedure takes a "local" approach for step-by-step backward computation of pre-images, which is vital for implementation. However, Chen et al. showed that, although the straight-line string logic with the general form of replaceAll (where the replacement parameter can be string variables) is decidable, it would become undecidable when extended with any of the length, indexOf, charAt functions [18]. This result implies that the replaceAll function considered in this paper must not be in the general form, namely, it should avoid string variables in the replacement string, in order to stay inside the decidability boundary.

Practical solvers. Many practical (often incomplete) string constraint solvers have been developed, where various heuristics are used to deal with strings and integers.

The solvers Kaluza [34] and Hampi [26] bound the lengths of strings and reduce the problem to the satisfiability of boolean formulas or formulae in the theory of bit vectors. Sushi [21] and Stranger [43, 42] use finite automata to over approximate sets of values of string variables. Moreover, SLOG and Slent use symbolic approaches to enhance the scalability of the string constraint solving process [40, 41]. In contrast, Trau as well as its variants Z3-Trau and Trau+ utilise flat automata to under-approximate sets of values of string variables [6, 7, 9, 14]. Instead of using automata, Z3-str/Z3-str2/Z3-str3 [45, 44, 12] and CVC4 [28, 32] apply rewrite or algebraic rules to solve the string constraints. Furthermore, S3 and S3P rely on recursive definitions to represent string functions and constraints, which are then unfolded during the constraint solving process

[36, 37]. Finally, ACO-Solver combined the ant colony optimisation meta-heuristic with automata-based string constraint solvers Sushi [21], in order to support reasoning about complex string operations related to XSS vulnerabilities [35].

2 Running example

We use the following JavaScript program as the running example, which defines a function urlXssSanitise for sanitising URLs. A typical URL consists of a hierarchical sequence of components commonly referred to as protocol, host, path, query, and fragment. For instance, in "`http://www.example.com/some/abc.html?name=john#print`", the protocol is "`http`", the host is "`www.example.com`", the path is "`/some/abc.html`", the query is "`name=john`" (preceded by ?), and the fragment is "`print`" (preceded by #). Both the query and the fragment could be empty in a URL. The aim of urlXssSanitise is to mitigate *URL reflection attacks* [5], by filtering out the dangerous substring "`script`" from the query and fragment components of the input URL. URL reflection attacks are a type of cross-site-scripting (XSS) attacks that do not rely on saving malicious code in database, but rather on hiding it in the query or fragment component of URLs, e.g., "`http://www.example.com/some/abc.html?name=<script>alert('xss!');</script>`".

```
function urlXssSanitise(url) {
    var prothostpath='', querfrag = '';
    url = url.trim();
    var qmarkpos = url.indexof('?'), sharppos = url.indexof('#');
    if(qmarkpos >= 0)
    {   prothostpath = url.substr(0, qmarkpos);
        querfrag = url.substr(qmarkpos); }
    else if(sharppos >= 0)
    {   prothostpath = url.substr(0, sharppos);
        querfrag = url.substr(sharppos); }
    else prothostpath = url;
    querfrag = querfrag.replace(/script/g, '');
    url = prothostpath.concat(querfrag);
    return url;
}
```

Note that urlXssSanitise uses the JavaScript sanitisation operation trim that removes whitespace from both ends of a string, which can be conveniently modelled by finite-state transducers. Two string functions involving the integer data type, namely, indexof and substr (indexOf and substring in this paper), as well as concatenation and replace (with the 'g'—global—flag, called replaceAll in this paper), are present.

The program analysis is to ascertain whether urlXssSanitise indeed works, namely, after applying urlXssSanitise to the input URL, "`script`" does not appear. This problem can be reduced to checking whether there is an execution path of urlXssSanitise that produces an output such that its query or fragment component contains occurrences of "`script`". For instance, assuming the "if" branch is executed, we will need to solve the path feasibility of the following JavaScript program in single static assignment (SSA) form,

```
prothostpath =''; querfrag = '';
url1 = url.trim(); qmarkpos = url1.indexof('?');
sharppos = url1.indexof('#'); assert(qmarkpos >= 0);
prothostpath1 = url1.substr(0, qmarkpos);
querfrag1 = url1.substr(qmarkpos);
querfrag2 = querfrag1.replace(/script/g, '');
url2 = prothostpath1.concat(querfrag2);
assert(/script/.test(querfrag2))
```

where the assert(*cond*) statement checks that the condition *cond* is satisfied. As one will see later, this can be directly encoded in our constraint language (cf. Section 4) and handled by the decision procedure (cf. Section 5). Our solver OSTRICH+ can solve the path feasibility of the aforementioned SSA program in several seconds. This is far from trivial since the solver needs to tackle complex string functions such as `trim()` (modelled as finite transducers) and `replaceAll`, as well as indexOf and substring, which is beyond the scope of other existing decision procedures.

3 Preliminaries

We write $\mathbb{N}$ and $\mathbb{Z}$ for the sets of natural and integer numbers, respectively. For $n \in \mathbb{N}$ with $n \geq 1$, $[n]$ denotes $\{1, \ldots, n\}$; for $m, n \in \mathbb{N}$ with $m \leq n$, $[m, n]$ denotes $\{i \in \mathbb{N} \mid m \leq i \leq n\}$. Throughout the paper, Σ is a finite alphabet, and $a, b, \ldots$ range over letters.

Let $\eta_1 : X \to Z$ and $\eta_2 : Y \to Z$ be two functions such that $X \cap Y = \emptyset$. We use $\eta_1 \cup \eta_2$ to denote the function $\eta : X \cup Y \to Z$ such that for each $\alpha \in X \cup Y$, $\eta(\alpha) = \eta_1(\alpha)$ if $\alpha \in X$, and $\eta(\alpha) = \eta_2(\alpha)$ otherwise.

Strings, languages, and transductions. A string over Σ is a (possibly empty) sequence of elements from Σ, denoted by $u, v, w, \ldots$. An empty string is denoted by ε. We write Σ^* (resp., Σ^+) for the set of all (resp. nonempty) strings over Σ.

Let u be a string over Σ. We use $|u|$ to denote the number of letters in u. In particular, $|\varepsilon| = 0$. Moreover, for $a \in \Sigma$, let $|u|_a$ denote the number of occurrences of a in u. Assume $u = a_0 \cdots a_{n-1}$ is nonempty and $i < j \in [0, n-1]$. We let $u[i]$ denote a_i and $u[i, j]$ for the substring $a_i \cdots a_j$.

Let u, v be two strings. We use $u \cdot v$ to denote the *concatenation* of u and v, that is, the string w such that $|w| = |u| + |v|$ and for each $i \in [0, |u| - 1]$, $w[i] = u[i]$, and for each $i \in [0, |v| - 1]$, $w[|u| + i] = v[i]$. The string u is said to be a *prefix* of v if $v = u \cdot v'$ for some string v'. In addition, if $u \neq v$, then u is said to be a *strict* prefix of v. If u is a prefix of v, that is, $v = u \cdot v'$ for some string v', then we use $u^{-1}v$ to denote v'. In particular, $\varepsilon^{-1}v = v$. If $u = a_0 \cdots a_{n-1}$ is nonempty, then we use $u^{(r)}$ to denote the *reverse* of u, that is, $u^{(r)} = a_{n-1} \cdots a_0$.

A *language* over Σ is a subset of Σ^*, denoted by $L_1, L_2, \ldots$. For two languages L_1 and L_2, let $L_1 \cdot L_2$ denote the concatenation of L_1 and L_2, that is, the language $\{u_1 \cdot u_2 \mid u_1 \in L_1, u_2 \in L_2\}$. For a language L and $n \in \mathbb{N}$, we define the *iteration* L^n of L inductively by $L^0 = \{\varepsilon\}$ and $L^n = L \cdot L^{n-1}$ for $n > 0$. We also use L^* to denote an arbitrary number of iterations of L, that is, $L^* = \bigcup_{n \in \mathbb{N}} L^n$. Moreover, let $L^+ = \bigcup_{n \in \mathbb{N} \setminus \{0\}} L^n$.

A *transduction* over Σ is a binary relation over Σ^*, namely, a subset of $\Sigma^* \times \Sigma^*$. We will use $T_1, T_2, \ldots$ to denote transductions. For two transductions T_1 and T_2, we will use $T_1 \cdot T_2$ to denote the *composition* of T_1 and T_2, namely, $T_1 \cdot T_2 = \{(u, w) \in \Sigma^* \times \Sigma^* \mid$ *there exists* $v \in \Sigma^*$ s.t. $(u, v) \in T_1$ and $(v, w) \in T_2\}$.

Regular languages. A language L is *regular* if it can be defined by a regular expression, or equivalently by a finite automaton. Regular expressions RegExp are defined by:

$$e \overset{\text{def}}{=} \emptyset \mid \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^*, \text{ where } a \in \Sigma.$$

Since + is associative and commutative, we also write $(e_1 + e_2) + e_3$ as $e_1 + e_2 + e_3$ for brevity. We use the abbreviation $e^+ \equiv e \cdot e^*$. Moreover, for $\Gamma = \{a_1, \ldots, a_n\} \subseteq \Sigma$, we use the abbreviations $\Gamma \equiv a_1 + \cdots + a_n$ and $\Gamma^* \equiv (a_1 + \cdots + a_n)^*$.

We define $\mathcal{L}(e)$ to be the language defined by e, that is, the set of strings that match e, inductively as follows: $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(a) = \{a\}$, $\mathcal{L}(e_1 + e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$, $\mathcal{L}(e_1 \cdot e_2) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$, $\mathcal{L}(e_1^*) = (\mathcal{L}(e_1))^*$. In addition, we use $|e|$ to denote the number of symbols occurring in e.

A *nondeterministic finite automaton* (NFA) $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, I, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $I, F \subseteq Q$ are the set of initial and final states respectively. For readability, we write a transition $(q, a, q') \in \delta$ as $q \xrightarrow[\delta]{a} q'$. Moreover, when δ is clear from context, we omit δ in $q \xrightarrow[\delta]{a} q'$ and write $q \xrightarrow{a} q'$. The *size* of an NFA $\mathcal{A}$, denoted by $|\mathcal{A}|$, is defined as the number of transitions of $\mathcal{A}$. A *run* of $\mathcal{A}$ on a string $w = a_1 \cdots a_n$ is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \cdots q_{n-1} \xrightarrow{a_n} q_n$ with $q_0 \in I$. The run is *accepting* if $q_n \in F$. A string w is accepted by an NFA $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. In particular, the empty string ε is accepted by $\mathcal{A}$ if $I \cap F \neq \emptyset$. The language defined by $\mathcal{A}$, denoted by $\mathscr{L}(\mathcal{A})$, is the set of strings accepted by $\mathcal{A}$. An NFA $\mathcal{A}$ is said to be *deterministic* if I is a singleton, moreover, for every $q \in Q$ and $a \in \Sigma$, there is at most one state $q' \in Q$ such that $(q, a, q') \in \delta$.

Recognisable relations. Intuitively, a *recognisable relation* is simply a finite union of Cartesian products of regular languages. Formally, an r-ary relation $R \subseteq \Sigma^* \times \cdots \times \Sigma^*$ is *recognisable* if $R = \bigcup_{i=1}^{n} L_1^{(i)} \times \cdots \times L_r^{(i)}$ where $L_j^{(i)}$ is regular for each $j \in [r]$. A *representation* of a recognisable relation $R = \bigcup_{i=1}^{n} L_1^{(i)} \times \cdots \times L_r^{(i)}$ is $(\mathcal{A}_1^{(i)}, \ldots, \mathcal{A}_r^{(i)})_{1 \leq i \leq n}$ such that each $\mathcal{A}_j^{(i)}$ is an NFA with $\mathscr{L}(\mathcal{A}_j^{(i)}) = L_j^{(i)}$. The tuples $(\mathcal{A}_1^{(i)}, \ldots, \mathcal{A}_r^{(i)})$ are called the *disjuncts* of the representation and the NFAs $\mathcal{A}_j^{(i)}$ are called the *atoms* of the representation.

Finite transducers. A *nondeterministic finite transducer* (NFT) $\mathcal{T}$ is an extension of NFA with outputs. Formally, an NFT $\mathcal{T}$ is a tuple $(Q, \Sigma, \delta, I, F)$, where Q, Σ, I, F are as in NFA and the transition relation δ is a finite subset of $Q \times \Sigma \times Q \times \Sigma^*$. Similarly to NFA, for readability, we write a transition $(q, a, q', u) \in \delta$ as $q \xrightarrow[\delta]{a,u} q'$ or $q \xrightarrow{a,u} q'$. The *size* of an NFT $\mathcal{T}$, denoted by $|\mathcal{T}|$, is defined as the sum of the sizes of the transitions of $\mathcal{T}$, where the size of a transition $q \xrightarrow{a,u} q'$ is defined as $|u| + 3$. A run of $\mathcal{T}$ over a string $w = a_1 \cdots a_n$ is a sequence of transitions $q_0 \xrightarrow{a_1,u_1} q_1 \cdots q_{n-1} \xrightarrow{a_n,u_n} q_n$ with

$q_0 \in I$. The run is accepting if $q_n \in F$. The string $u_1 \cdots u_n$ is called the output of the run. The transduction defined by $\mathcal{T}$, denoted by $\mathscr{T}(\mathcal{T})$, is the set of string pairs (w, u) such that there is an accepting run of T on w, with the output u. An NFT $\mathcal{T}$ is said to be *deterministic* if I is a singleton, and, for every $q \in Q$ and $a \in \Sigma$ there is at most one pair $(q', u) \in Q \times \Sigma^*$ such that $(q, a, q', u) \in \delta$. In this paper, we are mainly interested in *functional* finite transducers (FFT), i.e., finite transducers that define functions instead of relations. (For instance, deterministic finite transducers are always functional.)

In this paper, we consider logics involving two data-types, i.e., the string data-type and the integer data-type. We will use $u, v, \ldots$ to denote string constants, $c, d, \ldots$ to denote integer constants, $x, y, \ldots$ to denote string variables, and $i, j, \ldots$ to denote integer variables.

Linear integer arithmetic. A linear integer arithmetic (abbreviated as LIA, essentially Presburger) formula ϕ is defined by the following rules

$$\begin{aligned} t &::= i \mid c \mid ct \mid t + t, \text{ where } c \in \mathbb{Z}, \\ \phi &::= t\ o\ t \mid \neg\phi \mid \phi \vee \phi \mid \exists i.\ \phi, \text{ where } o \in \{=, \neq, \leq, \geq, <, >\}. \end{aligned}$$

The *size* of an LIA formula ϕ, denoted by $|\phi|$, is defined as the number of symbols in ϕ. Let ϕ be an LIA formula and i be a variable occurring in ϕ. Then an occurrence of i in ϕ is said to be *free* if the occurrence is not under the scope of $\exists i$. A formula ϕ is *quantifier-free* if it does not contain quantifiers. The semantics of LIA formulas is standard and its definition is omitted here. For a quantifier-free LIA formula ϕ that contains the free variables $i_1, \ldots, i_k$, we use $\mathcal{M}(\phi)$ to denote the set of models of ϕ, namely, $\mathcal{M}(\phi) = \big\{(n_1, \ldots, n_k) \in \mathbb{Z}^k \mid \phi[n_1/i_1, \ldots, n_k/i_k] \text{ is evaluated to } \mathit{true}\big\}$, where $\phi[n_1/i_1, \ldots, n_k/i_k]$ is the formula obtained from ϕ by simultaneously replacing $i_1, \ldots, i_k$ with $n_1, \ldots, n_k$. An *existential* LIA formula is a LIA formula where all the existential quantifiers are under the scope of an even number of negation symbols.

4 String-Manipulating Programs with Integer Data Type

We consider symbolic execution of string-manipulating programs with numeric conditions (abbreviated as $\mathrm{SL_{int}}$), defined by the following rules,

$$\begin{aligned} S &::= x := y \cdot z \mid x := \mathsf{replaceAll}_{e,u}(y) \mid x := \mathsf{reverse}(y) \mid x := \mathcal{T}(y) \mid \\ &\qquad x := \mathsf{substring}(y, t_1, t_2) \mid \mathsf{assert}\,(\varphi) \mid S\,; S, \\ \varphi &::= x \in \mathcal{A} \mid t_1\ o\ t_2 \mid \varphi \vee \varphi \mid \varphi \wedge \varphi, \end{aligned}$$

where e is a regular expression over Σ, $u \in \Sigma^*$, $\mathcal{T}$ is an FFT, $\mathcal{A}$ is an NFA, $o \in \{=, \neq, \geq, \leq, >, <\}$, and t_1, t_2 are integer terms defined by the following rules,

$$t ::= i \mid c \mid \mathsf{length}(x) \mid \mathsf{indexOf}_v(x, i) \mid ct \mid t + t, \text{ where } c \in \mathbb{Z}, v \in \Sigma^+.$$

We require that the string-manipulating programs are in **single static assignment (SSA) form**. Note that SSA form imposes restrictions only on the assignment statements, but not on the assertions. A string variable x in an $\mathrm{SL_{int}}$ program S is called an *input string variable* of S if it does not appear on the left-hand side of the assignment statements of S. A variable in S is called an *input variable* if it is either an input string variable or an integer variable.

Semantics. The semantics of $\mathrm{SL}_{\mathrm{int}}$ is explained as follows.

- The assignment $x := y \cdot z$ denotes that x is the concatenation of two strings y and z.
- The assignment $x := \mathsf{replaceAll}_{e,u}(y)$ denotes that x is the string obtained by replacing all occurrences of e in y with u, where the *leftmost and longest* matching of e is used. For instance, $\mathsf{replaceAll}_{(ab)^+,c}(aababaab) = ac \cdot \mathsf{replaceAll}_{(ab)^+,c}(aab) = acac$, since the leftmost and longest matching of $(ab)^+$ in $aababaab$ is $abab$. Here we require that the language defined by e does *not* contain the empty string, in order to avoid the troublesome definition of the semantics of the matching of the empty string. We refer the reader to [18] for the formal semantics of the $\mathsf{replaceAll}$ function.
- The assignment $x := \mathsf{reverse}(y)$ denotes that x is the reverse of y.
- The assignment $x := \mathcal{T}(y)$ denotes that $(y, x) \in \mathscr{T}(\mathcal{T})$.
- The assignment $x := \mathsf{substring}(y, t_1, t_2)$ denotes that x is equal to the return value of $\mathsf{substring}(y, t_1, t_2)$, where

$$\mathsf{substring}(y, t_1, t_2) = \begin{cases} \epsilon & \text{if } t_1 < 0 \vee t_1 \geq |y| \vee t_2 = 0 \\ y[t_1, \min\{t_1 + t_2 - 1, |y| - 1\}] & o/w \end{cases}$$

  For instance, $\mathsf{substring}(abaab, -1, 1) = \varepsilon$, $\mathsf{substring}(abaab, 3, 0) = \varepsilon$, $\mathsf{substring}(abaab, 3, 2) = ab$, and $\mathsf{substring}(abaab, 3, 3) = ab$.
- The conditional statement $\mathsf{assert}\,(x \in \mathcal{A})$ denotes that x belongs to $\mathscr{L}(\mathcal{A})$.
- The conditional statement $\mathsf{assert}\,(t_1\ o\ t_2)$ denotes that the value of t_1 is equal to (not equal to, . . .) that of t_2, if $o \in \{=, \neq, \geq, >, \leq, <\}$.
- The integer term $\mathsf{length}(x)$ denotes the length of x.
- The function $\mathsf{indexOf}_v(x, i)$ returns the starting position of the first occurrence of v in x after the position i, if such an occurrence exists, and -1 otherwise. Note that if $i < 0$, then $\mathsf{indexOf}_v(x, i)$ returns $\mathsf{indexOf}_v(x, 0)$, and if $i \geq \mathsf{length}(x)$, then $\mathsf{indexOf}_v(x, i)$ returns -1. For instance, $\mathsf{indexOf}_{ab}(aaba, -1) = 1$, $\mathsf{indexOf}_{ab}(aaba, 1) = 1$, $\mathsf{indexOf}_{ab}(aaba, 2) = -1$, and $\mathsf{indexOf}_{ab}(aaba, 4) = -1$.

Example 1. After adapting the syntax of the program corresponding to the "if" branch of `urlXssSanitise(url)` in Section 2, we get the following $\mathrm{SL}_{\mathrm{int}}$ program,

$$\begin{array}{l}
\mathsf{assert}\,(\mathtt{prothostpath} \in \mathcal{A}_\varepsilon)\,;\mathsf{assert}\,(\mathtt{querfrag} \in \mathcal{A}_\varepsilon)\,; \\
\mathtt{url1} := \mathcal{T}_{\mathrm{trim}}(\mathtt{url});\mathsf{assert}\,(\mathtt{qmarkpos} = \mathsf{indexOf}_?(\mathtt{url1}, 0))\,; \\
\mathsf{assert}\,(\mathtt{sharppos} = \mathsf{indexOf}_\#(\mathtt{url1}, 0))\,;\mathsf{assert}\,(\mathtt{qmarkpos} \geq 0)\,; \\
\mathtt{prothostpath1} := \mathsf{substring}(\mathtt{url1}, 0, \mathtt{qmarkpos}); \\
\mathtt{querfrag1} := \mathsf{substring}(\mathtt{url1}, \mathtt{qmarkpos}, \mathsf{length}(\mathtt{url1}) - \mathtt{qmarkpos}); \\
\mathtt{querfrag2} := \mathsf{replaceAll}_{\mathtt{script},\,\varepsilon}(\mathtt{querfrag1}); \\
\mathtt{url2} := \mathtt{prothostpath1} \cdot \mathtt{querfrag2};\mathsf{assert}\,(\mathtt{querfrag2} \in \mathcal{A}_{\Sigma^*\mathtt{script}\Sigma^*})
\end{array}$$

where $\mathcal{A}_\varepsilon$ is the NFA defining $\{\varepsilon\}$, $\mathcal{T}_{\mathrm{trim}}$ is an NFT to model the sanitisation operation `trim()`, and $\mathcal{A}_{\Sigma^*\mathtt{script}\Sigma^*}$ is the NFA defining $\{w\mathtt{script}w' \mid w, w' \in \Sigma^*\}$.

Remark 1. Note that for simplicity, strictly speaking, string constants are not allowed in the assignments. For instance, $x := \mathsf{reverse}(``abc")$ is disallowed. However, this is not a real restriction because it can be written as $x := \mathsf{reverse}(y); \mathsf{assert}\,(y \in \mathcal{A}_{abc})$, where y is a fresh variable and $\mathcal{A}_{abc}$ is the NFA which accepts "abc" only.

Remark 2. The function $\mathsf{replaceAll}_{e,u}$ can be seen as a special case of FFT, but the transformation to an equivalent FFT $\mathcal{T}_{\mathsf{replaceAll}_{e,u}}$ may incur an exponential blowup [18].

To exemplify the expressiveness of our language, we note that the function $\mathsf{charAt}(x, i)$ which returns $x[i]$ (i.e., the character of x at the position i) can be seen as a special case of substring, namely $\mathsf{charAt}(x, i) \equiv \mathsf{substring}(x, i, 1)$. Furthermore, string inequality $x \neq y$ can be expressed as the following $\mathrm{SL}_{\mathrm{int}}$ program (denoted by $S_{x \neq y}$)

$$z_1 := \mathsf{charAt}(x, i); z_2 := \mathsf{charAt}(y, i);$$
$$\mathsf{assert}\,(\mathsf{length}(x) \neq \mathsf{length}(y) \vee \textstyle\bigvee_{a \in \Sigma}(z_1 \in \mathcal{A}_a \wedge z_2 \in \mathcal{A}_{\Sigma \setminus a}))\,,$$

where z_1, z_2 are two freshly introduced string variables, and $\mathcal{A}_a$ (resp. $\mathcal{A}_{\Sigma \setminus a}$) is the NFA accepting $\{a\}$ (resp. $\Sigma \setminus \{a\}$). Intuitively, two strings are different if their lengths are different or otherwise, there exists some position where the characters of the two strings are different.
Path feasibility problem. Given a $\mathrm{SL}_{\mathrm{int}}$ program S, decide whether there are valuations of input variables so that S can execute to the end.

5 Decision Procedures for Path Feasibility

In this section, we present a decision procedure for the path feasibility problem of $\mathrm{SL}_{\mathrm{int}}$. A distinguished feature of the decision procedure is that it conducts backward computation which is local and can be done in a modular way. To support this, we extend a regular language with quantitative information of the strings in the language, giving rise to cost-enriched regular languages and corresponding finite automata (Section 5.1). The crux of the decision procedure is thus to show that the pre-images of cost-enriched regular languages under the string operations in $\mathrm{SL}_{\mathrm{int}}$ (i.e., concatenation $\cdot$, $\mathsf{replaceAll}_{e,u}$, reverse, FFTs $\mathcal{T}$, and substring) are representable by so called cost-enriched recognisable relations (Section 5.2). The overall decision procedure is presented in Section 5.3, supplied by additional complexity analysis and examples.

5.1 Cost-Enriched Regular Languages and Recognisable Relations

Let $k \in \mathbb{N}$ with $k > 0$. A *k-cost-enriched string* is $(w, (n_1, \cdots, n_k))$ where w is a string and $n_i \in \mathbb{Z}$ for all $i \in [k]$. A *k-cost-enriched language* L is a subset of $\Sigma^* \times \mathbb{Z}^k$. For our purpose, we identify a "regular" fragment of cost-enriched languages as follows.

Definition 1 (Cost-enriched regular languages). *Let $k \in \mathbb{N}$ with $k > 0$. A k-cost-enriched language is* regular *(abbreviated as CERL) if it can be accepted by a* cost-enriched finite automaton.

A cost-enriched finite automaton (CEFA) $\mathcal{A}$ is a tuple $(Q, \Sigma, R, \delta, I, F)$ where

- *Q, Σ, I, F are defined as in NFAs,*
- *$R = (r_1, \cdots, r_k)$ is a vector of (mutually distinct)* cost registers*,*
- *δ is the transition relation which is a finite set of tuples (q, a, q', η) where $q, q' \in Q$, $a \in \Sigma$, and $\eta : R \to \mathbb{Z}$ is a cost register update function.*
  For convenience, we usually write $(q, a, q', \eta) \in \Delta$ as $q \xrightarrow{a,\eta} q'$.

A run *of $\mathcal{A}$ on a k-cost-enriched string $(a_1 \cdots a_m, (n_1, \cdots, n_k))$ is a transition sequence $q_0 \xrightarrow{a_1, \eta_1} q_1 \cdots q_{m-1} \xrightarrow{a_m, \eta_m} q_m$ such that $q_0 \in I$ and $n_i = \sum\limits_{1 \le j \le m} \eta_j(r_i)$ for each $i \in [k]$ (Note that the initial values of cost registers are zero). The run is* accepting *if $q_m \in F$. A k-cost-enriched string $(w, (n_1, \cdots, n_k))$ is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on $(w, (n_1, \cdots, n_k))$. In particular, (ε, n) is accepted by $\mathcal{A}$ if $n = 0$ and $I \cap F \neq \emptyset$. The k-cost-enriched language defined by $\mathcal{A}$, denoted by $\mathscr{L}(\mathcal{A})$, is the set of k-cost-enriched strings accepted by $\mathcal{A}$.*

The *size* of a CEFA $\mathcal{A} = (Q, \Sigma, R, \delta, I, F)$, denoted by $|\mathcal{A}|$, is defined as the sum of the sizes of its transitions, where the size of each transition (q, a, q', η) is $\sum\limits_{r \in R} \lceil \log_2(|\eta(r)|) \rceil + 3$. Note here the integer constants in $\mathcal{A}$ are encoded in binary.

Remark 3. CEFAs can be seen as a variant of Cost Register Automata [10], by admitting nondeterminism and discarding partial final cost functions. CEFAs are also closely related to monotonic counter machines [29]. The main difference is that CEFAs discard guards in transitions and allow binary-encoded integers in cost updates, while monotonic counter machines allow guards in transitions but restrict the cost updates to being monotonic and unary, i.e. 0, 1 only. Moreover, we explicitly define CEFAs as language acceptors for cost-enriched languages.

Example 2 (CEFA for length*).* The string function length can be captured by CEFAs. For any NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, it is not difficult to see that the cost-enriched language $\{(w, \mathsf{length}(w)) \mid w \in \mathscr{L}(\mathcal{A})\}$ is accepted by a CEFA, i.e., $(Q, \Sigma, (r_1), \delta', I, F)$ such that for each $(q, a, q') \in \delta$, we let $(q, a, q', \eta) \in \delta'$, where $\eta(r_1) = 1$.

For later use, we identify a special $\mathcal{A}_{\mathsf{len}} = (\{q_0\}, \Sigma, (r_1), \{(q_0, a, q_0, \eta) \mid \eta(r_1) = 1\}, \{q_0\}, \{q_0\})$. In other words, $\mathcal{A}_{\mathsf{len}}$ accepts $\{(w, \mathsf{length}(w)) \mid w \in \Sigma^*\}$.

We can show that the function $\mathsf{indexOf}_v(\cdot, \cdot)$ can be captured by CEFAs as well, in the sense that, for any NFA $\mathcal{A}$ and constant string v, we can construct a CEFA accepting $\{(w, (n, \mathsf{indexOf}_v(w, n))) \mid w \in \mathscr{L}(\mathcal{A}), n \le \mathsf{indexOf}_v(w, n) < |w|\}$. The construction is slightly technical and can be found in Appendix, Section A.

For convenience, we use $\mathcal{A}_{\mathsf{indexOf}_v}$ which accepts $\{(w, (n, \mathsf{indexOf}_v(w, n))) \mid w \in \Sigma^*, n \le \mathsf{indexOf}_v(w, n) < |w|\}$. Note that $\mathcal{A}_{\mathsf{indexOf}_v}$ does not model the corner cases in the semantics of $\mathsf{indexOf}_v$, for instance, $\mathsf{indexOf}_v(w, n) = -1$ if v does not occur after the position n in w. In the sequel, we will give an example of the construction of $\mathcal{A}_{\mathsf{indexOf}_v}$ for the special case that v is a single letter.

Example 3 (CEFA for $\mathsf{indexOf}_a$*).* Let $a \in \Sigma$. Then $\mathcal{A}_{\mathsf{indexOf}_a} = (\{(q_0, q_1, q_2)\}, \Sigma, (r_1, r_2), \delta_{\mathsf{indexOf}_a}, \{q_0\}, \{q_2\})$, where $\delta_{\mathsf{indexOf}_a}$ comprises the tuples

- (q_0, b, q_0, η) such that $b \in \Sigma$, $\eta(r_1) = 1$, $\eta(r_2) = 1$,
- (q_0, b, q_1, η) such that $b \in \Sigma$, $\eta(r_1) = 0$, $\eta(r_2) = 1$,
- (q_0, a, q_2, η) such that $\eta(r_1) = 0$, $\eta(r_2) = 0$,
- (q_1, b, q_1, η) such that $b \in \Sigma \setminus \{a\}$, $\eta(r_1) = 0$, $\eta(r_2) = 1$,
- (q_1, a, q_2, η) such that $\eta(r_1) = 0$, $\eta(r_2) = 0$,
- (q_2, b, q_2, η) such that $b \in \Sigma$, $\eta(r_1) = 0$, $\eta(r_2) = 0$.

Intuitively, r_1 corresponds to the starting position i of $\mathsf{indexOf}_a(x, i)$, r_2 corresponds to the output of $\mathsf{indexOf}_a(x, i)$, q_0 specifies that the current position is before i, q_1 specifies that the current position is after i, while a has not occurred yet, and q_2 specifies that a has occurred after i.

Given two CEFAs $\mathcal{A}_1 = (Q_1, \Sigma, R_1, \delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, R_2, I_2, F_2)$ with $R_1 \cap R_2 = \emptyset$ (where R_1 and R_2 are treated as sets), the product of $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \times \mathcal{A}_2$, is defined as $(Q_1 \times Q_2, \Sigma, R_1 \cup R_2, \delta, I_1 \times I_2, F_1 \times F_2)$, where δ comprises the tuples $((q_1, q_2), \sigma, (q_1', q_2'), \eta)$ such that $(q_1, \sigma, q_1', \eta_1) \in \delta_1$, $(q_2, \sigma, q_2', \eta_2) \in \delta_2$, and $\eta = \eta_1 \cup \eta_2$.

For a CEFA $\mathcal{A}$, we use $R(\mathcal{A})$ to denote the vector of cost registers occurring in $\mathcal{A}$. Suppose $\mathcal{A}$ is CEFA with $R(\mathcal{A}) = (r_1, \cdots, r_k)$ and $\boldsymbol{i} = (i_1, \cdots, i_k)$ is a vector of mutually distinct integer variables such that $R(\mathcal{A}) \cap \boldsymbol{i} = \emptyset$. We use $\mathcal{A}[\boldsymbol{i}/R(\mathcal{A})]$ to denote the CEFA obtained from $\mathcal{A}$ by simultaneously replacing r_j with i_j for $j \in [k]$.

Definition 2 (Cost-enriched recognisable relations). *Let $(k_1, \cdots, k_l) \in \mathbb{N}^l$ with $k_j > 0$ for every $j \in [l]$. A cost-enriched recognisable relation (CERR) $\mathcal{R} \subseteq (\Sigma^* \times \mathbb{Z}^{k_1}) \times \cdots \times (\Sigma^* \times \mathbb{Z}^{k_l})$ is a finite union of products of CERLs. Formally, $\mathcal{R} = \bigcup_{i=1}^{n} L_{i,1} \times \cdots \times L_{i,l}$, where for every $j \in [l]$, $L_{i,j} \subseteq \Sigma^* \times \mathbb{Z}^{k_j}$ is a CERL. A CEFA representation of $\mathcal{R}$ is a collection of CEFA tuples $(\mathcal{A}_{i,1}, \cdots, \mathcal{A}_{i,l})_{i \in [n]}$ such that $\mathscr{L}(\mathcal{A}_{i,j}) = L_{i,j}$ for every $i \in [n]$ and $j \in [l]$.*

Example 4. The relation

$$\mathcal{R} = \{((w_1, |w_1|), (w_2, |w_2|)) \mid w_1 \in \mathscr{L}((aa)^*), w_2 \in \mathscr{L}(b(bb)^*), |w_1| + |w_2| \geq 2\}$$

is a CERR since $\mathcal{R} = L_{1,1} \times L_{1,2} \cup L_{2,1} \times L_{2,2}$, where $L_{1,1} = \{(w_1, |w_1|) \mid w_1 \in \mathscr{L}((aa)^*)\}$, $L_{1,2} = \{(w_2, |w_2|) \mid w_2 \in \mathscr{L}(bbb(bb)^*)\}$, $L_{2,1} = \{(w_1, |w_1|) \mid w_1 \in \mathscr{L}(aa(aa)^*)\}$, and $L_{2,2} = \{(w_2, |w_2|) \mid w_2 \in \mathscr{L}(b(bb)^*)\}$. Note that $L_{i,j}$ for $i, j \in [2]$ are CERLs, with corresponding CEFAs $\mathcal{A}_{i,j}$ by Example 2. It follows that $(\mathcal{A}_{i,1}, \mathcal{A}_{i,2})_{i \in [2]}$ gives a CEFA representation of $\mathcal{R}$.

5.2 Pre-images of CERLs under string operations

To unify the presentation of the decision procedure, we consider string functions $f : (\Sigma^* \times \mathbb{Z}^{k_1}) \times \cdots \times (\Sigma^* \times \mathbb{Z}^{k_l}) \to \Sigma^*$. (If there is no integer input parameter, then $k_1, \cdots, k_l$ are zero.)

Definition 3 (Cost-enriched pre-images of CERLs). *Suppose that $f : (\Sigma^* \times \mathbb{Z}^{k_1}) \times \cdots \times (\Sigma^* \times \mathbb{Z}^{k_l}) \to \Sigma^*$ is a string function, $L \subseteq \Sigma^* \times \mathbb{Z}^{k_0}$ is a CERL defined by a CEFA $\mathcal{A} = (Q, \Sigma, R, \delta, I, F)$ with $R = (r_1, \cdots, r_{k_0})$. Then the R-cost-enriched pre-image of L under f, denoted by $f_R^{-1}(L)$, is a pair $(\mathcal{R}, \boldsymbol{t})$ such that*

- *$\mathcal{R} \subseteq (\Sigma^* \times \mathbb{Z}^{k_1 + k_0}) \times \cdots \times (\Sigma^* \times \mathbb{Z}^{k_l + k_0})$;*
- *$\boldsymbol{t} = (t_1, \cdots, t_{k_0})$ is a vector of linear integer terms where for each $i \in [k_0]$, t_i is a term whose variables are from $\left\{r_i^{(1)}, \cdots, r_i^{(l)}\right\}$ which are fresh cost registers and are disjoint from R in $\mathcal{A}$;*

– *L is equal to the language comprising the k_0-cost-enriched strings*

$$\left(w_0, t_1\left[d_1^{(1)}/r_1^{(1)}, \cdots, d_1^{(l)}/r_1^{(l)}\right], \cdots, t_{k_0}\left[d_{k_0}^{(1)}/r_{k_0}^{(1)}, \cdots, d_{k_0}^{(l)}/r_{k_0}^{(l)}\right]\right),$$

such that

$$w_0 = f((w_1, \boldsymbol{c_1}), \cdots, (w_l, \boldsymbol{c_l})) \text{ for some } ((w_1, (\boldsymbol{c_1}, \boldsymbol{d_1})), \cdots, (w_l, (\boldsymbol{c_l}, \boldsymbol{d_l}))) \in \mathcal{R},$$

where $\boldsymbol{c_j} \in \mathbb{Z}^{k_j}$, $\boldsymbol{d_j} = (d_1^{(j)}, \cdots, d_{k_0}^{(j)}) \in \mathbb{Z}^{k_0}$ for $j \in [l]$.

The R-cost-enriched pre-image of L under f, say $f_R^{-1}(L) = (\mathcal{R}, \boldsymbol{t})$, is said to be CERR-definable if $\mathcal{R}$ is a CERR.

Definition 3 is essentially a semantic definition of the pre-images. For the decision procedure, one desires an effective representation of a CERR-definable $f_R^{-1}(L) = (\mathcal{R}, \boldsymbol{t})$ in terms of CEFAs. Namely, a CEFA representation of $(\mathcal{R}, \boldsymbol{t})$ (where t_j is over $\left\{r_j^{(1)}, \cdots, r_j^{(l)}\right\}$ for $j \in [k_0]$) is a tuple $((\mathcal{A}_{i,1}, \cdots, \mathcal{A}_{i,l})_{i\in[n]}, \boldsymbol{t})$ such that $(\mathcal{A}_{i,1}, \cdots, \mathcal{A}_{i,l})_{i\in[n]}$ is a CEFA representation of $\mathcal{R}$, where $R(\mathcal{A}_{i,j}) = \left(r'_{j,1}, \cdots, r'_{j,k_j}, r_1^{(j)}, \cdots, r_{k_0}^{(j)}\right)$ for each $i \in [n]$ and $j \in [l]$. (The cost registers $r'_{1,1}, \cdots, r'_{1,k_1}, \cdots, r'_{l,1}, \cdots, r'_{l,k_l}$ are mutually distinct and freshly introduced.)

Example 5 ($\mathsf{substring}_R^{-1}(L)$). Let $\Sigma = \{a\}$ and $L = \{(w, |w|) \mid w \in \mathscr{L}((aa)^*)\}$. Evidently L is a CERL defined by a CEFA $\mathcal{A} = (Q, \Sigma, R, \delta, \{q_0\}, \{q_0\})$ with $Q = \{q_0, q_1\}$, $R = (r_1)$ and $\delta = \{(q_0, a, q_1), (q_1, a, q_0)\}$. Since $\mathsf{substring}$ is from $\Sigma^* \times \mathbb{Z}^2$ to Σ^*, $\mathsf{substring}_R^{-1}(L)$, the R-cost-enriched pre-image of L under $\mathsf{substring}$, is the pair $(\mathcal{R}, t)$, where $t = r_1^{(1)}$ (note that in this case $l = 1$, $k_0 = 1$, and $k_1 = 2$) and

$$\mathcal{R} = \{(w, n_1, n_2, n_2) \mid w \in \mathscr{L}(a^*), n_1 \geq 0, n_2 \geq 0, n_1 + n_2 \leq |w|, n_2 \text{ is even}\},$$

which is represented by $(\mathcal{A}', t)$ such that $\mathcal{A}' = (Q', \Sigma, R', \delta', I', F')$, where

– $Q' = Q \times \{p_0, p_1, p_2\}$, (Intuitively, p_0, p_1, and p_2 denote that the current position is before the starting position, between the starting position and ending position, and after the ending position of the substring respectively.)
– $R' = \left(r'_{1,1}, r'_{1,2}, r_1^{(1)}\right)$,
– $I' = \{(q_0, p_0)\}$, $F' = \{(q_0, p_2), (q_0, p_0)\}$ (where (q_0, p_0) is used to accept the 3-cost-enriched strings $(w, n_1, 0, 0)$ with $0 \leq n_1 \leq |w|$), and
– δ' is

$$\left\{\begin{array}{l} (q_0, p_0) \xrightarrow{a,\eta_1} (q_0, p_0), (q_0, p_0) \xrightarrow{a,\eta_2} (q_1, p_1), (q_1, p_1) \xrightarrow{a,\eta_2} (q_0, p_1), \\ (q_0, p_1) \xrightarrow{a,\eta_2} (q_1, p_1), (q_1, p_1) \xrightarrow{a,\eta_2} (q_0, p_2), (q_0, p_2) \xrightarrow{a,\eta_3} (q_0, p_2) \end{array}\right\},$$

where $\eta_1(r'_{1,1}) = 1$, $\eta_1(r'_{1,2}) = 0$, $\eta_1(r_1^{(1)}) = 0$, $\eta_2(r'_{1,1}) = 0$, $\eta_2(r'_{1,2}) = 1$, and $\eta_2(r_1^{(1)}) = 1$, $\eta_3(r'_{1,1}) = 0$, $\eta_3(r'_{1,2}) = 0$, and $\eta_3(r_1^{(1)}) = 0$.

Therefore, $\mathsf{substring}_R^{-1}(L)$ is CERR-definable.

It turns out that for each string function f in the assignment statements of $\mathrm{SL}_{\mathrm{int}}$, the cost-enriched pre-images of CERLs under f are CERR-definable.

Proposition 1. *Let L be a CERL defined by a CEFA $\mathcal{A} = (Q, \Sigma, R, \delta, I, F)$. Then for each string function f ranging over $\cdot$,* $\mathsf{replaceAll}_{e,u}$*,* $\mathsf{reverse}$*, FFTs $\mathcal{T}$, and* $\mathsf{substring}$*, $f_R^{-1}(L)$ is CERR-definable. In addition,*

- *a CEFA representation of $\cdot_R^{-1}(L)$ can be computed in time $O(|\mathcal{A}|^2)$,*
- *a CEFA representation of $\mathsf{reverse}_R^{-1}(L)$ (resp. $\mathsf{substring}_R^{-1}(L)$) can be computed in time $O(|\mathcal{A}|)$,*
- *a CEFA representation of $(\mathscr{T}(\mathcal{T}))_R^{-1}(L)$ can be computed in time polynomial in $|\mathcal{A}|$ and exponential in $|\mathcal{T}|$,*
- *a CEFA representation of $(\mathsf{replaceAll}_{e,u})_R^{-1}(L)$ can be computed in time polynomial in $|\mathcal{A}|$ and exponential in $|e|$ and $|u|$.*

The proof of Proposition 1 is given in Appendix, Section B.

5.3 The Decision Procedure

Let S be a $\mathrm{SL}_{\mathrm{int}}$ program. Without loss of generality, we assume that for every occurrence of assignments of the form $y := \mathsf{substring}(x, t_1, t_2)$, it holds that t_1 and t_2 are integer variables. This is not really a restriction, since, for instance, if in $y := \mathsf{substring}(x, t_1, t_2)$, neither t_1 nor t_2 is an integer variable, then we introduce fresh integer variables i and j, replace t_1, t_2 by i, j respectively, and add $\mathsf{assert}\,(i = t_1)\,; \mathsf{assert}\,(j = t_2)$ in S. We present a decision procedure for the path feasibility problem of S which is divided into five steps.

Step I: Reducing to atomic asserations.

Note first that in our language, each assertion is a positive Boolean combination of atomic formulas of the form $x \in \mathcal{A}$ or $t_1\ o\ t_2$ (cf. Section 4). Nondeterministically choose, for each assertion $\mathsf{assert}\,(\varphi)$ of S, a set of atomic formulas $\Phi_\varphi = \{\alpha_1, \cdots, \alpha_n\}$ such that φ holds when atomic formulas in Φ_φ are true.

Then each assertion $\mathsf{assert}\,(\varphi)$ in S with $\Phi_\varphi = \{\alpha_1, \cdots, \alpha_n\}$ is replaced by $\mathsf{assert}\,(\alpha_1)\,; \cdots\,; \mathsf{assert}\,(\alpha_n)$, and thus S constains atomic asserations only.

Step II: Dealing with the case splits in the semantics of $\mathsf{indexOf}_v$ and $\mathsf{substring}$.

For each integer term of the form $\mathsf{indexOf}_v(x, i)$ in S, nondeterministically choose one of the following five options (which correspond to the semantics of $\mathsf{indexOf}_v$ in Section 4).

(1) Add $\mathsf{assert}\,(i < 0)$ to S, and replace $\mathsf{indexOf}_v(x, i)$ with $\mathsf{indexOf}_v(x, 0)$ in S.
(2) Add $\mathsf{assert}\,(i < 0)\,; \mathsf{assert}\left(x \in \mathcal{A}_{\overline{\Sigma^* v \Sigma^*}}\right)$ to S, and replace $\mathsf{indexOf}_v(x, i)$ with -1 in S.
(3) Add $\mathsf{assert}\,(i \geq \mathsf{length}(x))$ to S, and replace $\mathsf{indexOf}_v(x, i)$ with -1 in S.
(4) Add $\mathsf{assert}\,(i \geq 0)\,; \mathsf{assert}\,(i < \mathsf{length}(x))$ to S.
(5) Add

$$\mathsf{assert}\,(i \geq 0)\,; \mathsf{assert}\,(i < \mathsf{length}(x))\,; \mathsf{assert}\,(j = \mathsf{length}(x) - i)\,;$$
$$y := \mathsf{substring}(x, i, j); \mathsf{assert}\left(y \in \mathcal{A}_{\overline{\Sigma^* v \Sigma^*}}\right)$$

to S, where y is a fresh string variable, j is a fresh integer variable, and $\mathcal{A}_{\overline{\Sigma^* v \Sigma^*}}$ is an NFA defining the language $\{w \in \Sigma^* \mid v$ does not occur as a substring in $w\}$. Replace $\mathsf{indexOf}_v(x, i)$ with -1 in S.

For each assignment $y := \mathsf{substring}(x, i, j)$ in S, nondeterministically choose one of the following three options (which correspond to the semantics of substring in Section 4).

(1) Add the statements $\mathsf{assert}\,(i \geq 0)\,;\mathsf{assert}\,(i + j \leq \mathsf{length}(x))$ to S.
(2) Add the statements $\mathsf{assert}\,(i \geq 0)\,;\mathsf{assert}\,(i \leq \mathsf{length}(x))\,;\mathsf{assert}\,(i + j > \mathsf{length}(x))$; $\mathsf{assert}\,(i' = \mathsf{length}(x) - i)$ to S, and replace $y := \mathsf{substring}(x, i, j)$ with $y := \mathsf{substring}(x, i, i')$, where i' is a fresh integer variable.
(3) Add the statement $\mathsf{assert}\,(i < 0)\,;\mathsf{assert}\,(y \in \mathcal{A}_\varepsilon)$ to S, and remove $y := \mathsf{substring}(x, i, j)$ from S, where $\mathcal{A}_\varepsilon$ is the NFA defining the language $\{\varepsilon\}$.

Step III: Removing length **and** indexOf.

For each term $\mathsf{length}(x)$ in S, we introduce a *fresh* integer variable i, replace every occurrence of $\mathsf{length}(x)$ by i, and add the statement $\mathsf{assert}\,(x \in \mathcal{A}_{\mathsf{len}}[i/r_1])$ to S. (See Example 2 for the definition of $\mathcal{A}_{\mathsf{len}}$.)

For each term $\mathsf{indexOf}_v(x, i)$ occurring in S, introduce two fresh integer variables i_1 and i_2, replace every occurrence of $\mathsf{indexOf}_v(x, i)$ by i_2, and add the statements $\mathsf{assert}\,(i = i_1)\,;\mathsf{assert}\,(x \in \mathcal{A}_{\mathsf{indexOf}_v}[i_1/r_1, i_2/r_2])$ to S. (See Example 3 for an illustration of $\mathcal{A}_{\mathsf{indexOf}_v}$.)

Step IV: Removing the assignment statements backwards.

Repeat the following procedure until S contains no assignment statements.

Suppose $y := f(x_1, \boldsymbol{i_1}, \cdots, x_l, \boldsymbol{i_l})$ is the *last* assignment of S, where $f : (\Sigma^* \times \mathbb{Z}^{k_1}) \times \cdots \times (\Sigma^* \times \mathbb{Z}^{k_l}) \rightarrow \Sigma^*$ is a string function and $\boldsymbol{i_j} = (i_{j,1}, \cdots, i_{j,k_j})$ for each $j \in [l]$.
Let $\{\mathcal{A}_1, \cdots, \mathcal{A}_s\}$ be the set of all CEFAs such that $\mathsf{assert}\left(y \in \mathcal{A}_j\right)$ occurs in S for every $j \in [s]$. Let $j \in [s]$ and $R(\mathcal{A}_j) = (r_{j,1}, \cdots, r_{j,\ell_j})$. Then from Proposition 1, a CEFA representation of $f^{-1}_{R(\mathcal{A}_j)}(\mathscr{L}(\mathcal{A}_j))$, say $\left(\left(\mathcal{B}^{(1)}_{j,j'}, \cdots, \mathcal{B}^{(l)}_{j,j'}\right)_{j' \in [m_j]}, \boldsymbol{t}\right)$, can be effectively computed from $\mathcal{A}$ and f, where we write

$$R\left(\mathcal{B}^{(j'')}_{j,j'}\right) = \left((r')^{(j'',1)}_j, \cdots, (r')^{(j'',k_{j''})}_j, r^{(j'')}_{j,1}, \cdots, r^{(j'')}_{j,\ell_j}\right)$$

for each $j' \in [m_j]$ and $j'' \in [l]$, and $\boldsymbol{t} = (t_1, \cdots, t_{\ell_j})$. Note that the cost registers $(r')^{(1,1)}_j, \cdots, (r')^{(1,k_1)}_j, \cdots, (r')^{(l,1)}_j, \cdots, (r')^{(l,k_l)}_j, r^{(1)}_{j,1}, \cdots, r^{(1)}_{j,\ell_j}, \cdots, r^{(l)}_{j,1}, \cdots, r^{(l)}_{j,\ell_j}$ are mutually distinct and freshly introduced, moreover, $R\left(\mathcal{B}^{(j'')}_{j,j'_1}\right) = R\left(\mathcal{B}^{(j'')}_{j,j'_2}\right)$ for distinct $j'_1, j'_2 \in [m_j]$.
Remove $y := f(x_1, \boldsymbol{i_1}, \cdots, x_l, \boldsymbol{i_l})$, as well as all the statements $\mathsf{assert}\,(y \in \mathcal{A}_1)$, $\cdots$, $\mathsf{assert}\,(y \in \mathcal{A}_s)$ from S. For every $j \in [s]$, nondeterministically choose $j' \in [m_j]$, and add the following statements to S,

$$\mathsf{assert}\left(x_1 \in \mathcal{B}^{(1)}_{j,j'}\right);\ \cdots;\ \mathsf{assert}\left(x_l \in \mathcal{B}^{(l)}_{j,j'}\right); S_{j,j',\boldsymbol{i_1},\cdots,\boldsymbol{i_l}}; S_{j,\boldsymbol{t}}$$

where

$$\begin{aligned} S_{j,j',\boldsymbol{i_1},\cdots,\boldsymbol{i_l}} \equiv\ & \mathsf{assert}\left(i_{1,1} = (r')^{(1,1)}_{j,j'}\right); \cdots; \mathsf{assert}\left(i_{1,k_1} = (r')^{(1,k_1)}_{j,j'}\right); \\ & \cdots \\ & \mathsf{assert}\left(i_{l,1} = (r')^{(l,1)}_{j,j'}\right); \cdots; \mathsf{assert}\left(i_{l,k_l} = (r')^{(l,k_l)}_{j,j'}\right) \end{aligned}$$

and

$$S_{j,t} \equiv \mathsf{assert}\left(r_{j,1} = t_1\right); \cdots, \mathsf{assert}\left(r_{j,\ell_j} = t_{\ell_j}\right).$$

Step V: Final satisfiability checking.

In this step, S contains no assignment statements and only assertions of the form $\mathsf{assert}(x \in \mathcal{A})$ and $\mathsf{assert}(t_1\ o\ t_2)$ where $\mathcal{A}$ are CEFAs and t_1, t_2 are linear integer terms. Let X denote the set of string variables occurring in S. For each $x \in X$, let $\Lambda_x = \{\mathcal{A}_x^1, \cdots, \mathcal{A}_x^{s_x}\}$ denote the set of CEFAs $\mathcal{A}$ such that $\mathsf{assert}(x \in \mathcal{A})$ appears in S. Moreover, let ϕ denote the conjunction of all the LIA formulas $t_1\ o\ t_2$ occurring in S. It is straightforward to observe that ϕ is over $R' = \bigcup_{x \in X, j \in [s_x]} R(\mathcal{A}_x^j)$. Then the path feasibility of S is reduced to *the satisfiability problem of LIA formulas w.r.t. CEFAs (abbreviated as* $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ *problem)* which is defined as

> deciding whether ϕ is satisfiable w.r.t. $(\Lambda_x)_{x \in X}$, namely, whether there are an assignment function $\theta : R' \to \mathbb{Z}$ and strings $(w_x)_{x \in X}$ such that $\phi[\theta(R')/R']$ holds and $(w_x, \theta(R(\mathcal{A}_x^j))) \in \mathscr{L}(\mathcal{A}_x^j)$ for every $x \in X$ and $j \in [s_x]$.

This $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ problem is decidable and PSPACE-complete:

Proposition 2. $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ *is* PSPACE*-complete.*

A natural idea for the proof for Proposition 2 is to compute, for each string variable $x \in X$, an existential LIA formula ϕ_x defining the Parikh image of the product of the CEFAs in Λ_x, and then to solve the satisfiability of $\phi \wedge \bigwedge_{x \in X} \phi_x$. Nevertheless, computing the product of CEFAs followed by the computation of its Parikh image would require exponential space. To circumvent this exponential space blowup, we utilise Proposition 16 in [29] to show a small model property for the $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ problem: If a model of $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$, specifically, an assignment function $\theta : R' \to \mathbb{Z}$ and strings $(w_x)_{x \in X}$, exists, then θ can be assumed to satisfy that for each $x \in X$ and $r \in R_x = \bigcup_{j \in [s_x]} R(\mathcal{A}_x^j)$, $\theta(r)$ is at most exponential in the size of $\mathcal{A}_x^j$ for $j \in [s_x]$ and $|\phi|$. Since the binary encodings of $\theta(r)$ can be stored in polynomial space, one can *nondeterministically* guess the strings $(w_x)_{x \in X}$, simulate the runs of $\mathcal{A}_x^j$ on w_x, and finally evaluate ϕ over the register values after all CEFAs $\mathcal{A}_x^j$ accept, in polynomial space. A proof of Proposition 2 can be found in Appendix, Section C.

Complexity analysis of the decision procedure. Step I and Step II can be done in nondeterministic linear time. Step III can be done in linear time. In Step IV, for each input string variable x in S, at most exponentially many CEFAs can be generated for x, each of which is of at most exponential size. Therefore, Step IV can be done in nondeterministic exponential space. By Proposition 2, Step V can be done in exponential space. Therefore, we conclude that the path feasibility problem of $\mathrm{SL}_{\mathrm{int}}$ programs is in NEXPSPACE, thus in EXPSPACE by Savitch's theorem [31].

Remark 4. In this paper, we focus on functional finite transducers (cf. Section 3). Our decision procedure is applicable to general finite transducers as well with minor adaptation. However, the EXPSPACE complexity upper-bound does not hold any more, because the distributive property $f^{-1}(L_1 \cap L_2) = f^{-1}(L_1) \cap f^{-1}(L_2)$ for regular languages L_1, L_2 only holds for functional finite transducers f.

Algorithm 1: Function *checkSat* for Step II-III

```
Input: active: set of CEFA constraints, arith: arithmetic constraints, funApps: acyclic
        set of assignment statements.
Result: sat if the input constraints are satisfiable, and unsat otherwise.

1  for each partition $(\mathcal{I}_l)_{l\in[5]}$ of the set of $\mathsf{indexOf}_v(x,i)$ in arith and
        each partition $(\mathcal{J}_l)_{l\in[3]}$ of the set of $\mathsf{substring}(x,i,j)$ in funApps /* the
     partitions refer to (1)-(5) for $\mathsf{indexOf}_v(x,i)$ and (1)-(3) for
     $\mathsf{substring}(x,i,j)$ in Step II of Section 5.3                    */
2  do
       /* Case splits for semantics of indexOf and substring              */
3      $(active, arith, funApps) = indexofCaseSplit(active, arith, funApps, (\mathcal{I}_l)_{l\in[5]})$;
4      $(active, arith, funApps) = substringCaseSplit(active, arith, funApps, (\mathcal{J}_l)_{l\in[3]})$;
5      for each $\mathsf{length}(x)$ occurring in arith do
6          choose a fresh integer variable $i$;
7          $active \leftarrow active \cup \{x \in \mathcal{A}_{\mathsf{len}}[i/r_1]\}$; $arith \leftarrow arith[i/\mathsf{length}(x)]$;
8      for each $\mathsf{indexOf}_v(x,i)$ occurring in arith do
9          choose fresh integer variables $i_1, i_2$;
10         $active \leftarrow active \cup \{x \in \mathcal{A}_{\mathsf{indexOf}_v}[i_1/r_1, i_2/r_2]\}$;
             $arith \leftarrow arith[i_2/\mathsf{indexOf}_v(x,i)] \wedge i = i_1$;
11     if $BackDfsExp(active, \emptyset, arith, funApps)$ then
12         return sat;
13 return unsat;
```

An example to demonstrate the decision procedure applied to the program in Example 1 is provided in the Appendix, Section D.

6 Implementation

We have implemented the decision procedure based on the recent string constraint solver OSTRICH [19], resulting in a new solver OSTRICH+. OSTRICH is written in Scala and based on the SMT solver Princess [33]. OSTRICH+ reuses the parser of Princess, but replaces the NFAs from OSTRICH with CEFAs. Correspondingly, in OSTRICH+, the pre-image computation for concatenation, replaceAll, reverse, and finite transducers is reimplemented, and a new pre-image operator for substring is added. OSTRICH+ also implements CEFA constructions for length and indexOf.

OSTRICH+ performs a depth-first exploration of the search tree resulting from repeatedly splitting the disjunctions (or unions) in the cost-enriched recognisable pre-images of CERLs under string functions, as well as the case splits in the semantics of indexOf and substring. The pseudo-code of Step II-III of the decision procedure in Section 5 is given by the function *checkSat* in Algorithm 1, which calls two functions *indexofCaseSplit* and *substringCaseSplit* for the case splits in the semantics of $\mathsf{indexOf}_v$ and substring respectively. (The details of *indexofCaseSplit* and *substringCaseSplit* can be found in Appendix, Section E.) Moreover, *checkSat* calls a recursive function *BackDfsExp* in Algorithm 2 for the depth-first exploration (Step IV of the decision procedure), which in turn calls a function *CheckCefaLIASat* to solve the $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ problem (Step V). Note that Step I of the decision procedure is handled by the DPLL(T) procedure in Princess and is omitted here.

Algorithm 2: Function *BackDfsExp* for Step IV (depth-first exploration)

```
Input: active, passive: sets of CEFA constraints, arith: arithmetic constraints, funApps:
       acyclic set of assignment statements.
Result: sat if the input constraints are satisfiable, and unsat otherwise.

if $active = \emptyset$ then
    /* Check whether the LIA constraint arith is satisfiable with
       respect to the CEFA constraints in passive (i.e. Step V).   */
    return CheckCefaLIASat(passive, arith);
else
    choose a CEFA constraint $x \in \mathcal{A}$ in active with $R(\mathcal{A}) = (r_1, \cdots, r_k)$;
    if there is an assignment $x := f(y_1, \boldsymbol{i_1}, \ldots, y_l, \boldsymbol{i_l})$ defining x in funApps with
          $\boldsymbol{i_j} = (i_{j,1}, \cdots, i_{j,k_j})$ for $j \in [l]$ then
        compute $f^{-1}_{R(\mathcal{A})}(\mathscr{L}(\mathcal{A})) = \left((\mathcal{A}^{(1)}_j, \cdots, \mathcal{A}^{(l)}_j)_{j \in [n]}, \boldsymbol{t}\right)$ where
         $R\left(\mathcal{A}^{(j')}_j\right) = \left((r')^{(j',1)}, \cdots, (r')^{(j',k_{j'})}, r^{(j')}_1, \cdots, r^{(j')}_k\right)$ for $j \in [n]$ and $j' \in [l]$;
        $active \leftarrow active \setminus \{x \in \mathcal{A}\}$; $passive \leftarrow passive \cup \{x \in \mathcal{A}\}$;
        for $j \leftarrow 1$ to $n$ do
            $active \leftarrow active \cup \{y_1 \in \mathcal{A}^{(1)}_j, \ldots, y_l \in \mathcal{A}^{(l)}_j\}$;
            $arith \leftarrow arith \wedge \bigwedge_{j' \in [l], j'' \in [k_{j'}]} i_{j',j''} = (r')^{(j',j'')} \wedge \bigwedge_{j' \in [k]} r_{j'} = t_{j'}$;
            if $active \cup passive$ is inconsistent then
                continue ;                                   /* backtrack */
            else
                switch BackDfsExp(active, passive, arith, funApps) do
                    case sat do return sat;
                    case unsat do
                        continue ;                           /* backtrack */
        return unsat;
    else
        return BackDfsExp($active \setminus \{x \in \mathcal{A}\}$, $passive \cup \{x \in \mathcal{A}\}$, arith, funApps);
```

Optimisations for solving the $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ *problem.* From Proposition 2, a natural approach to solve the $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ problem is to compute an existential LIA formula defining the Parikh image of products of CEFAs, and then use off-the-shelf SMT solvers (e.g. CVC4 or Z3) to decide the satisfiability of the existential LIA formula. However, our preliminary experiments show that this approach suffers from a scalability issue, in particular, the state-space explosion when computing products of CEFAs. In the implementation of the function *CheckCefaLIASat* in Algorithm 2, we opt to utilise the symbolic model checker nuXmv [17] to mitigate the state-space explosion during the computation of products of CEFAs. The nuXmv tool is a well-known symbolic model checker that is capable of analysing both finite and infinite state systems. Our technique is to encode $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ as an instance of the model checking problem, which can be solved by nuXmv. Since $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ is a problem for quantifier-free LIA formulas and CEFAs that contain integer variables, the $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ problem actually corresponds to the problem of model checking *infinite state systems*.

7 Evaluations

We have compared OSTRICH+ with some of the state-of-the-art solvers on a wide range of benchmarks. We discuss the benchmarks in Section 7.1 and present the experimental results in Section 7.2.

7.1 Benchmarks

Our evaluation focuses on problems that combine string with integer constraints. To this end, we consider the following four sets of benchmarks, all in SMT-LIB 2 format.

Transducer+ is derived from the Transducer benchmark suite of OSTRICH [19]. The Transducer suite involves seven transducers: toUpper (replacing all lowercase letters with their uppercase ones) and its dual toLower, htmlEscape [3] and its dual htmlUnescape, escapeString [2], addslashes [1], and trim [4]. These transducers are collected from Stranger [43] and SLOTH [24]. Initially none of the benchmarks involved integers. In Transducer+, we encode four security-relevant properties of transducers [25], with the help of the functions charAt and length:

- idempotence: given $\mathcal{T}$, whether $\forall x.\ \mathcal{T}(\mathcal{T}(x)) = \mathcal{T}(x)$;
- duality: given $\mathcal{T}_1$ and $\mathcal{T}_2$, whether $\forall x.\ \mathcal{T}_2(\mathcal{T}_1(x)) = x$;
- commutativity: given $\mathcal{T}_1$ and $\mathcal{T}_2$, whether $\forall x.\ \mathcal{T}_2(\mathcal{T}_1(x)) = \mathcal{T}_1(\mathcal{T}_2(x))$;
- equivalence: given $\mathcal{T}_1$ and $\mathcal{T}_2$, whether $\forall x.\ \mathcal{T}_1(x) = \mathcal{T}_2(x)$.

For instance, we encode the non-idempotence of $\mathcal{T}$ into the path feasibility of the $\mathrm{SL}_{\mathrm{int}}$ program $y := \mathcal{T}(x); z := \mathcal{T}(y); S_{y \neq z}$, where y and z are two fresh string variables, and $S_{y \neq z}$ is the $\mathrm{SL}_{\mathrm{int}}$ program encoding $y \neq z$ (see Section 4 for its details). We also include in Transducer+ three instances generated from the running example in Section 2, where $\mathcal{T}_{\mathrm{trim}}$ is used. In total, we obtain 94 instances for the Transducer+ suite.

SLOG+ is adapted from the SLOG benchmark suite [40], containing 3,511 instances about strings only. We obtain SLOG+ by choosing a string variable x for each instance, and adding the statement $\mathsf{assert}(\mathsf{length}(x) < 2\ \mathsf{indexOf}_a(x, 0))$ for some $a \in \Sigma$. As in [19], we split SLOG+ into SLOG+(replace) and SLOG+(replaceall), comprising 3,391 and 120 instances respectively. In addition to the indexOf and length functions, the benchmarks use regular constraints and concatenation; SLOG+(replace) also contains the replace function (replacing the first occurrence), while SLOG+(replaceall) uses the replaceAll function (replacing all occurrences).

PyEx [32] contains 25,421 instances derived by the PyEx tool, a symbolic execution engine for Python programs. The PyEx suite was generated by the CVC4 group from four popular Python packages: httplib2, pip, pymongo, and requests. These instances

Benchmark	Output	CVC4	Z3-str3	Z3-Trau	OSTRICH+
TRANSDUCER+(94)	sat	–	–	–	**84**
	unsat	–	–	–	**4**
	inconclusive	–	–	–	6
SLOG+(REPLACEALL)(120)	sat	**104**	–	–	98
	unsat	11	–	–	**12**
	inconclusive	5	–	–	10
SLOG+(REPLACE)(3,391)	sat	**1,309**	878	–	584
	unsat	**2,082**	2,066	–	**2,082**
	inconclusive	0	447	–	725
PYEX-td(5,569)	sat	4,224	4,068	**4,266**	4,141
	unsat	1,284	1,289	**1,295**	1,203
	inconclusive	61	212	8	225
PYEX-z3(8,414)	sat	6,346	6,040	**7,003**	5,489
	unsat	1,358	1,370	**1,394**	1,239
	inconclusive	710	1,004	17	1,686
PYEX-zz(11,438)	sat	10,078	8,804	**10,129**	9,033
	unsat	1,204	1,207	**1,222**	868
	inconclusive	156	1,427	87	1,537
KALUZA(47,284)	sat	**35,264**	33,438	34,769	27,962
	unsat	**12,014**	11,799	**12,014**	9,058
	inconclusive	6	2,047	501	10,264
Total(76,310)	solved	**75,278**	70,959	72,092	61,857
	unsolved	1,032	5,351	4,218	14,453

Table 1. Experimental results on different benchmark suites. '–' means that the tool is not applicable to the benchmark suite, and 'inconclusive' means that a tool gave up, timed out, or crashed.

use regular constraints, concatenation, length, substring, and indexOf functions. Following [32], the PYEX suite is further divided into three parts: PYEX-td, PYEX-z3 and PYEX-zz, comprising 5,569, 8,414 and 11,438 instances, respectively.

KALUZA [34] is the most well-known benchmark suite in literature, containing 47,284 instances with regular constraints, concatenation, and the length function. The 47,284 benchmarks include 28,032 satisfiable and 9,058 unsatisfiable problems in SSA form.

7.2 Experiments

We compare OSTRICH+ to CVC4 [28], Z3-str3 [45], and Z3-Trau [14]. The experiments are executed on a computer with an Intel Xeon Silver 4210 2.20GHz and 2.19GHz CPU (2-core) and 8GB main memory, running 64bit Ubuntu 18.04 LTS OS and Java 1.8. We use a timeout of 30 seconds (wall-clock time), and report the number of satisfiable and unsatisfiable problems solved by each of the systems. Table 1 summarises the experimental results. We did not observe incorrect answers by any tool.

There are two additional state-of-the-art solvers Slent and Trau+ which were not included in the evaluation. We exclude Slent [39] because it uses its own input format laut, which is different from the SMT-LIB 2 format used for our benchmarks; also, TRANSDUCER+ is beyond the scope of Slent. Trau+ [9] integrates Trau with Sloth to deal with both finite transducers and integer constraints. We were unfortunately unable to obtain a working version of Trau+ by the deadline, possibly because Trau requires two separate versions of Z3 to run. In addition, the algorithm [9] focuses on length-preserving transducers, which means that TRANSDUCER+ is beyond the scope of Trau+.

OSTRICH+ is the only tool applicable to the problems in TRANSDUCER+. With a timeout of 30s, OSTRICH+ can solve 88 of the benchmarks, but this number rises to 94 when using a longer timeout of 600s. Given the complexity of those benchmarks, this is an encouraging result.

On SLOG+(REPLACEALL), OSTRICH+ and CVC4 are very close: OSTRICH+ solves 98 satisfiable instances, slightly less than the 104 instances solved by CVC4, while OSTRICH+ solves one more unsatisfiable instance than CVC4 (12 versus 11). The suite is beyond the scope of Z3-str3 and Z3-Trau, which do not support replaceAll.

On SLOG+(REPLACE), OSTRICH+, CVC4, and Z3-str3 solve a similar number of unsatisfiable problems, while CVC4 solves the largest number of satisfiable instances (1,309). The suite is beyond the scope of Z3-Trau which does not support replace.

On the three PYEX suites, Z3-Trau consistently solves the largest number of instances by some margin. OSTRICH+ solves a similar number of instances as Z3-str3. Interpreting the results, however, it has to be taken into account that PYEX includes 1,334 instances that are *not* in SSA form, which are beyond the scope of OSTRICH+.

The KALUZA problems can be solved most effectively by CVC4. OSTRICH+ can solve almost all of the 28,032 satisfiable problems in SSA form, and all 9,058 unsatisfiable problems in SSA form. This is consistent with the results in [19] for OSTRICH.

In summary, we can observe that OSTRICH+ is competitive with other solvers, while it is also able to handle benchmarks that are beyond the scope of the other tools due to the combination of string functions (in particular transducers) and integer constraints. Interestingly, the experiments show that OSTRICH+, at least in its current state, is better at solving unsatisfiable problems than satisfiable problems; this might be an artefact of the use of nuXmv for analysing products of CEFAs. We expect that further optimisation of our algorithm will lead to additional performance improvements. For instance, a natural optimisation that is not yet included in our implementation is to use standard finite automata like in OSTRICH, as opposed to CEFAs, for simpler problems such as the KALUZA benchmarks.

8 Conclusion

In this paper, we have proposed an expressive string constraint language which can specify constraints on both strings and integers. We provided an automata-theoretic decision procedure for the path feasibility problem of this language. The decision procedure is simple, generic, and amenable to implementation, giving rise to a new solver OSTRICH+. We have evaluated OSTRICH+ on a wide range of existing and newly created benchmarks, and have obtained very encouraging results. OSTRICH+ is shown to be the first solver which is capable of tackling finite transducers and integer constraints with completeness guarantees. Meanwhile, it demonstrates competitive performance against some of the best state-of-the-art string constraint solvers.

References

1. The addslashes operator. `http://php.net/manual/en/function.addslashes.php`.
2. The escapestring sanitisation operator. `https://github.com/google/closure-library/blob/master/closure/goog/string/string.js#L878`.
3. The htmlescape sanitisation operator. `https://github.com/google/closure-library/blob/master/closure/goog/string/string.js#L549`.
4. The trim sanitisation operator. `https://www.php.net/manual/en/function.trim.php`.
5. Url reflection attacks. `https://sites.google.com/site/xssvulnerabilities/types-of-javascript-injection/url-reflection-attacks`.
6. P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. Flatten and conquer: a framework for efficient analysis of string constraints. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 602–617, 2017.
7. P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. Trau: SMT solver for string constraints. In N. Bjørner and A. Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–5. IEEE, 2018.
8. P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *CAV*, pages 150–166, 2014.
9. P. A. Abdulla, M. F. Atig, B. P. Diep, L. Holík, and P. Janku. Chain-free string constraints. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, pages 277–293, 2019.
10. R. Alur, L. D'Antoni, J. Deshmukh, M. Raghothaman, and Y. Yuan. Regular functions and cost register automata. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 13–22. IEEE Computer Society, 2013.
11. P. Barceló, D. Figueira, and L. Libkin. Graph logics with rational relations. *Logical Methods in Computer Science*, 9(3), 2013.
12. M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59, 2017.
13. J. R. Büchi and S. Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. In *The Collected Works of J. Richard Büchi*, pages 671–683. Springer, 1990.
14. D. Bui and contributors. Z3-trau. `https://github.com/diepbp/z3-trau`, 2019.
15. T. Bultan and contributors. Abc string solver. `https://github.com/vlab-cs-ucsb/ABC`, 2015.
16. C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.
17. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 334–342, 2014.
18. T. Chen, Y. Chen, M. Hague, A. W. Lin, and Z. Wu. What is decidable about string constraints with the replaceall function. *PACMPL*, 2(POPL):3:1–3:29, 2018.
19. T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL), Jan. 2019.

20. J. D. Day, V. Ganesh, P. He, F. Manea, and D. Nowotka. The satisfiability of word equations: Decidable and undecidable theories. In *Reachability Problems - 12th International Conference, RP 2018, Marseille, France, September 24-26, 2018, Proceedings*, pages 15–29, 2018.
21. X. Fu, M. C. Powell, M. Bantegui, and C.-C. Li. Simple linear string constraints. *Formal Aspects of Computing*, 25(6):847–891, Nov 2013.
22. V. Ganesh and M. Berzish. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. *CoRR*, abs/1605.09442, 2016.
23. V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard. Word equations with length constraints: What's decidable? In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, pages 209–226, 2012.
24. L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL):4:1–4:32, 2018.
25. P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security Symposium*, 2011.
26. A. Kiezun et al. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25, 2012.
27. Q. L. Le and M. He. A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, pages 350–372, 2018.
28. T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV*, pages 646–662, 2014.
29. A. W. Lin and P. Barceló. String solving with word equations and transducers: Towards a logic for analysing mutation XSS. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 123–136. Springer, 2016.
30. A. W. Lin and R. Majumdar. Quadratic word equations with length constraints, counter systems, and presburger arithmetic with divisibility. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, pages 352–369, 2018.
31. C. H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994.
32. A. Reynolds, M. Woo, C. W. Barrett, D. Brumley, T. Liang, and C. Tinelli. Scaling up DPLL(T) string solvers using context-dependent simplification. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2017.
33. P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, LNCS 5330*, pages 274–289. Springer, 2008.
34. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. 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*, pages 513–528, 2010.
35. J. Thomé, L. K. Shar, D. Bianculli, and L. C. Briand. Search-driven string constraint solving for vulnerability detection. In S. Uchitel, A. Orso, and M. P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 198–208. IEEE / ACM, 2017.
36. M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS*, pages 1232–1243, 2014.

37. M. Trinh, D. Chu, and J. Jaffar. Progressive reasoning over recursively-defined strings. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 218–240. Springer, 2016.
38. A. van der Stock, B. Glas, N. Smithline, and T. Gigler. OWASP Top 10 – 2017. `https://www.owasp.org/index.php/Top_10-2017_Top_10`, 2017. Referred January 2018.
39. H. Wang, S. Chen, F. Yu, and J. R. Jiang. A symbolic model checking approach to the analysis of string and length constraints. In M. Huchard, C. Kästner, and G. Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 623–633. ACM, 2018.
40. H. Wang, T. Tsai, C. Lin, F. Yu, and J. R. Jiang. String analysis via automata manipulation with logic circuit representation. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2016.
41. H.-E. Wang, S.-Y. Chen, F. Yu, and J.-H. R. Jiang. A symbolic model checking approach to the analysis of string and length constraints. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 623–633. Association for Computing Machinery, 2018.
42. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS*, pages 154–157, 2010. Benchmark can be found at `http://www.cs.ucsb.edu/~vlab/stranger/`.
43. F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Form. Methods Syst. Des.*, 44(1):44–70, 2014.
44. Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 235–254. Springer, 2015.
45. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a Z3-based string solver for web application analysis. In *ESEC/SIGSOFT FSE*, pages 114–124, 2013.

A Construction of $\mathcal{A}_{\mathsf{indexOf}_v}$

In this section, we show that the function $\mathsf{indexOf}_v(\cdot,\cdot)$ can be captured by CEFA. Technically, for any NFA $\mathcal{A}$ and constant string v, we can construct a CEFA accepting $\{(w,(n,\mathsf{indexOf}_v(w,n))) \mid w \in \mathscr{L}(\mathcal{A})\}$.

For this purpose, we need a concept of window profiles of string positions w.r.t. v, which are elements of $\{\bot,\top\}^{n-1}$. The window profiles facilitate recognising the first occurrence of v in the input string. Intuitively, given a string u, the window profile of a position i in u w.r.t. v encodes the matchings of prefixes of v to the suffixes of $u[0,i]$ (see [18] for the details). For $\pi = \pi_1 \cdots \pi_{n-1} \in \{\bot,\top\}^{n-1}$ and $b \in \Sigma$, we use $\mathrm{uwp}(\boldsymbol{\pi}, b)$ to represent the window profile updated from π after reading the letter b, specifically, $\mathrm{uwp}(\boldsymbol{\pi}, b) = \boldsymbol{\pi'}$ such that

- $\pi'_1 = \top$ iff $b = a_1$,
- for each $i \in [n-2]$, $\pi'_{i+1} = \top$ iff $\pi_i = \top$ and $b = a_{i+1}$.

Let WP_v denote the set of window profiles of string positions w.r.t. v. From the result in [18], we know that $|WP_v| \leq |v|$.

Suppose $v = a_1 \cdots a_n$ with $n \geq 2$. Then $\mathsf{indexOf}_v$ is captured by the CEFA $\mathcal{A}_{\mathsf{indexOf}_v} = (Q,\Sigma,R,\delta,I,F)$, such that

- $Q = \{q_0,q_1\} \cup WP_v \cup WP_v \times [n]$,
- $R = (r_1,r_2)$ (where r_1,r_2 represent the input and output positions of $\mathsf{indexOf}_v$ respectively),
- $I = \{q_0\}$,
- $F = \{q_1\}$, and
- δ comprises
  - the tuples (q_0,a,q_0,η) such that $a \in \Sigma$, $\eta(r_1) = 1$, and $\eta(r_2) = 1$,
  - the tuples $(q_0,a,\boldsymbol{\pi},\eta)$ such that $a \in \Sigma$, $\boldsymbol{\pi} = \theta\bot^{n-2}$ where $\theta = \top$ iff $a = a_1$, $\eta(r_1) = 0$, and $\eta(r_2) = 0$ (recall that the first position of a string is 0),
  - the tuples $(\boldsymbol{\pi},a,\mathrm{uwp}(\boldsymbol{\pi},a),\eta)$ such that $\boldsymbol{\pi} \in WP_u$, $a \in \Sigma$, $\pi_{n-1} = \bot$ or $a \neq a_n$, $\eta(r_1) = 0$, and $\eta(r_2) = 1$,
  - the tuples $(\boldsymbol{\pi},a,(\mathrm{uwp}(\boldsymbol{\pi},a),1),\eta)$ such that $\boldsymbol{\pi} \in WP_u$, $a = a_1$, $\pi_{n-1} = \bot$ or $a \neq a_n$, $\eta(r_1) = 0$, and $\eta(r_2) = 1$,
  - the tuples $((\boldsymbol{\pi},i),a,(\mathrm{uwp}(\boldsymbol{\pi},a),i+1),\eta)$ such that $\boldsymbol{\pi} \in WP_u$, $i \in [n-2]$, $a = a_{i+1}$, $\pi_{n-1} = \bot$ or $a \neq a_n$, $\eta(r_1) = 0$, and $\eta(r_2) = 0$,
  - the tuples $((\boldsymbol{\pi},n-1),a,q_1,\eta)$ such that $\boldsymbol{\pi} \in WP_u$, $a = a_n$, $\eta(r_1) = 0$, and $\eta(r_2) = 0$,
  - the tuples (q_1,a,q_1,η) such that $a \in \Sigma$, $\eta(r_1) = 0$, and $\eta(r_2) = 0$.

B Proof of Proposition 1

Proposition 1. *Let L be a CERL defined by a CEFA $\mathcal{A} = (Q,\Sigma,R,\delta,I,F)$. Then for each string function f ranging over $\cdot$,* $\mathsf{replaceAll}_{e,u}$*,* $\mathsf{reverse}$*, FFTs $\mathcal{T}$, and* $\mathsf{substring}$*, $f_R^{-1}(L)$ is CERR-definable. In addition,*

- *a CEFA representation of $\cdot_R^{-1}(L)$ can be computed in time $O(|\mathcal{A}|^2)$,*

– *a CEFA representation of* $\mathsf{reverse}_R^{-1}(L)$ *(resp.* $\mathsf{substring}_R^{-1}(L)$*) can be computed in time* $O(|\mathcal{A}|)$*,*
– *a CEFA representation of* $(\mathscr{T}(\mathcal{T}))_R^{-1}(L)$ *can be computed in time polynomial in* $|\mathcal{A}|$ *and exponential in* $|\mathcal{T}|$*,*
– *a CEFA representation of* $(\mathsf{replaceAll}_{e,u})_R^{-1}(L)$ *can be computed in time polynomial in* $|\mathcal{A}|$ *and exponential in* $|e|$ *and* $|u|$*.*

Proof. Let $\mathcal{A} = (Q, \Sigma, R, \delta, I, F)$ be a CEFA with $R = (r_1, \cdots, r_k)$. We show how to construct a CEFA representation of $f_R^{-1}(L)$ for each function f in $\mathrm{SL}_{\mathrm{int}}$.

$\cdot_R^{-1}(L)$. A CEFA representation of $\cdot_R^{-1}(L)$ is given by $((\mathcal{A}_{I,q}, \mathcal{A}_{q,F})_{q \in Q}, \boldsymbol{t})$, where

– $\mathcal{A}_{I,q} = (Q, \Sigma, R^{(1)}, \delta^{(1)}, I, \{q\})$ and $\mathcal{A}_{q,F} = (Q, \Sigma, R^{(2)}, \delta^{(2)}, \{q\}, F)$ such that
  • $R^{(1)} = (r_1^{(1)}, \cdots, r_k^{(1)})$, $R^{(2)} = (r_1^{(2)}, \cdots, r_k^{(2)})$,
  • $\delta^{(1)}$ comprises the tuples (q, a, q', η') satisfying that there exists η such that $(q, a, q', \eta) \in \delta$ and for each $j \in [k]$, and $\eta'(r_j^{(1)}) = \eta(r_j)$, similarly for $\delta^{(2)}$,
– and $\boldsymbol{t} = (r_1^{(1)} + r_1^{(2)}, \cdots, r_k^{(1)} + r_k^{(2)})$.

Note that the size of $((\mathcal{A}_{I,q}, \mathcal{A}_{q,F})_{q \in Q}, \boldsymbol{t})$ is $O(|\mathcal{A}|^2)$.

$\mathsf{reverse}_R^{-1}(L)$. A CEFA representation of $\mathsf{reverse}_R^{-1}(L)$ is given by $(\mathcal{A}^{(r)}, \boldsymbol{t})$, where

– $\mathcal{A}^{(r)} = (Q, \Sigma, R^{(1)}, \delta', F, I)$ such that
  • $R^{(1)} = (r_1^{(1)}, \cdots, r_k^{(1)})$, and
  • δ' comprises the tuples (q', a, q, η') satisfying that there exists η such that $(q, a, q', \eta) \in \delta$, and $\eta'(r_i^{(1)}) = \eta(r_i)$ for each $i \in [k]$,
– and $\boldsymbol{t} = (r_1^{(1)}, \cdots, r_k^{(1)})$.

Note that $\mathscr{L}(\mathcal{A}^{(r)}) = \{(w^{(r)}, \boldsymbol{n}) \mid (w, \boldsymbol{n}) \in \mathscr{L}(\mathcal{A})\}$, and the size of $(\mathcal{A}^{(r)}, \boldsymbol{t})$ is $O(|\mathcal{A}|)$.

$\mathsf{substring}_R^{-1}(L)$. A CEFA representation of $\mathsf{substring}_R^{-1}(L)$ is given by $(\mathcal{B}, \boldsymbol{t})$, where

– $\mathcal{B} = (Q', \Sigma, R', \delta', I', F')$ such that
  • $Q' = Q \times \{p_0, p_1, p_2\}$, (intuitively, p_0, p_1, and p_2 denote that the current position is before the starting position, between the starting position and ending position, and after the ending position respectively)
  • $R' = \left(r'_{1,1}, r'_{1,2}, r_1^{(1)}, \cdots, r_k^{(1)}\right)$, (intuitively, $r'_{1,1}$ denotes the starting position, and $r'_{1,2}$ denotes the length of the substring)
  • $I' = I \times \{p_0\}$, $F' = F' \times \{p_2\} \cup (I \cap F) \times \{p_0\}$,
  • and δ' comprises
    ∗ the tuples $((q, p_0), a, (q, p_0), \eta')$ such that $q \in I$, $a \in \Sigma$, and η' satisfies that $\eta'(r'_{1,1}) = 1$, and $\eta'(r'_{1,2}) = 0$, and $\eta'(r_j^{(1)}) = 0$ for each $j \in [k]$,
    ∗ the tuples $((q, p_0), a, (q', p_1), \eta')$ such that $q \in I$ and there exists η satisfying that $(q, a, q', \eta) \in \delta$, moreover, $\eta'(r'_{1,1}) = 0$ (recall that the positions of strings start at 0), $\eta'(r'_{1,2}) = 1$, and $\eta'(r_j^{(1)}) = \eta(r_j)$ for each $j \in [k]$,

    * the tuples $((q, p_0), a, (q', p_2), \eta')$ such that $q \in I$ and there exists η satisfying that $(q, a, q', \eta) \in \delta$, moreover, $q' \in F$, and $\eta'(r'_{1,1}) = 0$ (recall that the positions of strings start at 0), $\eta'(r'_{1,2}) = 1$, and $\eta'(r^{(1)}_j) = \eta(r_j)$ for each $j \in [k]$,
    * the tuples $((q, p_1), a, (q', p_1), \eta')$ such that there exists η satisfying that $(q, a, q', \eta) \in \delta$, $\eta'(r'_{1,1}) = 0$, and $\eta'(r'_{1,2}) = 1$, and $\eta'(r^{(1)}_j) = \eta(r_j)$ for each $j \in [k]$,
    * the tuples $((q, p_1), a, (q', p_2), \eta')$ such that $q' \in F$, and there exists η satisfying that $(q, a, q', \eta) \in \delta$, moreover, $\eta'(r'_{1,1}) = 0$, $\eta'(r'_{1,2}) = 1$, and $\eta'(r^{(1)}_j) = \eta(r_j)$ for each $j \in [k]$,
    * the tuples $((q, p_2), a, (q, p_2), \eta')$ such that $q \in F$, $\eta'(r'_{1,1}) = 0$, and $\eta'(r'_{1,2}) = 0$, and $\eta'(r^{(1)}_j) = 0$ for each $j \in [k]$,
- $\boldsymbol{t} = (r^{(1)}_1, \cdots, r^{(1)}_k)$.

Note that the size of $(\mathcal{B}, \boldsymbol{t})$ is $O(|\mathcal{A}|)$.

$(\mathscr{T}(\mathcal{T}))^{-1}_R(L)$. Suppose $\mathcal{T} = (Q', \Sigma, \delta', I', F')$. Then a CEFA representation of $(\mathscr{T}(\mathcal{T}))^{-1}_R(L)$ is given by $(\mathcal{B}, \boldsymbol{t})$, where

- $\mathcal{B}$ simulates the run of $\mathcal{T}$ on the input string, meanwhile, it simulates the run of $\mathcal{A}$ on the output string of $\mathcal{T}$, formally, $\mathcal{B} = (Q' \times Q, \Sigma, R^{(1)}, \delta'', I' \times I, F' \times F)$ such that
  - $R^{(1)} = (r^{(1)}_1, \cdots, r^{(1)}_k)$, and
  - δ'' comprises the tuples $((q'_1, q_1), a, (q'_2, q_2), \eta')$ satisfying one of the following conditions,
    * there exist $u = a_1 \cdots a_n \in \Sigma^+$ and a transition sequence $p_0 \xrightarrow[\delta]{a_1, \eta_1} p_2 \cdots p_{n-1} \xrightarrow[\delta]{a_n, \eta_n} p_n$ in $\mathcal{A}$ such that $(q'_1, a, q'_2, u) \in \delta'$, $p_0 = q_1$, $p_n = q_2$, and for each $j \in [k]$, $\eta'(r^{(1)}_j) = \eta_1(r_j) + \cdots + \eta_n(r_j)$,
    * $(q'_1, a, q'_2, \varepsilon) \in \delta'$, $q_1 = q_2$, and $\eta'(r^{(1)}_j) = 0$ for each $j \in [k]$,
- $\boldsymbol{t} = (r^{(1)}_1, \cdots, r^{(1)}_k)$.

Note that the number of transitions of $\mathcal{B}$ can be exponential in the worst case, since it summarises the updates of cost registers of $\mathcal{A}$ on the output strings of the transitions of $\mathcal{T}$. More precisely, let

- ℓ be the maximum length of the output strings of transitions of $\mathcal{T}$,
- N be the maximum number of transitions between a given pair of states of $\mathcal{A}$, and
- C be the maximum absolute value of the integer constants occurring in $\mathcal{A}$,

then $|\delta''|$, the cardinality of δ'', is bounded by $|\delta'| \times |Q|^2 \times N^\ell$, and the integer constants occurring in each transition of δ'' are bounded by ℓC. Therefore, the size of $(\mathcal{B}, \boldsymbol{t})$ is

$$O(|\delta'| \times |Q|^2 \times N^\ell \times k \log_2(\ell C)).$$

Since $|\delta'|, \ell \leq |\mathcal{T}|$, $|Q|, N, k \leq |\mathcal{A}|$, and $C \leq 2^{|\mathcal{A}|}$, we deduce that the size of $(\mathcal{B}, \boldsymbol{t})$ is $O(|\mathcal{T}| \times |\mathcal{A}|^2 \times |\mathcal{A}|^{|\mathcal{T}|} \times |\mathcal{A}|^2 \log_2(|\mathcal{T}|)) = |\mathcal{A}|^{O(|\mathcal{T}|)} |\mathcal{T}| \log_2(|\mathcal{T}|)$.

$(\mathsf{replaceAll}_{e,u})_R^{-1}(L)$. From the result in [18], we know that a NFT $\mathcal{T}_{e,u} = (Q', \Sigma, \delta', I', F')$ can be constructed to capture $\mathsf{replaceAll}_{e,u}$. Moreover,

- $|Q'|$, as well as $|\delta'|$, is $2^{O(|e|)}$,
- ℓ, the maximum length of the output strings of transitions of $\mathcal{T}_{e,u}$, is $|u|$.

Then a CEFA representation of $(\mathsf{replaceAll}_{e,u})_R^{-1}(L)$ can be constructed as that of $(\mathscr{T}(\mathcal{T}_{e,u}))_R^{-1}(L)$. Let N denote the maximum number of transitions between a given pair of states of $\mathcal{A}$, and C be the maximum absolute value of the integer constants occurring in $\mathcal{A}$, which is bounded by $2^{|\mathcal{A}|}$. Then the CEFA representation of $(\mathsf{replaceAll}_{e,u})_R^{-1}(L)$ is of size

$$O(|\delta'| \times |Q|^2 \times N^\ell \times k \log_2(\ell C)) = 2^{O(|e|)}|\mathcal{A}|^2|\mathcal{A}|^{|u|}|\mathcal{A}|^2 \log_2 |u| = 2^{O(|e|)}|\mathcal{A}|^{O(|u|)}.$$

according to the aforementioned discussion for NFTs. □

C Proof of Proposition 2

Proposition 2. *The* $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ *problem is* PSPACE*-complete.*

Proof. The lower bound follows from the PSPACE-hardness of the intersection problem of NFAs.

For the upper bound, let $\{\mathcal{A}_i^j\}_{i\in I, j\in J_i}$ be a family of CEFAs each of which carries a vector of registers R_i^j and ϕ be a quantifier-free LIA formula such that R_i^j are pairwise disjoint and the variables of ϕ are from $R' := \bigcup_{i,j} R_i^j$.

First, we observe that we can focus on *monotonic CEFAs* where the cost registers are monotone in the sense that their values are non-decreasing during the course of execution. In other words, they can only be updated with natural number (as opposed to general integer) constants. This observation is justified by the following reduction.

For each register $r \in R_j^i$, we introduce two registers r^+, r^-. Let $(R_j^i)^\pm$ denote the vector of registers by replacing each $r \in R_j^i$ with (r^+, r^-). Intuitively, for each $r \in R_j^i$, the updates of r in $\mathcal{A}_i^j$ are split into non-negative ones and negative ones, with the former stored in r^+ and the latter in r^-. Suppose $(R')^\pm = \bigcup_{i,j}(R_i^j)^\pm$. Then we construct monotonic CEFAs $(\mathcal{B}_i^j)_{i\in I, j\in J_i}$ and an LIA formula $\phi^\pm$ such that

> there are an assignment function $\theta : R' \to \mathbb{Z}$ and strings $(w_i)_{i\in I}$ such that $\phi[\theta(R')/R']$ holds and $(w_i, \theta(R_i^j)) \in \mathscr{L}(\mathcal{A}_i^j)$ for every $i \in I$ and $j \in J_i$
>
> if and only if
>
> there are an assignment function $\theta^\pm : (R')^\pm \to \mathbb{N}$ and strings $(w_i)_{i\in I}$ such that $\phi^\pm[\theta^\pm((R')^\pm)/(R')^\pm]$ holds and $(w_i, \theta^\pm((R_i^j)^\pm)) \in \mathscr{L}(\mathcal{B}_i^j)$ for every $i \in I$ and $j \in J_i$.

For $i \in I$ and $j \in J_i$, the CEFA $\mathcal{B}_i^j$ is obtained from $\mathcal{A}_i^j$ by replacing each transition (q, a, q', η) in $\mathcal{A}_i^j$ by the transition (q, a, q', η') such that for each $r \in R_j^j$,

$$\eta'(r^+) = \begin{cases} \eta(r), & \text{if } \eta(r) \geq 0 \\ 0 & \text{otherwise} \end{cases}, \eta'(r^-) = \begin{cases} 0, & \text{if } \eta(r) \geq 0 \\ -\eta(r) & \text{otherwise} \end{cases}.$$

In addition, $\phi^{\pm}$ is obtained from ϕ by replacing each $r \in R'$ with $r^{+} - r^{-}$.

It remains to prove the $\mathrm{SAT}_{\mathrm{CEFA}}[\mathrm{LIA}]$ problem for monotonic CEFAs is in PSPACE, namely,

> given a family of *monotonic* CEFAs $\{\mathcal{A}_i^j\}_{i \in I, j \in J_i}$ each of which carries a vector of registers R_i^j and a quantifier-free LIA formula ϕ such that R_i^j are pairwise disjoint, and the variables of ϕ are from $R' = \bigcup_{i,j} R_i^j$, deciding whether there are an assignment function $\theta : R' \to \mathbb{N}$ and strings $(w_i)_{i \in I}$ such that $\phi[\theta(R')/R']$ holds and $(w_i, \theta(R_i^j)) \in \mathscr{L}(\mathcal{A}_i^j)$ for every $i \in I$ and $j \in J_i$ is in PSPACE.

We use Proposition 16 in [29] to show the result. Proposition 16 in [29] mainly considered monotonic counter machines, which can be seen as monotonic CEFAs where each transition contains no alphabet symbol, and $\eta(r) \in \{0, 1\}$ for the update function η therein.

For each $i \in I$ and $j \in J_i$, let $(\mathcal{A}')_i^j$ be the monotonic counter machine obtained from $\mathcal{A}_i^j$ by the following two-step procedure:

1. [Remove the alphabet symbols]: Remove alphabet symbols a in each transition (q, a, q', η) of $\mathcal{A}_i^j$.
2. [From binary encoding to unary encoding]: Replace each transition (q, q', η) such that $\ell = \max_{r \in R_i^j} \eta(r) > 1$ with a sequence of transitions $(q, p_1, \eta'_1), \cdots, (p_{\ell-1}, q', \eta'_\ell)$, where $p_1, \cdots, p_{\ell-1}$ are the freshly introduced states, moreover, $\eta'_j(r) = 1$ if $\eta(r) \geq j$, and $\eta'_j(r) = 0$ otherwise.

According to Proposition 16 in [29], we have the following property.

> Given a family of monotonic counter machines $\{C_i\}_{i \in I}$ each of which carries a vector of counters R_i and a quantifier-free LIA formula ϕ such that R_i are pairwise disjoint, and the variables of ϕ are from $R' = \bigcup_i R_i$. If there is an assignment function $\theta : R' \to \mathbb{N}$ such that $\phi[\theta(R')/R']$ holds and $\theta(R_i)$ is a reachable valuation of counters in C_i for every $i \in I$, then there are desired θ such that for each $i \in I$ and $r \in R_i$, $\theta(r)$ is at most polynomial in the number of states in C_i, exponential in $|R_i|$, and exponential in $|\phi|$.

For each $i \in I$, let C_i be the product of monotonic counter machines $(\mathcal{A}')_i^j$ for $j \in J_i$. From the fact that the number of states of $(\mathcal{A}')_i^j$ is at most the product of the number of transitions of $\mathcal{A}_i^j$ and $B_{\mathcal{A}_i^j}$ (where $B_{\mathcal{A}_i^j}$ denotes the maximum natural number constants $\eta(r)$ in $\mathcal{A}_i^j$), we deduce the following,

> if there are an assignment function $\theta : R' \to \mathbb{N}$ and strings $(w_i)_{i \in I}$ such that $\phi[\theta(R')/R']$ holds and $(w_i, \theta(R_i^j)) \in \mathscr{L}(\mathcal{A}_i^j)$ for every $i \in I$ and $j \in J_i$, then there are desired θ and $(w_i)_{i \in I}$ such that for each $i \in I$ and $r \in \bigcup_{j \in J_i} R_i^j$, $\theta(r)$ is at most polynomial in the product of the number of transitions in $\mathcal{A}_i^j$ and $B_{\mathcal{A}_i^j}$ for $j \in J_i$, exponential in $\left|\bigcup_{j \in J_i} R_i^j\right|$, and exponential in $|\phi|$.

Since the values of all the registers in $\mathcal{A}_i^j$ for $i \in I$ and $j \in J_i$ can be assumed to be at most exponential, and thus their binary encodings can be stored in polynomial space, one can nondeterministically guess the strings $(w_i)_{i \in I}$, and for each $i \in I$ and $j \in J_i$, simulate the runs of CEFAs $\mathcal{A}_i^j$ on w_i, and finally evaluate ϕ with the register values after all $\mathcal{A}_i^j$ accept, in polynomial space. From Savitch's theorem [31], we conclude that the $\text{SAT}_{\text{CEFA}}[\text{LIA}]$ problem for monotonic CEFAs is in PSPACE. This concludes the proof of the proposition. □

D Example of `urlXssSanitise(url)` for the decision procedure

Consider the program S associated with `urlXssSanitise(url)` in Section 2. We show how to decide its path feasibility.

Step I. Vacuous since S contains only atomic assertions already.

Step II. Nondeterministically choose to replace $\mathsf{indexOf}_{\#}(\texttt{url1}, 0)$ with -1 and add $\mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\overline{\Sigma^* \# \Sigma^*}}\right)$ to S.

Step III. Replace $\mathsf{length}(\texttt{url1})$ with i'_1 and add $\mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\mathsf{len}}[i'_1/r_1]\right)$ to S, moreover, replace $\mathsf{indexOf}_{?}(\texttt{url1}, 0)$ with i'_3 and add $\mathsf{assert}\left(0 = i'_2\right); \mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\mathsf{indexOf}}[i'_2/r_1, i'_3/r_2]\right)$ to S, where i'_1, i'_2, i'_3 are fresh integer variables. Then we get the following program (still denoted by S),

$$\begin{array}{l}
\mathsf{assert}(\texttt{prothostpath} \in \mathcal{A}_\varepsilon); \mathsf{assert}(\texttt{querfrag} \in \mathcal{A}_\varepsilon); \texttt{url1} := \mathcal{T}_{\text{trim}}(\texttt{url}); \\
\mathsf{assert}\left(\texttt{qmarkpos} = i'_3\right); \mathsf{assert}(\texttt{sharppos} = -1); \mathsf{assert}(\texttt{qmarkpos} \geq 0); \\
\texttt{prothostpath1} := \mathsf{substring}(\texttt{url1}, 0, \texttt{qmarkpos}); \\
\texttt{querfrag1} := \mathsf{substring}(\texttt{url1}, \texttt{qmarkpos}, i'_1 - \texttt{qmarkpos}); \\
\texttt{querfrag2} := \mathsf{replaceAll}_{\texttt{script}, \varepsilon}(\texttt{querfrag1}); \\
\texttt{url2} := \texttt{prothostpath1} \cdot \texttt{querfrag2}; \mathsf{assert}(\texttt{querfrag2} \in \mathcal{A}_{\Sigma^* \texttt{script} \Sigma^*}); \\
\mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\overline{\Sigma^* \# \Sigma^*}}\right); \mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\mathsf{len}}[i'_1/r_1]\right); \\
\mathsf{assert}\left(0 = i'_2\right); \mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\mathsf{indexOf}}[i'_2/r_1, i'_3/r_2]\right).
\end{array}$$

Step IV. Since there is no CEFA constraints for `url2`, removing the last assignment statement of S, i.e. $\texttt{url2} := \texttt{prothostpath1} \cdot \texttt{querfrag2}$, is easy and in this case we add no statements to S. After this, $\texttt{querfrag2} := \mathsf{replaceAll}_{\texttt{script}, \varepsilon}(\texttt{querfrag1})$ becomes the last assignment statement. Suppose $\mathcal{A}' = (Q', \Sigma, \delta', I', F')$ is an NFA representing $(\mathsf{replaceAll}_{\texttt{script}, \varepsilon})^{-1}_{\emptyset}(\mathscr{L}(\mathcal{A}_{\Sigma^* \texttt{script} \Sigma^*}))$, namely, the pre-image of $\mathscr{L}(\mathcal{A}_{\Sigma^* \texttt{script} \Sigma^*})$ under $\mathsf{replaceAll}_{\texttt{script}, \varepsilon}$. Then we remove this assignment statement and add $\mathsf{assert}(\texttt{querfrag1} \in \mathcal{A}')$, resulting into the following program

$$\begin{array}{l}
\mathsf{assert}(\texttt{prothostpath} \in \mathcal{A}_\varepsilon); \mathsf{assert}(\texttt{querfrag} \in \mathcal{A}_\varepsilon); \texttt{url1} := \mathcal{T}_{\text{trim}}(\texttt{url}); \\
\mathsf{assert}\left(\texttt{qmarkpos} = i'_3\right); \mathsf{assert}(\texttt{sharppos} = -1); \mathsf{assert}(\texttt{qmarkpos} \geq 0); \\
\texttt{prothostpath1} := \mathsf{substring}(\texttt{url1}, 0, \texttt{qmarkpos}); \\
\texttt{querfrag1} := \mathsf{substring}(\texttt{url1}, \texttt{qmarkpos}, i'_1 - \texttt{qmarkpos}); \\
\mathsf{assert}(\texttt{querfrag2} \in \mathcal{A}_{\Sigma^* \texttt{script} \Sigma^*}); \mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\overline{\Sigma^* \# \Sigma^*}}\right); \\
\mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\mathsf{len}}[i'_1/r_1]\right); \mathsf{assert}\left(0 = i'_2\right); \\
\mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\mathsf{indexOf}}[i'_2/r_1, i'_3/r_2]\right); \mathsf{assert}(\texttt{querfrag1} \in \mathcal{A}').
\end{array}$$

From Example 5, we know that $\mathsf{substring}_{\emptyset}^{-1}(\mathscr{L}(\mathcal{A}'))$ can be represented by some CEFA $\mathcal{B} = (Q'', R'', \delta'', I'', F'')$ with $Q'' = Q' \times \{p_0, p_1, p_2\}$ and $R'' = (r'_{1,1}, r'_{1,2})$ (where $r'_{1,1}$ and $r'_{1,2}$ are fresh integer variables). Then we remove $\texttt{querfrag1} := \mathsf{substring}(\texttt{url1}, \texttt{qmarkpos}, i'_1 - \texttt{qmarkpos})$, add $\mathsf{assert}(\texttt{url1} \in \mathcal{B})$; $\mathsf{assert}\left(\texttt{r}'_{1,1} = \texttt{qmarkpos}\right)$; $\mathsf{assert}\left(r'_{1,2} = i'_1 - \texttt{qmarkpos}\right)$, and get the following program

$$
\begin{array}{l}
\mathsf{assert}(\texttt{prothostpath} \in \mathcal{A}_{\varepsilon})\,;\mathsf{assert}(\texttt{querfrag} \in \mathcal{A}_{\varepsilon})\,;\texttt{url1} := \mathcal{T}_{\mathrm{trim}}(\texttt{url});\\
\mathsf{assert}\left(\texttt{qmarkpos} = i'_3\right);\mathsf{assert}(\texttt{sharppos} = -1)\,;\mathsf{assert}(\texttt{qmarkpos} \geq 0)\,;\\
\texttt{prothostpath1} := \mathsf{substring}(\texttt{url1}, 0, \texttt{qmarkpos});\\
\mathsf{assert}(\texttt{querfrag2} \in \mathcal{A}_{\Sigma^*\texttt{script}\Sigma^*})\,;\mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\overline{\Sigma^*\#\Sigma^*}}\right);\\
\mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\mathsf{len}}[i'_1/r_1]\right);\mathsf{assert}\left(0 = i'_2\right);\\
\mathsf{assert}\left(\texttt{url1} \in \mathcal{A}_{\mathsf{indexOf}}[i'_2/r_1, i'_3/r_2]\right);\mathsf{assert}(\texttt{querfrag1} \in \mathcal{A}')\,;\\
\mathsf{assert}(\texttt{url1} \in \mathcal{B})\,;\mathsf{assert}\left(\texttt{r}'_{1,1} = \texttt{qmarkpos}\right);\mathsf{assert}\left(r'_{1,2} = i'_1 - \texttt{qmarkpos}\right).
\end{array}
$$

We can continue the process until the problem contains no assignment statement.
Step V. Straightforward by utilising Proposition 2.

E Algorithms for case splits in the semantics of $\mathsf{indexOf}_v$ and $\mathsf{substring}$

Algorithm 3: *indexofCaseSplit* for case splits in the semantics of $\mathsf{indexOf}_v$

```
Input: active: set of CEFA constraints, arith: arithmetic constraint, funApps: acyclic set
       of assignment statements, and $(\mathcal{I}_l)_{l\in[5]}$: subsets of $\mathsf{indexOf}_v(x,i)$ string terms
Result: (active, arith, funApps)

1  for each $\mathsf{indexOf}_v(x,i) \in \mathcal{I}_1$ do
2      $arith \leftarrow arith[\mathsf{indexOf}_v(x,0)/\mathsf{indexOf}_v(x,i)] \wedge i < 0$;
3  for each $\mathsf{indexOf}_v(x,i) \in \mathcal{I}_2$ do
4      $active \leftarrow active \cup \{x \in \mathcal{A}_{\overline{\Sigma^* v \Sigma^*}}\}$;
5      $arith \leftarrow arith[-1/\mathsf{indexOf}_v(x,i)] \wedge i < 0$;
6  for each $\mathsf{indexOf}_v(x,i) \in \mathcal{I}_3$ do
7      $arith \leftarrow arith[-1/\mathsf{indexOf}_v(x,i)] \wedge i \geq \mathsf{length}(x)$;
8  for each $\mathsf{indexOf}_v(x,i) \in \mathcal{I}_4$ do
9      $arith \leftarrow arith[-1/\mathsf{indexOf}_v(x,i)] \wedge i \geq 0 \wedge i < \mathsf{length}(x)$;
10 for each $\mathsf{indexOf}_v(x,i) \in \mathcal{I}_5$ do
11     choose fresh variables $y$ and $j$;
12     $active \leftarrow active \cup \{y \in \mathcal{A}_{\overline{\Sigma^* v \Sigma^*}}\}$;
13     $arith \leftarrow arith[-1/\mathsf{indexOf}_v(x,i)] \wedge i \geq 0 \wedge i < \mathsf{length}(x) \wedge j = \mathsf{length}(x) - i$;
14     $funApps \leftarrow funApps \cup \{y := \mathsf{substring}(x,i,j)\}$;
```

Algorithm 4: *substringCaseSplit* for case splits in the semantics of substring

Input: *active*: set of CEFA constraints, *arith*: arithmetic constraint, *funApps*: acyclic set of assignment statements, and $(\mathcal{I}_l)_{l\in[5]}$: subsets of $\mathsf{indexOf}_v(x, i)$ string terms

Result: $(\mathit{active}, \mathit{arith}, \mathit{funApps})$

```
1  for each $y := \mathsf{substring}(x, i, j) \in \mathcal{J}_1$ do
2      $\mathit{arith} \leftarrow \mathit{arith} \wedge i \geq 0 \wedge i + j \leq \mathsf{length}(x)$;
3  for each $y := \mathsf{substring}(x, i, j) \in \mathcal{J}_2$ do
4      choose a fresh integer variable $i'$;
5      $\mathit{arith} \leftarrow \mathit{arith} \wedge i \geq 0 \wedge i \leq \mathsf{length}(x) \wedge i + j > \mathsf{length}(x) \wedge i' = \mathsf{length}(x) - i$;
6      $\mathit{funApps} \leftarrow \mathit{funApps}[y := \mathsf{substring}(x, i, i')/y := \mathsf{substring}(x, i, j)]$;
7  for each $y := \mathsf{substring}(x, i, j) \in \mathcal{J}_3$ do
8      $\mathit{arith} \leftarrow \mathit{arith} \wedge i < 0$;
9      $\mathit{active} \leftarrow \mathit{active} \cup \{y \in \mathcal{A}_\varepsilon\}$;
10     $\mathit{funApps} \leftarrow \mathit{funApps} \setminus \{y := \mathsf{substring}(x, i, j)\}$;
```