What Is Decidable about String Constraints with the ReplaceAll Function

Taolue Chen[1], Yan Chen[2,3], Matthew Hague[4], Anthony W. Lin[5] and Zhilin Wu[2]

[1] Department of Computer Science and Information Systems, Birkbeck, University of London
[2] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[3] University of Chinese Academy of Sciences
[4] Royal Holloway, University of London
[5] Department of Computer Science, University of Oxford

Abstract

The theory of strings with concatenation has been widely argued as the basis of constraint solving for verifying string-manipulating programs. However, this theory is far from adequate for expressing many string constraints that are also needed in practice; for example, the use of regular constraints (pattern matching against a regular expression), and the string-replace function (replacing either the first occurrence or all occurrences of a "pattern" string constant/variable/regular expression by a "replacement" string constant/variable), among many others. Both regular constraints and the string-replace function are crucial for such applications as analysis of JavaScript (or more generally HTML5 applications) against cross-site scripting (XSS) vulnerabilities, which motivates us to consider a richer class of string constraints. The importance of the string-replace function (especially the replace-all facility) is increasingly recognised, which can be witnessed by the incorporation of the function in the input languages of several string constraint solvers.

Recently, it was shown that any theory of strings containing the string-replace function (even the most restricted version where pattern/replacement strings are both constant strings) becomes undecidable if we do not impose some kind of straight-line (aka acyclicity) restriction on the formulas. Despite this, the straight-line restriction is still practically sensible since this condition is typically met by string constraints that are generated by symbolic execution. In this paper, we provide the first systematic study of straight-line string constraints with the string-replace function and the regular constraints as the basic operations. We show that a large class of such constraints (i.e. when only a constant string or a regular expression is permitted in the pattern) is decidable. We note that the string-replace function, even under this restriction, is sufficiently powerful for expressing the concatenation operator and much more (e.g. extensions of regular expressions with string variables). This gives us the most expressive decidable logic containing concatenation, replace, and regular constraints under the same umbrella. Our decision procedure for the straight-line fragment follows an automata-theoretic approach, and is modular in the sense that the string-replace terms are removed one by one to generate more and more regular constraints, which can then be discharged by the state-of-the-art string constraint solvers. We also show that this fragment is, in a way, a maximal decidable subclass of the straight-line fragment with string-replace and regular constraints. To this end, we show undecidability results for the following two extensions: (1) variables are permitted in the pattern parameter of the replace function, (2) length constraints are permitted.

1 Introduction

The problem of automatically solving string constraints (aka satisfiability of logical theories over strings) has recently witnessed renewed interests [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?] because of important applications in the analysis of string-manipulating programs. For example, program analysis techniques like symbolic execution [?, ?, ?, ?] would systematically explore executions in a program and collect symbolic path constraints, which could then be solved using a constraint solver and used to determine which location in the program to continue exploring. To successfully apply a constraint solver in this instance, it is crucial that the constraint language precisely models the data types in the program, along with the data-type operations used. In the context of string-manipulating programs, this could include concatenation, regular constraints (i.e. pattern matching against a regular expression), string-length functions, and the string-replace functions, among many others.

Perhaps the most well-known theory of strings for such applications as the analysis of string-manipulating programs is the theory of strings with concatenation (aka *word equations*), whose decidability was shown by Makanin [?] in 1977 after it was open for many years. More importantly, this theory remains decidable even when regular constraints are incorporated into the language [?]. However, whether adding the string-length function preserves the decidability remains a long-standing open problem [?, ?].

Another important string operation—especially in popular scripting languages like Python, JavaScript, and PHP—is the *string-replace function*, which may be used to replace either the *first* occurrence or *all* occurrences of a string (a string constant/variable, or a regular expression) by another string (a string constant/variable). The replace function (especially the replace-all functionality) is omnipresent in HTML5 applications [?, ?, ?]. For example, a standard industry defense against cross-site scripting (XSS) vulnerabilities includes sanitising untrusted strings before adding them into the DOM (Document Object Model) or the HTML document. This is typically done by various metacharacter-escaping mechanisms (see, for instance, [?, ?, ?]). An example of such a mechanism is backslash-escape, which replaces *every occurrence* of quotes and double-quotes (i.e. `'` and `"`) in the string by `\'` and `\"`. In addition to sanitisers, common JavaScript functionalities like `document.write()` and `innerHTML` apply an *implicit browser transduction* — which decodes HTML codes (e.g. `&#39;` is replaced by `'`) in the input string — before inserting the input string into the DOM. Both of these examples can be expressed by (perhaps multiple) applications of the string-replace function. Moreover, although these examples replace constants by constants, the popularity of template systems such as Mustache [?] and Closure Templates [?] demonstrate the need for replacements involving variables. Using Mustache, a web-developer, for example, may define an HTML fragment with placeholders that is instantiated with user data during the construction of the delivered page.

Example 1.1. *We give a simple example demonstrating a (naive) XSS vulnerability to illustrate the use of string-replace functions. Consider the HTML fragment below.*

```
<h1> User <span onMouseOver="popupText('{{bio}}')">{{userName}}</span> </h1>
```

This HTML fragment is a template as might be used with systems such as Mustache to display a user on a webpage. For each user that is to be displayed – with their username and biography stored in variables user *and* bio *respectively – the string* `{{userName}}` *will be replaced by* user *and the string* `{{bio}}` *will be replaced by* bio*. For example, a user* `Amelia` *with biography* `Amelia was born in 1979...` *would result in the HTML below.*

```
<h1> User
     <span onMouseOver="popupText('Amelia was born in 1979...')">
         Amelia </span> </h1>
```

This HTML would display `User Amelia`*, and, when the mouse is placed over* `Amelia`*, her biography would appear, thanks to the* `onMouseOver` *attribute in the* `span` *element.*

Unfortunately, this template could be insecure if the user biography is not adequately sanitised: A user could enter a malicious biography, such as `'); alert('Boo!'); alert('` *which would cause the following instantiation of the* `span` *element*[1]*.*

```
<span onMouseOver="popupText(''); alert('Boo!'); alert('')">
```

Now, when the mouse is placed over the user name, the malicious JavaScript `alert('Boo!')` *is executed.*

The presence of such malicious injections of code can be detected using string constraint solving and XSS attack patterns *given as regular expressions [?, ?, ?]. For our example, given an attack pattern P and template temp, we would generate the constraint*

$$x_1 = \mathsf{replaceAll}(temp, \texttt{\{\{userName\}\}}, user) \wedge x_2 = \mathsf{replaceAll}(x_1, \texttt{\{\{bio\}\}}, bio) \wedge x_2 \in P$$

which would detect if the HTML generated by instantiating the template is susceptible to the attack identified by P. □

In general, the string-replace function has three parameters, and in the current mainstream language such as Python and JavaScript, *all of the three parameters can be inserted as string variables*. As result, when we perform program analysis for, for instance, detecting security vulnerabilities as described above, one often obtains string constraints of the form $z = \mathsf{replaceAll}(x, p, y)$, where x, y are string constants/variables, and p is either a string constant/variable or a regular expression. Such a constraint means that z is obtained by replacing all occurrences of p in x with y. For convenience, we call x, p, y as the *subject*, the *pattern*, and the *replacement* parameters respectively.

The $\mathsf{replaceAll}$ function is a powerful string operation that goes beyond the expressiveness of concatenation. (On the contrary, as we will see later, concatenation can be expressed by the $\mathsf{replaceAll}$ function easily.) It was shown in a recent POPL paper [?] that any theory of strings containing the string-replace function (even the most restricted version where pattern/replacement strings are both constant strings) becomes undecidable if we do not impose some kind of *straight-line restriction*[2] on the formulas. Nonetheless, as already noted in [?], the straight-line restriction is reasonable since it is typically satisfied by constraints that are generated by symbolic execution, e.g., all constraints in the standard Kaluza benchmarks [?] with 50,000+ test cases generated by symbolic execution on JavaScript applications were noted in [?] to satisfy this condition. Intuitively, as elegantly described in [?], constraints from symbolic execution on string-manipulating programs can be viewed as the problem of path feasibility over loopless string-manipulating programs S with variable assignments and assertions, i.e., generated by the grammar

$$S ::= y := f(x_1, \ldots, x_n) \mid \mathbf{assert}(g(x_1, \ldots, x_n)) \mid S_1; S_2$$

where $f : (\Sigma^*)^n \to \Sigma^*$ and $g : (\Sigma^*)^n \to \{0, 1\}$ are some string functions. Straight-line programs with assertions can be obtained by turning such programs into a Static Single Assignment (SSA) form (i.e. introduce a new variable on the left hand side of each assignment). A partial decidability result can be deduced from [?] for the straight-line fragment of the theory of strings, where (1) f in the above grammar is either a concatenation of string constants and variables, or the $\mathsf{replaceAll}$ function where *the pattern and the replacement are both string constants*,

[1] Readers familiar with Mustache and Closure Templates may expect single quotes to be automatically escaped. However, we have tested our example with the latest versions of mustache.js [?] and Closure Templates [?] (as of July 2017) and observed that the exploit is not disarmed by their automatic escaping features.

[2] Similar notions that appear in the literature of string constraints (without replace) include acyclicity [?] and solved form [?]

and (2) g is a boolean combination of regular constraints. In fact, the decision procedure therein admits finite-state transducers, which subsume only the aforementioned simple form of the replaceAll function. The decidability boundary of the straight-line fragment involving the replaceAll function in its general form (e.g., when the replacement parameter is a variable) remains open.

Contribution. We investigate the decidability boundary of the theory SL[replaceAll] of strings involving the replaceAll function and regular constraints, with the straight-line restriction introduced in [**?**]. We provide a decidability result for a large fragment of SL[replaceAll], which is sufficiently powerful to express the concatenation operator. We show that this decidability result is in a sense maximal by showing that several important natural extensions of the logic result in undecidability. We detail these results below:

- If the pattern parameters of the replaceAll function are allowed to be variables, then the satisfiability of SL[replaceAll] is undecidable (cf. Proposition 4.1).
- If the pattern parameters of the replaceAll function are regular expressions, then the satisfiability of SL[replaceAll] is decidable and in EXPSPACE (cf. Theorem 4.2). In addition, we show that the satisfiability problem is PSPACE-complete for several cases that are meaningful in practice (cf. Corollary 4.7). This strictly generalises the decidability result in [**?**] of the straight-line fragment with concatenation, regular constraints, and the replaceAll function where patterns/replacement parameters are constant strings.
- If SL[replaceAll], where the pattern parameter of the replaceAll function is a constant letter, is extended with the string-length constraint, then satisfiability becomes undecidable again. In fact, this undecidability can be obtained with either integer constraints, character constraints, or constraints involving the IndexOf function (cf. Theorem 9.4 and Proposition 9.6).

Our decision procedure for SL[replaceAll] where the pattern parameters of the replaceAll function are regular expressions follows an automata-theoretic approach. The key idea can be illustrated as follows. Let us consider the simple formula $C \equiv x = \mathsf{replaceAll}(y, a, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$. Suppose that $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ are the nondeterministic finite state automata corresponding to e_1, e_2, e_3 respectively. We effectively eliminate the use of replaceAll by nondeterministically generating from $\mathcal{A}_1$ a new regular constraint $\mathcal{A}'_2$ for y as well as a new regular constraint $\mathcal{A}'_3$ for z. These constraints incorporate the effect of the replaceAll function (i.e. all regular constraints are on the "source" variables). Then, the satisfiability of C is turned into testing the nonemptiness of the intersection of $\mathcal{A}_2$ and $\mathcal{A}'_2$, as well as the nonemptiness of the intersection of $\mathcal{A}_3$ and $\mathcal{A}'_3$. When there are multiple occurrences of the replaceAll function, this process can be iterated. Our decision procedure enjoys the following advantages:

- It is automata-theoretic and built on clean automaton constructions, moreover, when the formula is satisfiable, a solution can be synthesised. For example, in the aforementioned XSS vulnerability detection example, one can synthesise the values of the variables *user* and *bio* for a potential attack.
- The decision procedure is modular in that the replaceAll terms are removed one by one to generate more and more regular constraints (emptiness of the intersection of regular constraints could be efficiently handled by state-of-the-art solvers like [**?**]).

- The decision procedure requires exponential space (thus double exponential time), but under assumptions that are reasonable in practice, the decision procedure uses only polynomial space, which is not worse than other string logics (which can encode the PSPACE-complete problem of checking emptiness of the intersection of regular constraints).

Organisation. This paper is organised as follows: Preliminaries are given in Section 2. The core string language is defined in Section 3. The main results of this paper are summarised in Section 4. The decision procedure is presented in Section 6-8, case by case. The extensions of the core string language are investigated in Section 9. The related work can be found in Section 10. The appendix contains missing proofs and additional examples.

2 Preliminaries

General Notation Let $\mathbb{Z}$ and $\mathbb{N}$ denote the set of integers and natural numbers respectively. For $k \in \mathbb{N}$, let $[k] = \{1, \cdots, k\}$. For a vector $\vec{x} = (x_1, \cdots, x_n)$, let $|\vec{x}|$ denote the length of $\vec{x}$ (i.e., n) and $\vec{x}[i]$ denote x_i for each $i \in [n]$.

Regular Languages Fix a finite *alphabet* Σ. Elements in Σ^* are called *strings*. Let ε denote the empty string and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. We will use $a, b, \cdots$ to denote letters from Σ and $u, v, w, \cdots$ to denote strings from Σ^*. For a string $u \in \Sigma^*$, let $|u|$ denote the *length* of u (in particular, $|\varepsilon| = 0$). A *position* of a nonempty string u of length n is a number $i \in [n]$ (Note that the first position is 1, instead of 0). In addition, for $i \in [|u|]$, let $u[i]$ denote the i-th letter of u. For two strings u_1, u_2, we use $u_1 \cdot u_2$ to denote the *concatenation* of u_1 and u_2, that is, the string v such that $|v| = |u_1| + |u_2|$ and for each $i \in [|u_1|]$, $v[i] = u_1[i]$ and for each $i \in |u_2|$, $v[|u_1| + i] = u_2[i]$. Let u, v be two strings. If $v = u \cdot v'$ for some string v', then u is said to be a *prefix* of 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, $\varepsilon^{-1}v = v$.

A *language* over Σ is a subset of Σ^*. We will use $L_1, L_2, \ldots$ to denote languages. For two languages L_1, L_2, we use $L_1 \cup L_2$ to denote the union of L_1 and L_2, and $L_1 \cdot L_2$ to 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 L^n, the *iteration* of L for n times, inductively as follows: $L^0 = \{\varepsilon\}$ and $L^n = L \cdot L^{n-1}$ for $n > 0$. We also use L^* to denote the iteration of L for arbitrarily many times, that is, $L^* = \bigcup\limits_{n \in \mathbb{N}} L^n$. Moreover, let $L^+ = \bigcup\limits_{n \in \mathbb{N} \setminus \{0\}} L^n$.

Definition 2.1 (Regular expressions RegExp)**.**

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

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* $e^+ \equiv e \circ e^*$*. Moreover, for* $\Gamma = \{a_1, \cdots, a_n\} \subseteq \Sigma$*, 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: $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(a) = \{a\}$, $\mathcal{L}(e_1 + e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$, $\mathcal{L}(e_1 \circ e_2) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$, $\mathcal{L}(e_1^*) = (\mathcal{L}(e_1))^*$. In addition, we use $|e|$ to denote the number of symbols occurring in e.

A *nondeterministic finite automaton* (NFA) $\mathcal{A}$ on Σ is a tuple (Q, δ, q_0, F), where Q is a finite set of *states*, $q_0 \in Q$ is the *initial* state, $F \subseteq Q$ is the set of *final* states, and $\delta \subseteq Q \times \Sigma \times Q$

is the *transition relation*. For a string $w = a_1 \ldots a_n$, a *run* of $\mathcal{A}$ on w is a state sequence $q_0 \ldots q_n$ such that for each $i \in [n]$, $(q_{i-1}, a_i, q_i) \in \delta$. A run $q_0 \ldots q_n$ is *accepting* if $q_n \in F$. A string w is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. We use $\mathcal{L}(\mathcal{A})$ to denote the language defined by $\mathcal{A}$, that is, the set of strings accepted by $\mathcal{A}$. We will use $\mathcal{A}, \mathcal{B}, \cdots$ to denote NFAs. For a string $w = a_1 \ldots a_n$, we also use the notation $q_1 \xrightarrow[\mathcal{A}]{w} q_{n+1}$ to denote the fact that there are $q_2, \ldots, q_n \in Q$ such that for each $i \in [n]$, $(q_i, a_i, q_{i+1}) \in \delta$. For an NFA $\mathcal{A} = (Q, \delta, q_0, F)$ and $q, q' \in Q$, we use $\mathcal{A}(q, q')$ to denote the NFA obtained from $\mathcal{A}$ by changing the initial state to q and the set of final states to $\{q'\}$. The *size* of an NFA $\mathcal{A} = (Q, \delta, q_0, F)$, denoted by $|\mathcal{A}|$, is defined as $|Q|$, the number of states. For convenience, we will also call an NFA without initial and final states, that is, a pair (Q, δ), as a *transition graph.*

It is well-known (e.g. see [?]) that regular expressions and NFAs are expressively equivalent, and generate precisely all *regular languages.* In particular, from a regular expression, an equivalent NFA can be constructed in linear time. Moreover, regular languages are closed under Boolean operations, i.e., union, intersection, and complementation. In particular, given two NFA $\mathcal{A}_1 = (Q_1, \delta_1, q_{0,1}, F_1)$ and $\mathcal{A}_2 = (Q_2, \delta_2, q_{0,2}, F_2)$ on Σ, the intersection $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ is recognised by the *product automaton* $\mathcal{A}_1 \times \mathcal{A}_2$ of $\mathcal{A}_1$ and $\mathcal{A}_2$ defined as $(Q_1 \times Q_2, \delta, (q_{0,1}, q_{0,2}), F_1 \times F_2)$, where δ comprises the transitions $((q_1, q_2), a, (q_1', q_2'))$ such that $(q_1, a, q_1') \in \delta_1$ and $(q_2, a, q_2') \in \delta_2$.

Graph-Theoretical Notation A DAG (*directed acyclic graph*) G is a finite directed graph (V, E) with no directed cycles, where V (resp. $E \subseteq V \times V$) is a set of vertices (resp. edges). Equivalently, a DAG is a directed graph that has a topological ordering, which is a sequence of the vertices such that every edge is directed from an earlier vertex to a later vertex in the sequence. An edge (v, v') in G is called an *incoming* edge of v' and an *outgoing* edge of v. If $(v, v') \in E$, then v' is called a *successor* of v and v is called a *predecessor* of v'. A *path* π in G is a sequence $v_0 e_1 v_1 \cdots v_{n-1} e_n v_n$ such that for each $i \in [n]$, we have $e_i = (v_{i-1}, v_i) \in E$. The *length* of the path π is the number n of edges in π. If there is a path from v to v' (resp. from v' to v) in G, then v' is said to be *reachable* (resp. *co-reachable*) from v in G. If v is reachable from v' in G, then v' is also called an *ancestor* of v in G. In addition, an edge (v', v'') is said to be reachable (resp. co-reachable) from v if v' is reachable from v (resp. v'' is co-reachable from v). The *in-degree* (resp. *out-degree*) of a vertex v is the number of incoming (resp. outgoing) edges of v. A *subgraph* G' of $G = (V, E)$ is a directed graph (V', E') with $V' \subseteq V$ and $E' \subseteq E$. Let G' be a subgraph of G. Then $G \setminus G'$ is the graph obtained from G by removing all the edges in G'.

Computational Complexity In this paper, we study not only decidability but also the complexity of string logics. In particular, we shall deal with the following computational complexity classes (see [?] for more details): PSPACE (problems solvable in polynomial space and thus in exponential time), and EXPSPACE (problems solvable in exponential space and thus in double exponential time). Verification problems that have complexity PSPACE or beyond (see [?] for a few examples) have substantially benefited from techniques such as symbolic model checking [?].

3 The core constraint language

In this section, we define a general string constraint language that supports concatenation, the replaceAll function, and regular constraints. Throughout this section, we fix an alphabet Σ.

3.1 Semantics of the replaceAll Function

To define the semantics of the replaceAll function, we note that the function encompasses three parameters: the first parameter is the *subject* string, the second parameter is a *pattern* that is a string or a regular expression, and the third parameter is the *replacement* string. When the pattern parameter is a string, the semantics is somehow self-explanatory. However, when it is a regular expression, there is no consensus on the semantics even for the mainstream programming languages such as Python and Javascript. This is particularly the case when interpreting the union (aka alternation) operator in regular expressions or performing a replaceAll with a pattern that matches ε. In this paper, we mainly focus on the semantics of *leftmost and longest matching*. Our handling of ε matches is consistent with our testing of the implementation in Python and the `sed` command with the `--posix` flag. We also assume union is commutative (e.g. $\mathsf{replaceAll}(aa, a + aa, b) = \mathsf{replaceAll}(aa, aa + a, b) = b$) as specified by POSIX, but often ignored in practice (where bb is a common result in the former case).

Definition 3.1. *Let u, v be two strings such that $v = v_1 u v_2$ for some v_1, v_2 and e be a regular expression. We say that u is the* leftmost and longest *matching of e in v if one of the following two conditions hold,*

- *case $\varepsilon \notin \mathcal{L}(e)$:*
  1. *leftmost: $u \in \mathcal{L}(e)$, and $(v_1')^{-1} v \notin \mathcal{L}(e \circ \Sigma^*)$ for every strict prefix v_1' of v_1,*
  2. *longest: for every nonempty prefix v_2' of v_2, $u \cdot v_2' \notin \mathcal{L}(e)$.*
- *case $\varepsilon \in \mathcal{L}(e)$:*
  1. *leftmost: $u \in \mathcal{L}(e)$, and $v_1 = \varepsilon$,*
  2. *longest: for every nonempty prefix v_2' of v_2, $u \cdot v_2' \notin \mathcal{L}(e)$.*

Example 3.2. *Let us first consider $\Sigma = \{0, 1\}$, $v = 1010101$, $v_1 = 1$, $u = 010$, $v_2 = 101$, and $e = 0^*01(0^* + 1^*)$. Then $v = v_1 u v_2$, and the leftmost and longest matching of e in v is u. This is because $u \in \mathcal{L}(e)$, $\varepsilon^{-1} v = v \notin \mathcal{L}(e \circ \Sigma^*)$ (notice that v_1 has only one strict prefix, i.e. ε), and none of $u1 = 0101$, $u10 = 01010$, and $u101 = 010101$ belong to $\mathcal{L}(e)$ (notice that v_2 has three nonempty prefixes, i.e. $1, 10, 101$). For another example, let us consider $\Sigma = \{a, b, c\}$, $v = baac$, $v_1 = \varepsilon$, $u = \varepsilon$, $v_2 = v$, and $e = a^*$. Then $v = v_1 u v_2$ and the leftmost and longest matching of e in v is u. This is because $u \in \mathcal{L}(e)$, $v_1 = \varepsilon$, and $b, ba, baa, baac \notin \mathcal{L}(e)$. On the other hand, similarly, one can verify that the leftmost and longest matching of $e = a^*$ in $v = aac$ is $u = aa$.*

Definition 3.3. *The semantics of $\mathsf{replaceAll}(u, e, v)$, where u, v are strings and e is a regular expression, is defined inductively as follows:*

- *if $u \notin \mathcal{L}(\Sigma^* \circ e \circ \Sigma^*)$, that is, u does* not *contain any substring from $\mathcal{L}(e)$, then $\mathsf{replaceAll}(u, e, v) = u$,*
- *otherwise,*
  - *if $\varepsilon \in \mathcal{L}(e)$ and u is the leftmost and longest matching of e in u, then $\mathsf{replaceAll}(u, e, v) = v$,*
  - *if $\varepsilon \in \mathcal{L}(e)$, $u = u_1 \cdot a \cdot u_2$, u_1 is the leftmost and longest matching of e in u, and $a \in \Sigma$, then $\mathsf{replaceAll}(u, e, v) = v \cdot a \cdot \mathsf{replaceAll}(u_2, e, v)$,*
  - *if $\varepsilon \notin \mathcal{L}(e)$, $u = u_1 \cdot u_2 \cdot u_3$, and u_2 is the leftmost and longest matching of e in u, then $\mathsf{replaceAll}(u, e, v) = u_1 \cdot v \cdot \mathsf{replaceAll}(u_3, e, v)$.*

Example 3.4. *At first,* $\mathsf{replaceAll}(abab, ab, d) = d\cdot\mathsf{replaceAll}(ab, ab, d) = dd\cdot\mathsf{replaceAll}(\epsilon, ab, d) = dd \cdot \varepsilon = dd$ *and* $\mathsf{replaceAll}(baac, a^+, b) = bbc$*. In addition,* $\mathsf{replaceAll}(aaaa, \text{“”}, d) = dadadadad$ *and* $\mathsf{replaceAll}(baac, a^*, b) = bbbcb$*. The argument for* $\mathsf{replaceAll}(baac, a^*, b) = bbbcb$ *proceeds as follows: The leftmost and longest matching of* a^* *in* $baac$ *is* $u_1 = \varepsilon$*, where* $baac = u_1 \cdot b \cdot u_2$ *and* $u_2 = aac$*. Then* $\mathsf{replaceAll}(baac, a^*, b) = b \cdot b \cdot \mathsf{replaceAll}(aac, a^*, b)$*. Since* aa *is the leftmost and longest matching of* a^* *in* aac*, we have* $\mathsf{replaceAll}(aac, a^*, b) = b \cdot c \cdot \mathsf{replaceAll}(\varepsilon, a^*, b) = bcb$*. Therefore, we get* $\mathsf{replaceAll}(baac, a^*, b) = bbbcb$*. (The readers are invited to test this in Python and* `sed`*.)*

3.2 Straight-Line String Constraints With the replaceAll Function

We consider the String data type Str, and assume a countable set of variables $x, y, z, \cdots$ of Str.

Definition 3.5 (Relational and regular constraints)**.** *Relational constraints and regular constraints are defined by the following rules,*

$$\begin{array}{rcll} s & \overset{def}{=} & x \mid u & \textit{(string terms)} \\ p & \overset{def}{=} & x \mid e & \textit{(pattern terms)} \\ \varphi & \overset{def}{=} & x = s \circ s \mid x = \mathsf{replaceAll}(s, p, s) \mid \varphi \wedge \varphi & \textit{(relational constraints)} \\ \psi & \overset{def}{=} & x \in e \mid \psi \wedge \psi & \textit{(regular constraints)} \end{array}$$

where x *is a string variable,* $u \in \Sigma^*$ *and* e *is a regular expression over* Σ*.*

For a formula φ (resp. ψ), let $\mathsf{Vars}(\varphi)$ (resp. $\mathsf{Vars}(\psi)$) denote the set of variables occurring in φ (resp. ψ). Given a relational constraint φ, a variable x is called a *source variable* of φ if φ *does not* contain a conjunct of the form $x = s_1 \circ s_2$ or $x = \mathsf{replaceAll}(-, -, -)$.

We then notice that, with the replaceAll function in its general form, the concatenation operation is in fact redundant.

Proposition 3.6. *The concatenation operation* ($\circ$) *can be simulated by the* replaceAll *function.*

Proof. It is sufficient to observe that a relational constraint $x = s_1 \circ s_2$ can be rewritten as

$$x' = \mathsf{replaceAll}(ab, a, s_1) \wedge x = \mathsf{replaceAll}(x', b, s_2),$$

where a, b are two fresh letters. □

In light of Proposition 3.6, in the sequel, we will *dispense the concatenation operator* mostly and focus on **the string constraints that involve the replaceAll function only**.

Another example to show the power of the replaceAll function is that it can simulate the extension of regular expressions with string variables, which is supported by the mainstream scripting languages like Python, Javascript, and PHP. For instance, $x \in y^*$ can be expressed by $x = \mathsf{replaceAll}(x', a, y) \wedge x' \in a^*$, where x' is a fresh variable and a is a fresh letter.

The generality of the constraint language makes it undecidable, even in very simple cases. To retain decidability, we follow [?] and focus on the “straight-line fragment” of the language. This straight-line fragment captures the structure of straight-line string-manipulating programs with the replaceAll string operation.

Definition 3.7 (Straight-line relational constraints)**.** *A relational constraint* φ *with the* replaceAll *function is straight-line, if* $\varphi \overset{def}{=} \bigwedge\limits_{1 \leq i \leq m} x_i = P_i$ *such that*

- *$x_1, \ldots, x_m$ are mutually distinct,*

- *for each $i \in [m]$, all the variables in P_i are either source variables, or variables from $\{x_1, \ldots, x_{i-1}\}$,*

Remark 3.8. *Checking whether a relational constraint φ is straight-line can be done in linear time.*

Definition 3.9 (Straight-line string constraints)**.** *A straight-line string constraint C with the* replaceAll *function (denoted by* SL[replaceAll]*) is defined as $\varphi \wedge \psi$, where*

- *φ is a straight-line relational constraint with the* replaceAll *function, and*

- *ψ is a regular constraint.*

Example 3.10. *The following string constraint belongs to* SL[replaceAll]*:*

$$C \equiv x_2 = \mathsf{replaceAll}(x_1, 0, y_1) \wedge x_3 = \mathsf{replaceAll}(x_2, 1, y_2) \wedge x_1 \in \{0,1\}^* \wedge y_1 \in 1^* \wedge y_2 \in 0^*.$$

4 The satisfiability problem

In this paper, we focus on the satisfiability problem of SL[replaceAll], which is formalised as follows.

Given an SL[replaceAll] constraint C, decide whether C is satisfiable.

To approach this problem, we identify several fragments of SL[replaceAll], depending on whether the pattern and the replacement parameters are constants or variables. We shall investigate extensively the satisfiability problem of the fragments of SL[replaceAll].

We begin with the case where the pattern parameters of the replaceAll terms are variables. It turns out that in this case the satisfiability problem of SL[replaceAll] is undecidable. The proof is by a reduction from Post's Correspondence Problem. Due to space constraints we relegate the proof to Appendix A.

Proposition 4.1. *The satisfiability problem of* SL[replaceAll] *is undecidable, if the pattern parameters of the* replaceAll *terms are allowed to be variables.*

In light of Proposition 4.1, we shall focus on the case that the pattern parameters of the replaceAll terms are constants, being a single letter, a constant string, or a regular expression. The main result of the paper is summarised as the following Theorem 4.2.

Theorem 4.2. *The satisfiability problem of* SL[replaceAll] *is decidable in EXPSPACE, if the pattern parameters of the* replaceAll *terms are regular expressions.*

The following three sections are devoted to the proof of Theorem 4.2.

- We start with the *single-letter* case that the pattern parameters of the replaceAll terms are single letters (Section 6),

- then consider the *constant-string* case that the pattern parameters of the replaceAll terms are constant strings (Section 7),

- and finally the *regular-expression* case that the pattern parameters of the $\mathsf{replaceAll}$ terms are regular expressions (Section 8).

We first introduce a graphical representation of $\mathsf{SL}[\mathsf{replaceAll}]$ formulae as follows.

Definition 4.3 (Dependency graph)**.** *Suppose $C = \varphi \wedge \psi$ is an $\mathsf{SL}[\mathsf{replaceAll}]$ formula where the pattern parameters of the $\mathsf{replaceAll}$ terms are regular expressions. Define the* dependency graph *of C as $G_C = (\mathsf{Vars}(\varphi), E_C)$, such that for each $i \in [m]$, if $x_i = \mathsf{replaceAll}(z, e_i, z')$, then $(x_i, (\mathsf{l}, e_i), z) \in E_C$ and $(x_i, (\mathsf{r}, e_i), z') \in E_C$. A final (resp. initial) vertex in G_C is a vertex in G_C without successors (resp. predecessors). The edges labelled by (l, e_i) and (r, e_i) are called the l-edges and r-edges respectively. The* depth *of G_C is the maximum length of the paths in G_C. In particular, if φ is empty, then the depth of G_C is zero.*

Note that G_C is a DAG where the out-degree of each vertex is two or zero.

Definition 4.4 (Diamond index and l-length)**.** *Let C be an $\mathsf{SL}[\mathsf{replaceAll}]$ formula and $G_C = (\mathsf{Vars}(\varphi), E_C)$ be its dependency graph. A* diamond *Δ in G_C is a pair of vertex-disjoint simple paths from z to z' for some $z, z' \in \mathsf{Vars}(\varphi)$. The vertices z and z' are called the* source *and* destination *vertex of the diamond respectively. A diamond Δ_2 with the source vertex z_2 and destination vertex z'_2 is said to be reachable from another diamond Δ_1 with the source vertex z_1 and destination vertex z'_1 if z_2 is reachable from z'_1 (possibly $z_2 = z'_1$). The* diamond index *of G_C, denoted by $\mathsf{Idx}_{\mathsf{dmd}}(G_C)$, is defined as the maximum length of the diamond sequences $\Delta_1 \cdots \Delta_n$ in G_C such that for each $i \in [n-1]$, Δ_{i+1} is reachable from Δ_i. The* l-length *of a path in G_C is the number of l-edges in the path. The l-length of G_C, denoted by $\mathsf{Len}_{\mathsf{lft}}(G_C)$, is the maximum l-length of paths in G_C.*

For each dependency graph G_C, since each diamond uses at least one l-edge, we know that $\mathsf{Idx}_{\mathsf{dmd}}(G_C) \leq \mathsf{Len}_{\mathsf{lft}}(G_C)$.

Proposition 4.5. *Let C be an $\mathsf{SL}[\mathsf{replaceAll}]$ formula and $G_C = (\mathsf{Vars}(\varphi), E_C)$ be its dependency graph. For each pair of distinct vertices z, z' in G_C, there are at most $(|\mathsf{Vars}(\varphi)||E_C|)^{O(\mathsf{Idx}_{\mathsf{dmd}}(G_C))}$ different paths from z to z'.*

It follows from Proposition 4.5 that for a class of $\mathsf{SL}[\mathsf{replaceAll}]$ formulae C such that $\mathsf{Idx}_{\mathsf{dmd}}(G_C)$ is bounded by a constant c, there are polynomially many different paths between each pair of distinct vertices in G_C.

Example 4.6. *Let G_C be the dependency graph illustrated in Figure 1. It is easy to see that $\mathsf{Idx}_{\mathsf{dmd}}(G_C)$ is 3. In addition, there are $2^3 = 8$ paths from x_1 to y_1. If we generalise G_C in Figure 1 to a dependency graph comprising n diamonds from x_1 to x_2, $\cdots$, from x_{n-1} to x_n, and from x_n to y_1 respectively, then the diamond index of the resulting dependency graph is n and there are 2^n paths from x_1 to y_1 in the graph.*

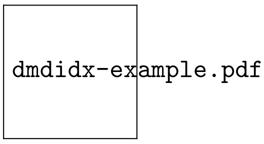

Figure 1: The diamond index and the number of paths in G_C

In Section 6–8, we will apply a refined analysis of the complexity of the decision procedures for proving Theorem 4.2 and get the following results.

Corollary 4.7. *The satisfiability problem is PSPACE-complete for the following fragments of* $\mathsf{SL}[\mathsf{replaceAll}]$*:*

- *the single-letter case, plus the condition that the diamond indices of the dependency graphs are bounded by a constant c,*
- *the constant-string case, plus the condition that the* l*-lengths of the dependency graphs are bounded by a constant c,*
- *the regular-expression case, plus the condition that the* l*-lengths of the dependency graphs are at most* 1*.*

Corollary 4.7 partially justifies our choice to present the decision procedures for the single-letter, constant-string, and regular-expression case separately. Intuitively, when the pattern parameters of the replaceAll terms become less restrictive, the decision procedures become more involved, and more constraints should be imposed on the dependency graphs in order to achieve the PSPACE upper-bound. The PSPACE lower-bound follows from the observation that nonemptiness of the intersection of the regular expressions $e_1, \cdots, e_n$ over the alphabet $\{0, 1\}$, which is a PSPACE-complete problem, can be reduced to the satisfiability of the formula $x \in e_1 \wedge \cdots \wedge x \in e_n$, which falls into all fragments of $\mathsf{SL}[\mathsf{replaceAll}]$ specified in Corollary 4.7. At last, we remark that the restrictions in Corollary 4.7 are partially inspired by the benchmarks in practice. Diamond indices (intuitively, the "nesting depth" of $\mathsf{replaceAll}(x, a, x)$) are likely to be small in practice because the constraints like $\mathsf{replaceAll}(x, a, x)$ are rather artificial and rarely occur in practice. Moreover, the l-length reflects the nesting depth of replaceall w.r.t. the first parameter, which is also likely to be small. Finally, for string constraints with concatenation and replaceAll where pattern/replacement parameters are constants, the diamond index is no greater than the "dimension" defined in [**?**], where it was shown that existing benchmarks mostly have "dimensions" at most three for such string constraints.

5 Outline of Decision Procedures

We describe our decision procedure across three sections (Section 6–Section 8). This means the ideas can be introduced in a step-by-step fashion, which we hope helps the reader. In addition, by presenting separate algorithms, we can give the fine-grained complexity analysis required to show Corollary 4.7. We first outline the main ideas needed by our approach.

We will use automata-theoretic techniques. That is, we make use of the fact that regular expressions can be represented as NFAs. We can then consider a very simple string expression, which is a single regular constraint $x \in e$. It is well-known that an NFA $\mathcal{A}$ can be constructed that is equivalent to e. We can also test in LOGSPACE whether there is some word w accepted by $\mathcal{A}$. If this is the case, then this word can be assigned to x, giving a satisfying assignment to the constraint. If this is not the case, then there is no satisfying assignment.

A more complex case is a conjunction of several constraints of the form $x \in e$. If the constraints apply to different variables, they can be treated independently to find satisfying assignments. If the constraints apply to the same variable, then they can be merged into a single NFA. Intuitively, take $x \in e_1 \wedge x \in e_2$ and $\mathcal{A}_1$ and $\mathcal{A}_2$ equivalent to e_1 and e_2 respectively. We can use the fact that NFA are closed under intersection a check if there is a word accepted by $\mathcal{A}_1 \times \mathcal{A}_2$. If this is the case, we can construct a satisfying assignment to x from an accepting run of $\mathcal{A}_1 \times \mathcal{A}_2$.

In the general case, however, variables are not independent, but may be related by a use of $\mathsf{replaceAll}$. In this case, we perform a kind of $\mathsf{replaceAll}$ *elimination*. That is, we successively remove instances of $\mathsf{replaceAll}$ from the constraint, building up an expanded set of regular constraints (represented as automata). Once there are no more instances of $\mathsf{replaceAll}$ we can solve the regular constraints as above. Briefly, we identify some $x = \mathsf{replaceAll}(y, e, z)$ where x does not appear as an argument to any other use of $\mathsf{replaceAll}$. We then transform any regular constraints on x into additional constraints on y and z. This allows us to remove the variable x since the extended constraints on y and z are sufficient for determining satisfiability. Moreover, from a satisfying assignment to y and z we can construct a satisfying assignment to x as well. This is the technical part of our decision procedure and is explained in detail in the following sections, for increasingly complex uses of $\mathsf{replaceAll}$.

6 Decision procedure for $\mathsf{SL}[\mathsf{replaceAll}]$: The single-letter case

In this section, we consider the single-letter case, that is, for the $\mathsf{SL}[\mathsf{replaceAll}]$ formula $C = \varphi \wedge \psi$, every term of the form $\mathsf{replaceAll}(z, e, z')$ in φ satisfies that $e = a$ for $a \in \Sigma$. We begin by explaining the idea of the decision procedure in the case where there is a single use of a $\mathsf{replaceAll}(-, -, -)$ term. Then we describe the decision procedure in full details.

6.1 A Single Use of $\mathsf{replaceAll}(-, -, -)$

Let us start with the simple case that

$$C \equiv x = \mathsf{replaceAll}(y, a, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3,$$

where, for $i = 1, 2, 3$, we suppose $\mathcal{A}_i = (Q_i, \delta_i, q_{0,i}, F_i)$ is the NFA corresponding to the regular expression e_i.

From the semantics, C is satisfiable if and only if x, y, z can be assigned with strings u, v, w so that: (1) u is obtained from v by replacing all the occurrences of a in v with w, and (2) u, v, w are accepted by $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ respectively. Let u, v, w be the strings satisfying these two constraints. As u is accepted by $\mathcal{A}_1$, there must be an accepting run of $\mathcal{A}_1$ on u. Let $v = v_1 a v_2 a \cdots a v_k$ such that for each $i \in [k]$, $v_i \in (\Sigma \setminus \{a\})^*$. Then $u = v_1 w v_2 w \cdots w v_k$ and there are states $q_1, q'_1, \cdots, q_{k-1}, q'_{k-1}, q_k$ such that

$$q_{0,1} \xrightarrow[\mathcal{A}_1]{v_1} q_1 \xrightarrow[\mathcal{A}_1]{w} q'_1 \xrightarrow[\mathcal{A}_1]{v_2} q_2 \xrightarrow[\mathcal{A}_1]{w} q'_2 \cdots q_{k-1} \xrightarrow[\mathcal{A}_1]{w} q'_{k-1} \xrightarrow[\mathcal{A}_1]{v_k} q_k$$

and $q_k \in F_1$. Let T_z denote $\{(q_i, q'_i) \mid i \in [k-1]\}$. Then $w \in \mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q'))$. In addition, let $\mathcal{B}_{\mathcal{A}_1,a,T_z}$ be the NFA obtained from $\mathcal{A}_1$ by removing all the a-transitions first and then adding the a-transitions (q, a, q') for $(q, q') \in T_z$. Then

$$q_{0,1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_1} q_1 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q'_1 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_2} q_2 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q'_2 \cdots q_{k-1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q'_{k-1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_k} q_k.$$

Therefore, $v \in \mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z})$. We deduce that there is $T_z \subseteq Q_1 \times Q_1$ such that $\mathcal{L}(\mathcal{A}_3) \cap$

$\bigcap_{(q,q')\in T_z} \mathcal{L}(\mathcal{A}_1(q,q')) \neq \emptyset$ and $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z}) \neq \emptyset$. In addition, it is not hard to see that this condition is also sufficient for the satisfiability of C. The arguments proceed as follows: Let $v \in \mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z})$ and $w \in \mathcal{L}(\mathcal{A}_3) \cap \bigcap_{(q,q')\in T_z} \mathcal{L}(\mathcal{A}_1(q,q'))$. From $v \in \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z})$, we know that there is an accepting run of $\mathcal{B}_{\mathcal{A}_1,a,T_z}$ on v. Recall that $\mathcal{B}_{\mathcal{A}_1,a,T_z}$ is obtained from $\mathcal{A}_1$ by first removing all the a-transitions, then adding all the transitions (q,a,q') for $(q,q') \in T_z$. Suppose $v = v_1 a v_2 \cdots a v_k$ such that $v_i \in (\Sigma \setminus \{a\})^*$ for each $i \in [k]$ and

$$q_{0,1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_1} q_1 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q_1' \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_2} q_2 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q_2' \cdots q_{k-1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q_{k-1}' \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_k} q_k$$

is an accepting run of $\mathcal{B}_{\mathcal{A}_1,a,T_z}$ on v. Then $q_{0,1} \xrightarrow[\mathcal{A}_1]{v_1} q_1$, and for each $i \in [k-1]$ we have $(q_i, q_i') \in T_z$ and $q_i' \xrightarrow[\mathcal{A}_1]{v_{i+1}} q_{i+1}$; moreover, $q_k \in F_1$. Let $u = \mathsf{replaceAll}(v,a,w) = v_1 w v_2 \cdots w v_k$. Since $w \in \bigcap_{(q,q')\in T_z} \mathcal{L}(\mathcal{A}_1(q,q'))$, we infer that

$$q_{0,1} \xrightarrow[\mathcal{A}_1]{v_1} q_1 \xrightarrow[\mathcal{A}_1]{w} q_1' \xrightarrow[\mathcal{A}_1]{v_2} q_2 \xrightarrow[\mathcal{A}_1]{w} q_2' \cdots q_{k-1} \xrightarrow[\mathcal{A}_1]{w} q_{k-1}' \xrightarrow[\mathcal{A}_1]{v_k} q_k$$

is an accepting run of $\mathcal{A}_1$ on u. Therefore, u is accepted by $\mathcal{A}_1$ and C is satisfiable.

Proposition 6.1. *We have $C \equiv x = \mathsf{replaceAll}(y,a,z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$ is satisfiable iff there exists $T_z \subseteq Q_1 \times Q_1$ with $\mathcal{L}(\mathcal{A}_3) \cap \bigcap_{(q,q')\in T_z} \mathcal{L}(\mathcal{A}_1(q,q')) \neq \emptyset$ and $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z}) \neq \emptyset$.*

From Proposition 6.1, we can decide the satisfiability of C in polynomial space as follows:

Step I. Nondeterministically choose a set $T_z \subseteq Q_1 \times Q_1$.

Step II. Nondeterministically choose an accepting run of the product automaton of $\mathcal{A}_3$ and $\mathcal{A}_1(q,q')$ for $(q,q') \in T_z$.

Step III. Nondeterministically choose an accepting run of the product automaton of $\mathcal{A}_2$ and $\mathcal{B}_{\mathcal{A}_1,a,T_z}$.

During Step II and III, it is sufficient to record T_z and a state of the product automaton, which occupies only a polynomial space.

The above decision procedure can be easily generalised to the case that there are multiple atomic regular constraints for x. For instance, let $x \in e_{1,1} \wedge x \in e_{1,2}$ and for $j = 1,2$, $\mathcal{A}_{1,j} = (Q_{1,j}, \delta_{1,j}, q_{0,1,j}, F_{1,j})$ be the NFA corresponding to $e_{1,j}$. Then in Step I, two sets $T_{1,z} \subseteq Q_{1,1} \times Q_{1,1}$ and $T_{2,z} \subseteq Q_{1,2} \times Q_{1,2}$ are nondeterministically chosen, moreover, Step II and III are adjusted accordingly.

Example 6.2. *Let $C \equiv x = \mathsf{replaceAll}(y,0,z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$, where $e_1 = (0+1)^*(00(0+1)^*+11(0+1)^*)$, $e_2 = (01)^*$, and $e_3 = (10)^*$. The NFA $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ corresponding to e_1, e_2, e_3 respectively are illustrated in Figure 2. Let $T_z = \{(q_0,q_0),(q_1,q_2)\}$. Then*

$$\begin{aligned}
\mathcal{L}(\mathcal{A}_3) \cap \bigcap_{(q,q')\in T_z} \mathcal{L}(\mathcal{A}_1(q,q')) &= \mathcal{L}(\mathcal{A}_3) \cap \mathcal{L}(\mathcal{A}_1(q_0,q_0)) \cap \mathcal{L}(\mathcal{A}_1(q_1,q_2)) \\
&= \mathcal{L}((10)^*) \cap \mathcal{L}((0+1)^*) \cap \mathcal{L}(1(0+1)^*) \\
&\neq \emptyset.
\end{aligned}$$

In addition, $\mathcal{B}_{\mathcal{A}_1,0,T_z}$ (also illustrated in Figure 2) is obtained from $\mathcal{A}_1$ by removing all the 0-transitions, then adding the transitions $(q_0, 0, q_0)$ and $(q_1, 0, q_2)$. Then

$$\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,0,T_z}) = \mathcal{L}((01)^*) \cap \mathcal{L}((0+1)^*101^*) \neq \emptyset.$$

We can choose z to be a string from $\mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q')\in T_z} \mathcal{L}(\mathcal{A}_1(q,q')) = \mathcal{L}((10)^*) \cap \mathcal{L}((0+1)^*) \cap \mathcal{L}(1(0+1)^*)$, say 10, and y to be a string from $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,0,T_z}) = \mathcal{L}((01)^*) \cap \mathcal{L}((0+1)^*101^*)$, say 0101, then we set x to $\mathsf{replaceAll}(0101, 0, 10) = 101101$, which is in $\mathcal{L}(\mathcal{A}_1)$. Thus, C is satisfiable. □

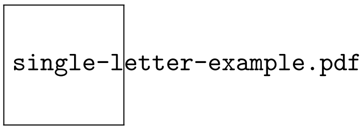

Figure 2: An example for the single-letter case: One replaceAll

6.2 The General Case

Let us now consider the general case where C contains multiple occurrences of $\mathsf{replaceAll}(-,-,-)$ terms. Then the satisfiability of C is decided by the following two-step procedure.

Step I. We utilise the dependency graph C and compute nondeterministically a collection of atomic regular constraints $\mathcal{E}(x)$ for each variable x, in a top-down manner.

Notice that $\mathcal{E}(x)$ is represented succinctly as a set of pairs $(\mathcal{T}, \mathcal{P})$, where $\mathcal{T} = (Q, \delta)$ is a transition graph and $\mathcal{P} \subseteq Q \times Q$. The intention of $(\mathcal{T}, \mathcal{P})$ is to represent succinctly the collection of the atomic regular constraints containing $(Q, \delta, q, \{q'\})$ for each $(q, q') \in \mathcal{P}$, where q is the initial state and $\{q'\}$ is the set of final states.

Initially, let $G_0 := G_C$. In addition, for each variable x, we define $\mathcal{E}_0(x)$ as follows: Let $x \in e_1 \wedge \cdots \wedge x \in e_n$ be the conjunction of all the atomic regular constraints related to x in C. For each $i \in [n]$, let $\mathcal{A}_i = (Q_i, \delta_i, q_{0,i}, F_i)$ be the NFA corresponding to e_i. We nondeterministically choose $q_i \in F_i$ and set $\mathcal{E}_0(x) := \{((Q_i, \delta_i), \{(q_{0,i}, q_i)\}) \mid i \in [n]\}$.

We begin with $i := 0$ and repeat the following procedure until we reach some i where G_i is an empty graph, i.e. a graph without edges. Note that G_0 was defined above.

1. Select a vertex x of G_i such that x has no predecessors and has two successors via edges $(x, (\mathsf{l}, a), y)$ and $(x, (\mathsf{r}, a), z)$ in G_i. Suppose $\mathcal{E}_i(x) = \{(\mathcal{T}_1, \mathcal{P}_1), \cdots, (\mathcal{T}_k, \mathcal{P}_k)\}$, where for each $j \in [k]$, $\mathcal{T}_j = (Q_j, \delta_j)$. Then $\mathcal{E}_{i+1}(z)$ and $\mathcal{E}_{i+1}(y)$ and G_{i+1} are computed as follows:
   (a) For each $j \in [k]$, nondeterministically choose a set $T_{j,z} \subseteq Q_j \times Q_j$.
   (b) If $y \neq z$, then let
   $$\mathcal{E}_{i+1}(z) := \mathcal{E}_i(z) \cup \{(\mathcal{T}_j, T_{j,z}) \mid j \in [k]\}$$
   and
   $$\mathcal{E}_{i+1}(y) := \mathcal{E}_i(y) \cup \left\{(\mathcal{T}_{\mathcal{T}_j,a,T_{j,z}}, \mathcal{P}_j) \mid j \in [k]\right\}$$
   where $\mathcal{T}_{\mathcal{T}_j,a,T_{j,z}}$ is obtained from $\mathcal{T}_j$ by first removing all the a-transitions, then adding all the transitions (q, a, q') for $(q, q') \in T_{j,z}$. Otherwise, let $\mathcal{E}_{i+1}(z) := \mathcal{E}_i(z) \cup \{(\mathcal{T}_j, T_{j,z}) \mid j \in [k]\} \cup \left\{(\mathcal{T}_{\mathcal{T}_j,a,T_{j,z}}, \mathcal{P}_j) \mid j \in [k]\right\}$. In addition, for each vertex x' distinct from y, z, let $\mathcal{E}_{i+1}(x') := \mathcal{E}_i(x')$.

(c) Let $G_{i+1} := G_i \setminus \{(x, (\mathsf{l}, a), y), (x, (\mathsf{r}, a), z)\}$.

2. Let $i := i + 1$.

For each variable x, let $\mathcal{E}(x)$ denote the set $\mathcal{E}_i(x)$ after exiting the above loop.

Step II. Output "satisfiable" if for each source variable x there is an accepting run of the product of all the NFA in $\mathcal{E}(x)$; otherwise, output "unsatisfiable".

It remains to argue the correctness and complexity of the above procedure and show how to obtain satisfying assignments to satisfiable constraints. Correctness follows a similar argument to Proposition 6.1 and is presented in Appendix B. Intuitively, Proposition 6.1 shows our procedure correctly eliminates occurrences of replaceAll until only regular constraints remain.

If, in the case that the equation is satisfiable, one wishes to obtain a satisfying assignment to all variables, we can proceed as follows. First, for each source variable x, nondeterministically choose an accepting run of the product of all the NFA in $\mathcal{E}(x)$. As argued in Appendix B, the word labelling this run satisfies all regular constraints on x since it is taken from a language that is guaranteed to be a subset of the set of words satisfying the original constraints. For non-source variables, we derive an assignment as in Proposition 6.1, proceeding by induction from the source variables. That is, select some variable x such that x is derived from variables y and z and assignments to both y and z have already been obtained. The value for x is immediately obtained by performing the replaceAll operation using the assignments to y and z. That this value satisfies all regular constraints on x follows the same argument as Proposition 6.1. The procedure terminates when all variables have been assigned.

We now give an example before proceeding to the complexity analysis.

Example 6.3. *Suppose $C \equiv x = \mathsf{replaceAll}(y, 0, z) \wedge y = \mathsf{replaceAll}(y', 1, z') \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3 \wedge y' \in e_4 \wedge z' \in e_5$, where e_1, e_2, e_3 are as in Example 6.2, $e_4 = 0^*1^*0^*1^*$, and $e_5 = 0^*1^*$. Let $\mathcal{A}_4, \mathcal{A}_5$ be the NFA corresponding to e_4 and e_5 respectively (see Figure 3). The dependency graph G_C of C is illustrated in Figure 3. Let $\mathcal{T}_1, \cdots, \mathcal{T}_5$ be the transition graph of $\mathcal{A}_1, \cdots, \mathcal{A}_5$ respectively. Then the collection of regular constraints $\mathcal{E}(\cdot)$ are computed as follows.*

- *Let $G_0 = G_C$. Pick the sets $\mathcal{E}_0(x) = \{(\mathcal{T}_1, \{(q_0, q_2)\})\}$, $\mathcal{E}_0(y) = \{(\mathcal{T}_2, \{(q_0', q_0')\})\}$, $\mathcal{E}_0(z) = \{(\mathcal{T}_3, \{(q_0'', q_0'')\})\}$, $\mathcal{E}_0(y') = \{(\mathcal{T}_4, \{(p_0, p_1)\})\}$, and $\mathcal{E}_0(z') = \{(\mathcal{T}_5, \{(p_0', p_1')\})\}$ nondeterministically.*

- *Select the vertex x in G_0, construct $\mathcal{E}_1(y)$ and $\mathcal{E}_1(z)$ as in Example 6.2, that is, nondeterministically choose $T_z = \{(q_0, q_0), (q_1, q_2)\}$, let*
$$\mathcal{E}_1(z) = \{(\mathcal{T}_3, \{(q_0'', q_0'')\}), (\mathcal{T}_1, \{(q_0, q_0), (q_1, q_2)\})\}$$
and
$$\mathcal{E}_1(y) = \{(\mathcal{T}_2, \{(q_0', q_0')\}), (\mathcal{T}_{\mathcal{T}_1,0,T_z}, \{(q_0, q_2)\})\},$$
where $\mathcal{T}_{\mathcal{T}_1,0,T_z}$ is the transition graph of $\mathcal{B}_{\mathcal{A}_1,0,T_z}$ illustrated in Figure 2. In addition, $\mathcal{E}_1(x) = \mathcal{E}_0(x)$, $\mathcal{E}_1(y') = \mathcal{E}_0(y')$ and $\mathcal{E}_1(z') = \mathcal{E}_0(z')$. Finally, we get G_1 from G_0 by removing the two edges from x.

- *Select the vertex y in G_1, construct $\mathcal{E}_2(y')$ and $\mathcal{E}_2(z')$ as follows: Nondeterministically choose $T_{1,z'} = \{(q_0', q_0')\}$ for $\mathcal{T}_2$ and $T_{2,z'} = \{(q_0, q_1), (q_1, q_2)\}$ for $\mathcal{T}_{\mathcal{T}_1,0,T_z}$, let*
$$\mathcal{E}_2(z') = \{(\mathcal{T}_5, \{(p_0', p_1')\}), (\mathcal{T}_2, \{(q_0', q_0')\}), (\mathcal{T}_{\mathcal{T}_1,0,T_z}, \{(q_0, q_1), (q_1, q_2)\})\}, \text{ and}$$

$$\mathcal{E}_2(y') = \Big\{(\mathcal{T}_4, \{(p_0,p_1)\}), (\mathcal{T}_{\mathcal{T}_2,1,T_{1,z'}}, \{(q_0',q_0')\}), (\mathcal{T}_{\mathcal{T}_{T_1,0,T_z},1,T_{2,z'}}, \{(q_0,q_2)\})\Big\},$$

where $\mathcal{T}_{\mathcal{T}_2,1,T_{1,z'}}$ and $\mathcal{T}_{\mathcal{T}_{T_1,0,T_z},1,T_{2,z'}}$ are shown in Figure 4. In addition, $\mathcal{E}_2(x) = \mathcal{E}_1(x)$, $\mathcal{E}_2(y) = \mathcal{E}_1(y)$, and $\mathcal{E}_2(z) = \mathcal{E}_1(z)$. Finally, we get G_2 from G_1 by removing the two edges from y.

Since G_2 contains no edges, we have $\mathcal{E}(x) = \mathcal{E}_2(x)$, similarly for $\mathcal{E}(y)$, $\mathcal{E}(z)$, $\mathcal{E}(y')$, and $\mathcal{E}(z')$. For the three source variables y', z', z, it is not hard to check that 01 *belongs to the intersection of the regular constraints in $\mathcal{E}(z')$,* 11 *belongs to the intersection of the regular constraints in $\mathcal{E}(y')$, and* 10 *belongs to the intersection of the regular constraints in $\mathcal{E}(z)$. Then y takes the value* $\mathsf{replaceAll}(11,1,01) = 0101 \in \mathcal{L}(e_2)$*, and x takes the value* $\mathsf{replaceAll}(0101,0,10) = 101101 \in \mathcal{L}(e_1)$*. Therefore, C is satisfiable.* □

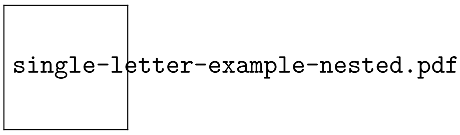

Figure 3: An example for the single-letter case: Multiple replaceAll

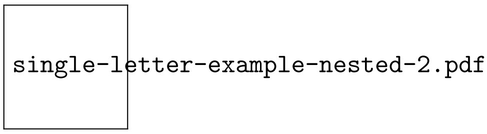

Figure 4: $\mathcal{T}_{\mathcal{T}_2,1,T_{1,z'}}$ and $\mathcal{T}_{\mathcal{T}_{T_1,0,T_z},1,T_{2,z'}}$

6.2.1 Complexity

To show our decision procedure works in exponential space, it is sufficient to show that the cardinalities of the sets $\mathcal{E}(x)$ are exponential w.r.t. the size of C.

Proposition 6.4. *The cardinalities of $\mathcal{E}(x)$ for the variables x in G_C are at most exponential in $\mathsf{Idx}_{\mathsf{dmd}}(G_C)$, the diamond index of G_C.*

Therefore, according to Proposition 6.4, if the diamond index of G_C is bounded by a constant c, then the cardinalities of $\mathcal{E}(x)$ become *polynomial* in the size of C and we obtain a polynomial space decision procedure. In this case, we conclude that the satisfiability problem is PSPACE-complete.

Proof of Proposition 6.4. Let K be the maximum of $|\mathcal{E}_0(x)|$ for $x \in \mathsf{Vars}(\varphi)$. For each variable x in G_C, all the regular constraints in $\mathcal{E}(x)$ are either from $\mathcal{E}_0(x)$, or are generated from some regular constraints from $\mathcal{E}_0(x')$ for the ancestors x' of x. Let x' be an ancestor of x. Then for each $(\mathcal{T}, \mathcal{P}) \in \mathcal{E}_0(x')$, according to Step I in the decision procedure, by an induction on the maximum length of the paths in from x' to x, we can show that the number of elements in $\mathcal{E}(x)$ that are generated from $(\mathcal{T}, \mathcal{P})$ is at most the number of different paths from x' to x. From Proposition 4.5, we know that there are at most $(|\mathsf{Vars}(\varphi)| \cdot |E_C|)^{O(\mathsf{Idx}_{\mathsf{dmd}}(G_C))}$ different paths from x' to x. Since there are at most $|\mathsf{Vars}(\varphi)|$ ancestors of x, we deduce that $|\mathcal{E}(x)| \leq K \cdot |\mathsf{Vars}(\varphi)| \cdot (|\mathsf{Vars}(\varphi)||E_C|)^{O(\mathsf{Idx}_{\mathsf{dmd}}(G_C))}$. □

7 Decision procedure for $\mathsf{SL}[\mathsf{replaceAll}]$: The constant-string case

In this section, we consider the constant-string special case, that is, for an $\mathsf{SL}[\mathsf{replaceAll}]$ formula $C = \varphi \wedge \psi$, every term of the form $\mathsf{replaceAll}(z, e, z')$ in φ satisfies that $e = u$ for $u \in \Sigma^+$. Note that the case when $u = \epsilon$ will be dealt with in Section 8.

Again, let us start with the simple situation that $C \equiv x = \mathsf{replaceAll}(y, u, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$, where $|u| \geq 2$. For $i = 1, 2, 3$, let $\mathcal{A}_i = (Q_i, \delta_i, q_{0,i}, F_i)$ be the NFA corresponding to e_i. In addition, let $k = |u|$ and $u = a_1 \cdots a_k$ with $a_i \in \Sigma$ for each $i \in [k]$.

From the semantics, C is satisfiable iff x, y, z can be assigned with strings v, w, w' such that: (1) $v = \mathsf{replaceAll}(w, u, w')$, and (2) v, w, w' are accepted by $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ respectively. Let v, w, w' be the strings satisfying these two constraints. Since $v = \mathsf{replaceAll}(w, u, w')$, we know that there are strings $w_1, w_2, \cdots, w_n$ such that $w = w_1 u w_2 \cdots u w_n$ and $v = w_1 w' w_2 \cdots w' w_n$. As v is accepted by $\mathcal{A}_1$, there is an accepting run of $\mathcal{A}_1$ on v, say

$$q_{0,1} \xrightarrow[\mathcal{A}_1]{w_1} q_1 \xrightarrow[\mathcal{A}_1]{w'} q'_1 \xrightarrow[\mathcal{A}_1]{w_2} q_2 \xrightarrow[\mathcal{A}_1]{w'} q'_2 \cdots q_{n-1} \xrightarrow[\mathcal{A}_1]{w'} q'_{n-1} \xrightarrow[\mathcal{A}_1]{w_n} q_n.$$

Let $T_z = \{(q_i, q'_i) \mid i \in [n]\}$. Then $w' \in \mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q'))$. Therefore, $\mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q')) \neq \emptyset$. Similar to the single-letter case, we construct an NFA $\mathcal{B}_{\mathcal{A}_1, u, T_z}$ to characterise the satisfiability of C. More precisely, C is satisfiable iff there is $T_z \subseteq Q_1 \times Q_1$ such that $\mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q')) \neq \emptyset$ and $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1, u, T_z}) \neq \emptyset$. Intuitively, when reading the string w, $\mathcal{B}_{\mathcal{A}_1, u, T_z}$ simulates the generation of v from w and w' (that is, the replacement of every occurrence of u in w with w') and verifies that v is accepted by $\mathcal{A}_1$, by using T_z. To build $\mathcal{B}_{\mathcal{A}_1, u, T_z}$, we utilise the concepts of window profiles and parsing automata defined below. Intuitively, a window profile keeps track of which positions in the preceding characters could form the beginning of a match of u.

Definition 7.1 (window profiles w.r.t. u)**.** *Let v be a nonempty string with $k = |v|$, and $i \in [k]$. Then the* window profile of the position i in v w.r.t. u *is $\overrightarrow{W} \in \{\bot, \top\}^{k-1}$ defined as follows:*

- *If $i \geq k-1$, then for each $j \in [k-1]$, $\overrightarrow{W}[j] = \top$ iff $v[i-j+1] \cdots v[i] = u[1] \cdots u[j]$.*
- *If $i < k-1$, then for each $j \in [i]$, $\overrightarrow{W}[j] = \top$ iff $v[i-j+1] \cdots v[i] = u[1] \cdots u[j]$, and for each $j : i < j \leq k-1$, $\overrightarrow{W}[j] = \bot$.*

Let WP_u denote the set of window profiles of the positions in nonempty strings w.r.t. u.

Proposition 7.2. $|\mathsf{WP}_u| \leq |u|$.

Proof. Let $k = |u|$. For each profile $\overrightarrow{W}$, let v be a nonempty string and i be a position of v such that for each $j \in [k-1]$, $\overrightarrow{W}[j] = \top$ iff $v[i-j+1] \ldots v[i] = u[1] \ldots u[j]$. Define $\mathsf{idx}_{\overrightarrow{W}}$ as follows: If there is $j \in [k-1]$ such that $\overrightarrow{W}[j] = \top$, then $\mathsf{idx}_{\overrightarrow{W}}$ is the maximum of such indices $j \in [k-1]$, otherwise, $\mathsf{idx}_{\overrightarrow{W}} = 0$. The following fact holds for $\overrightarrow{W}$ and $\mathsf{idx}_{\overrightarrow{W}}$:

- for each $j' : \mathsf{idx}_{\overrightarrow{W}} < j' \leq k-1$, $\overrightarrow{W}[j'] = \bot$,

- in addition, since $v[i-\mathsf{idx}_{\overrightarrow{W}}+1]\cdots v[i] = u[1]\cdots u[\mathsf{idx}_{\overrightarrow{W}}]$, the values of $\overrightarrow{W}[1],\cdots,\overrightarrow{W}[\mathsf{idx}_{\overrightarrow{W}}]$ are completely determined by $u[1]\cdots u[\mathsf{idx}_{\overrightarrow{W}}]$.

Let $\eta : \mathsf{WP}_u \to \{0\}\cup[k-1]$ be a function such that for each $\overrightarrow{W}\in\mathsf{WP}_u$, $\eta(\overrightarrow{W}) = \mathsf{idx}_{\overrightarrow{W}}$. Then η is an injective function, since for every $\overrightarrow{W},\overrightarrow{W'}\in\mathsf{WP}_u$, $\mathsf{idx}_{\overrightarrow{W}} = \mathsf{idx}_{\overrightarrow{W'}}$ iff $\overrightarrow{W} = \overrightarrow{W'}$. Therefore, we conclude that $|\mathsf{WP}_u| \leq k$. □

Example 7.3. *Let $\Sigma = \{0,1\}$, $u = 010$. Then $\mathsf{WP}_u = \{\bot\bot, \top\bot, \bot\top\}$.*

- *Consider the string $v = 1$ and the position $i = 1$ in v. Since $v[1] = 1 \neq u[1] = 0$, the window profile of i in v w.r.t. u is $\bot\bot$.*
- *Consider the string $v = 00$ and the position $i = 2$ in v. Since $v[2] = u[1]$ and $v[1]v[2] \neq u[1]u[2]$, the window profile of i in v w.r.t. u is $\top\bot$.*
- *Consider the string $v = 01$ and the position $i = 2$ in v. Since $v[2] \neq u[1]$ and $v[1]v[2] = u[1]u[2]$, the window profile of i in v w.r.t. u is $\bot\top$.*

Note that $\top\top \notin \mathsf{WP}_u$, since for every string v and the position i in v, if $v[i-1]v[i] = u[1]u[2] = 01$, then $v[i] = 1 \neq 0 = u[1]$.

We will construct a parsing automaton $\mathcal{A}_u$ from u, which parses a string v containing at least one occurrence of u (i.e. $v \in \Sigma^* u \Sigma^*$) into $v_1 u v_2 u \ldots v_l u v_{l+1}$ such that $v_j u[1]\ldots u[k-1] \notin \Sigma^* u \Sigma^*$ for each $1 \leq j \leq l$. This ensures that the only occurrence of u in each $v_j u$ is a suffix. Finally, we also require $v_{l+1} \notin \Sigma^* u \Sigma^*$. The window profiles w.r.t. u will be used to ensure that v is correctly parsed, namely, the first, second, $\cdots$, occurrences of u are correctly identified.

Definition 7.4 (Parsing automata)**.** *Given a string u we define the* parsing automaton *$\mathcal{A}_u$ to be the NFA $(Q_u, \delta_u, q_{0,u}, F_u)$ where $q_{0,u} = q_0$ and the remaining components are given below.*

- *$Q_u = \{q_0\} \cup \left\{\left(\mathsf{search}, \overrightarrow{W}\right) \mid \overrightarrow{W} \in \mathsf{WP}_u\right\} \cup \left\{\left(\mathsf{vfy}, j, \overrightarrow{W}\right) \mid j \in [k-1], \overrightarrow{W} \in \mathsf{WP}_u\right\}$, where q_0 is a distinguished state whose purpose will become clear later on, and the tags "*search*" and "*vfy*" are used to denote whether $\mathcal{A}_u$ is in the "search" mode to search for the next occurrence of u, or in the "verify" mode to verify that the current position is a part of an occurrence of u.*
- *δ_u is defined as follows.*
  - *The transition $\left(q_0, a, \left(\mathsf{search}, \overrightarrow{W}\right)\right) \in \delta_u$, where $\overrightarrow{W}[1] = \top$ iff $a = u[1]$, and for each $i : 2 \leq i \leq k-1$, $\overrightarrow{W}[i] = \bot$.*
  - *The transition $\left(q_0, u[1], \left(\mathsf{vfy}, 1, \overrightarrow{W}\right)\right) \in \delta_u$, where $\overrightarrow{W}[1] = \top$ and for each $i : 2 \leq i \leq k-1$, $\overrightarrow{W}[i] = \bot$.*
  - *For each state $\left(\mathsf{search}, \overrightarrow{W}\right)$ and $a \in \Sigma$ such that $\overrightarrow{W}[k-1] = \bot$ or $a \neq u[k]$,*
    - *the transition $\left(\left(\mathsf{search}, \overrightarrow{W}\right), a, \left(\mathsf{search}, \overrightarrow{W'}\right)\right) \in \delta_u$, where $\overrightarrow{W'}[1] = \top$ iff $a = u[1]$, and for each $i : 2 \leq i \leq k-1$, $\overrightarrow{W'}[i] = \top$ iff ($\overrightarrow{W}[i-1] = \top$ and $a = u[i]$),*
    - *if $a = u[1]$, then the transition $\left(\left(\mathsf{search}, \overrightarrow{W}\right), a, \left(\mathsf{vfy}, 1, \overrightarrow{W'}\right)\right) \in \delta_u$, where $\overrightarrow{W'}[1] = \top$, and for each $i : 2 \leq i \leq k-1$, $\overrightarrow{W'}[i] = \top$ iff ($\overrightarrow{W}[i-1] = \top$ and $a = u[i]$).*

- *For each state* $\left(\mathsf{vfy}, i-1, \overrightarrow{W}\right)$ *and* $a \in \Sigma$ *such that*
  - $2 \leq i \leq k-1$,
  - $\overrightarrow{W}[i-1] = \top$, $a = u[i]$, *and*
  - *either* $\overrightarrow{W}[k-1] = \bot$ *or* $a \neq u[k]$,

  we have $\left(\left(\mathsf{vfy}, i-1, \overrightarrow{W}\right), a, \left(\mathsf{vfy}, i, \overrightarrow{W'}\right)\right) \in \delta_u$, *where for each* $j : 2 \leq j \leq k-1$, $\overrightarrow{W'}[j] = \top$ *iff* $\overrightarrow{W}[j-1] = \top$ *and* $a = u[j]$.
- *For each state* $\left(\mathsf{vfy}, k-1, \overrightarrow{W}\right)$ *and* $a \in \Sigma$ *such that* $\overrightarrow{W}[k-1] = \top$ *and* $a = u[k]$, *we have* $\left(\left(\mathsf{vfy}, k-1, \overrightarrow{W}\right), a, q_0\right) \in \delta_u$.

Note that the constraint $\overrightarrow{W}[k-1] = \bot$ *or* $a \neq u[k]$ *is used to guarantee that each occurrence of the state* q_0*, except the first one, witnesses the* first *occurrence of* u *from the beginning or after its previous occurrence. In other words, the constraint* $\overrightarrow{W}[k-1] = \bot$ *or* $a \neq u[k]$ *is used to guarantee that after an occurrence of* q_0*, if* q_0 *has not been reached again, then* u *is forbidden to occur.*

- $F_u = \{q_0\} \cup \left\{\left(\mathsf{search}, \overrightarrow{W}\right) \mid \overrightarrow{W} \in \mathsf{WP}_u\right\}$.

  Note that the states $\left(\mathsf{vfy}, j, \overrightarrow{W}\right)$ *are not final states, since, when in these states, the verification of the current occurrence of* u *has not been complete yet.*

Let $Q_{\mathsf{search}} = \left\{\left(\mathsf{search}, \overrightarrow{W}\right) \mid \overrightarrow{W} \in \mathsf{WP}_u\right\}$, and $Q_{\mathsf{vfy},i} = \left\{\left(\mathsf{vfy}, i, \overrightarrow{W}\right) \mid \overrightarrow{W} \in \mathsf{WP}_u\right\}$ for each $i \in [k-1]$. In addition, let $Q_{\mathsf{vfy}} = \bigcup\limits_{i \in [k-1]} Q_{\mathsf{vfy},i}$. Suppose $v = v_1 u v_2 u \cdots v_l u v_{l+1}$ such that $v_j u[1] \dots u[k-1] \notin \Sigma^* u \Sigma^*$ for each $1 \leq j \leq l$, in addition, $v_{l+1} \notin \Sigma^* u \Sigma^*$. Then there exists a *unique* accepting run r of $\mathcal{A}_u$ on v such that the state sequence in r is of the form $q_0\ r_1\ q_0\ r_2\ q_0\ \cdots\ r_l\ q_0\ r_{l+1}$, where for each $j \in [l]$, $r_j \in \mathcal{L}((Q_{\mathsf{search}})^+ \circ Q_{\mathsf{vfy},1} \circ \cdots \circ Q_{\mathsf{vfy},k-1})$, and $r_{l+1} \in \mathcal{L}((Q_{\mathsf{search}})^*)$.

Example 7.5. *Consider* $u = 010$ *in Example 7.3. The parsing automaton* $\mathcal{A}_u$ *is illustrated in Figure 5. Note that there are no* 0*-transitions out of* $(\mathsf{search}, \bot\top)$*, since this would imply an occurrence of* $u = 010$*, which should be verified by the states from* Q_{vfy}*, more precisely, by the state sequence* $q_0(\mathsf{vfy}, 1, \top\bot)(\mathsf{vfy}, 2, \bot\top)q_0$.

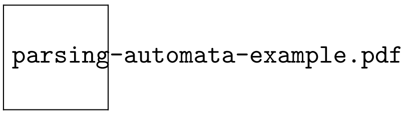

Figure 5: The parsing automaton $\mathcal{A}_u$ for $u = 010$

We are ready to present the construction of $\mathcal{B}_{\mathcal{A}_1,u,T_z}$. The NFA $\mathcal{B}_{\mathcal{A}_1,u,T_z}$ is constructed by the following three-step procedure.

1. Construct the product automaton $\mathcal{A}_1 \times \mathcal{A}_u$. Note that the initial state of $\mathcal{A}_1 \times \mathcal{A}_u$ is (q_0, q_0) and the set of final states of $\mathcal{A}_1 \times \mathcal{A}_u$ is $F_1 \times F_u$.

2. Remove from $\mathcal{A}_1 \times \mathcal{A}_u$ all the (incoming or outgoing) transitions associated with the states from $Q_1 \times Q_{\mathsf{vfy}}$.

3. For each pair $(q, q') \in T_z$ and each sequence of transitions in $\mathcal{A}_u$ of the form

$$\left(p, u[1], \left(\mathsf{vfy}, 1, \overrightarrow{W'_1}\right)\right), \left(\left(\mathsf{vfy}, 1, \overrightarrow{W'_1}\right), u[2], \left(\mathsf{vfy}, 2, \overrightarrow{W'_2}\right)\right), \cdots, \left(\left(\mathsf{vfy}, k-1, \overrightarrow{W'_{k-1}}\right), u[k], q_0\right),$$

where $p = q_0$ or $p = \left(\mathsf{search}, \overrightarrow{W}\right)$, add the following transitions

$$\begin{gathered}\left((q,p), u[1], \left(q, \left(\mathsf{vfy}, 1, \overrightarrow{W'_1}\right)\right)\right), \\ \left(\left(q, \left(\mathsf{vfy}, 1, \overrightarrow{W'_1}\right)\right), u[2], \left(q, \left(\mathsf{vfy}, 2, \overrightarrow{W'_2}\right)\right)\right), \\ \cdots, \\ \left(\left(q, \left(\mathsf{vfy}, k-2, \overrightarrow{W'_{k-2}}\right)\right), u[k-1], \left(q, \left(\mathsf{vfy}, k-1, \overrightarrow{W'_{k-1}}\right)\right)\right), \\ \left(\left(q, \left(\mathsf{vfy}, k-1, \overrightarrow{W'_{k-1}}\right)\right), u[k], (q', q_0)\right).\end{gathered}$$

Note that the number of aforementioned sequences of transitions in $\mathcal{A}_u$ is at most $|Q_{\mathsf{search}}| + 1$, since $\overrightarrow{W'_1}, \ldots, \overrightarrow{W'_{k-1}}$ are completely determined by $\overrightarrow{W}$ and u. Intuitively, when $\mathcal{A}_u$ identifies an occurrence of u, if the current state of $\mathcal{A}_1$ is q, then after reading the occurrence of u, $\mathcal{B}_{\mathcal{A}_1,u,T_z}$ jumps from q to some state q' such that $(q, q') \in T_z$.

Example 7.6. *Consider $C \equiv x = \mathsf{replaceAll}(y, u, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$, where $u = 010$, and e_1, e_2, e_3 are as in Example 6.2 (cf. Figure 2). Let $T_z = \{(q_0, q_0), (q_1, q_2)\}$. The NFA $\mathcal{B}_{\mathcal{A}_1,u,T_z}$ is obtained from the product automaton $\mathcal{A}_1 \times \mathcal{A}_u$ (which we give in the appendix for reference) by first removing all the transitions associated with the states from $Q_1 \times Q_{\mathsf{vfy}}$, then adding the transitions according to T_z as aforementioned (see Figure 6, where thick edges indicate added transitions). It is routine to check that 01010101 is accepted by $\mathcal{B}_{\mathcal{A}_1,u,T_z}$ and $\mathcal{A}_2$. Moreover, $10 \in \mathcal{L}(\mathcal{A}_3) \cap \mathcal{L}(\mathcal{A}_1(q_0, q_0)) \cap \mathcal{L}(\mathcal{A}_1(q_1, q_2))$. Let y be 01010101 and z be 10. Then x takes the value $\mathsf{replaceAll}(01010101, 010, 10) = 101101$, which is accepted by $\mathcal{A}_1$. Therefore, C is satisfiable.*

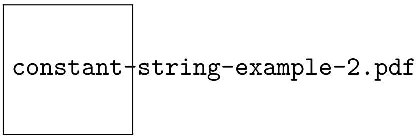

Figure 6: The NFA $\mathcal{B}_{\mathcal{A}_1,u,T_z}$ for $u = 010$ and $T_z = \{(q_0, q_0), (q_1, q_2)\}$

For the more general case that the $\mathsf{SL}[\mathsf{replaceAll}]$ formula C contains more than one occurrence of $\mathsf{replaceAll}(-,-,-)$ terms, similar to the single-letter case in Section 6, we can nondeterministically remove the edges in the dependency graph G_C in a top-down manner and reduce the satisfiability of C to the satisfiability of a collection of regular constraints for source variables.

Complexity When constructing G_{i+1} from G_i, suppose the two edges from x to y and z respectively are currently removed, let the labels of the two edges be (l, u) and (r, u) respectively.

Then each element $(\mathcal{T}, \mathcal{P})$ of $\mathcal{E}_i(x)$ may be transformed into an element $(\mathcal{T}', \mathcal{P}')$ of $\mathcal{E}_{i+1}(y)$ such that $|\mathcal{T}'| = O(|u||\mathcal{T}|)$, meanwhile, it may also be transformed into an element $(\mathcal{T}'', \mathcal{P}'')$ of $\mathcal{E}_{i+1}(z)$ such that $\mathcal{T}''$ has the same state space as $\mathcal{T}$. In each step of the decision procedure, the state space of the regular constraints may be multiplied by a factor $|u|$. The state space of these regular constraints is at most exponential in the end, so that we can still solve the nonemptiness problem of the intersection of all these regular constraints in exponential space. In addition, if the l-length of G_C is bounded by a constant c, then for each source variable, we get polynomially many regular constraints, where each of them has a state space of polynomial size. Therefore, we can get a polynomial space algorithm. See Appendix D for a detailed analysis.

8 Decision procedure for $\mathsf{SL}[\mathsf{replaceAll}]$: The regular-expression case

We consider the case that the second parameter of the $\mathsf{replaceAll}$ function is a regular expression. The decision procedure presented below is a generalisation of those in Section 6 and Section 7.

As in the previous sections, we will again start with the simple situation that $C \equiv x = \mathsf{replaceAll}(y, e_0, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$. For $0 \leq i \leq 3$, let $\mathcal{A}_i = (Q_i, \delta_i, q_{0,i}, F_i)$ be the NFA corresponding to e_i.

Let us first consider the special case $\mathcal{L}(e_0) = \{\varepsilon\}$. Then according to the semantics, for each string $u = a_1 \cdots a_n$, $\mathsf{replaceAll}(u, e_0, v) = va_1v \cdots va_nv$. We can solve the satisfiability of C as follows:

1. Guess a set $T_z \subseteq Q_1 \times Q_1$.

2. Construct $\mathcal{B}_{\mathcal{A}_1, \varepsilon, T_z}$ from $\mathcal{A}_1$ and T_z as follows: For each $(q, q') \in T_z$, add to $\mathcal{A}_1$ a transition (q, ε, q'). Then transform the resulting NFA into one without ε-transitions (which can be done in polynomial time).

3. Decide the nonemptiness of $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1, \varepsilon, T_z})$ and $\mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q'))$.

Next, let us assume that $\mathcal{L}(e_0) \neq \{\varepsilon\}$. For simplicity of presentation, we assume $\varepsilon \notin \mathcal{L}(e_0)$. The case that $\varepsilon \in \mathcal{L}(e_0)$ can be dealt with in a slightly more technical albeit similar way.

Since $\varepsilon \notin \mathcal{L}(e_0)$, we have $q_{0,0} \notin F_0$. In addition, without loss of generality, we assume that there are no incoming transitions for $q_{0,0}$ in $\mathcal{A}_0$.

To check the satisfiability of C, similar to the constant-string case, we construct a parsing automaton $\mathcal{A}_{e_0}$ that parses a string $v \in \Sigma^* e_0 \Sigma^*$ into $v_1 u_1 v_2 u_2 \ldots v_l u_l v_{l+1}$ such that

- for each $j \in [l]$, u_j is the leftmost and longest matching of e_0 in $(v_1 u_1 \ldots v_{j-1} u_{j-1})^{-1} v$,
- $v_{l+1} \notin \Sigma^* e_0 \Sigma^*$.

We will first give an intuitive description of the behaviour of the automaton $\mathcal{A}_{e_0}$. We start with an automaton that can have an infinite number of states and describe the automaton as starting new "threads", i.e., run multiple copies of $\mathcal{A}_0$ on the input word (similar to alternating automata). We also show how this automaton can be implemented using only a finite number of states. Intuitively, in order to search for the leftmost and longest matching of e_0, $\mathcal{A}_{e_0}$ behaves as follows.

- $\mathcal{A}_{e_0}$ has two modes, "left" and "long", which intuitively means searching for the first and last position of the leftmost and longest matching of e_0 respectively.
- When in the "left" mode, $\mathcal{A}_{e_0}$ starts a new thread of $\mathcal{A}_0$ in each position and records *the set of states* of the threads into a vector. In addition, it nondeterministically makes a "leftmost" guessing, that is, guesses that the current position is the first position of the leftmost and longest matching. If it makes such a guessing, it enters the "long" mode, runs the thread started in the current position and searches for the last position of the leftmost and longest matching. Moreover, it stores in a set S the union of the sets of states of all the threads that were started before the current position and continues running these threads to make sure that, in these threads, the final states will not be reached (thus, the current position is indeed the first position of the leftmost and longest matching).
- When in the "long" mode, $\mathcal{A}_{e_0}$ runs a thread of $\mathcal{A}_0$ to search for the last position of the leftmost and longest matching. If the set of states of the thread contains a final state, then $\mathcal{A}_{e_0}$ nondeterministically guesses that the current position is the last position of the leftmost and longest matching. If it makes such a guessing, then it resets the set of states of the thread and starts a new round of searching for the leftmost and longest matching. In addition, it stores the original set of states of the thread into a set S and continues running the thread to make sure that in this thread, the final states will not be reached (thus, the current position is indeed the last position of the leftmost and longest matching).
- Since the length of the vectors of the sets of states of the threads may become unbounded, in order to obtain a finite state automaton, the following trick is applied. Suppose that the vector is $S_1 S_2 \cdots S_n$. For each pair of indices $i, j : i < j$ and each $q \in S_i \cap S_j$, remove q from S_j. The application of this trick is justified by the following arguments: Since q occurs in both S_i and S_j and the thread i was started before the thread j, even if from q a final state can be reached in the future, the position where the thread j was started *cannot* be the first position of the leftmost and longest matching, since the state q is also a state of the thread i and the position where the thread i was started is before the position where the thread i was started.

Before presenting the construction of $\mathcal{A}_{e_0}$ in detail, let us introduce some additional notation.

For $S \subseteq Q_0$ and $a \in \Sigma$, let $\delta_0(S, a)$ denote $\{q' \in Q_0 \mid \exists q \in S.\ (q, a, q') \in \delta_0\}$. For $a \in \Sigma$ and a vector $\rho = S_1 \cdots S_n$ such that $S_i \subseteq Q_0$ for each $i \in [n]$, let $\delta_0(\rho, a) = \delta_0(S_1, a) \cdots \delta_0(S_n, a)$.

For a vector $S_1 \cdots S_n$ such that $S_i \subseteq Q_0$ for each $i \in [n]$, we define $\mathsf{red}(S_1 \cdots S_n)$ inductively:

- If $n = 1$, then $\mathsf{red}(S_1) = S_1$ if $S_1 \neq \emptyset$, and $\mathsf{red}(S_1) = \varepsilon$ otherwise.
- If $n > 1$, then

$$\mathsf{red}(S_1 \cdots S_n) = \begin{cases} \mathsf{red}(S_1 \cdots S_{n-1}) & \text{if } S_n \subseteq \bigcup\limits_{i \in [n-1]} S_i, \\ \mathsf{red}(S_1 \cdots S_{n-1})(S_n \setminus \bigcup\limits_{i \in [n-1]} S_i) & \text{o/w} \end{cases}$$

For instance, $\mathsf{red}(\emptyset\{q\}) = \{q\}$ and

$$\mathsf{red}(\{q_1, q_2\}\{q_1, q_3\}\{q_2, q_4\}) = \mathsf{red}(\{q_1, q_2\}\{q_1, q_3\})\{q_4\} = \mathsf{red}(\{q_1, q_2\})\{q_3\}\{q_4\} = \{q_1, q_2\}\{q_3\}\{q_4\}.$$

We give the formal description of $\mathcal{A}_{e_0} = (Q_{e_0}, \delta_{e_0}, q_{0,e_0}, F_{e_0})$ below. The automaton will contain states of the form (ρ, m, S) where ρ is the vector $S_1 \cdots S_n$ recording the set of states of the threads of $\mathcal{A}_0$. The second component m is either left or long indicating the mode. Finally S is the set of states representing all threads for which final states must not be reached.

- Q_{e_0} comprises
  - the tuples $(\{q_{0,0}\}, \mathsf{left}, S)$ such that $S \subseteq Q_0$,
  - the tuples $(\rho\{q_{0,0}\}, \mathsf{left}, S)$ such that $\rho = S_1 \cdots S_n$ with $n \geq 1$ satisfying that for each $i \in [n]$, $S_i \subseteq Q_0 \setminus \{q_{0,0}\}$, and for each pair of indices $i, j : i < j$, $S_i \cap S_j = \emptyset$, moreover, $S \subseteq Q_0 \setminus F_0$,
  - the tuples (S_1, long, S) such that $S_1 \subseteq Q_0$, $S \subseteq Q_0 \setminus F_0$ and $S_1 \not\subseteq S$;
- $q_{0,e_0} = (\{q_{0,0}\}, \mathsf{left}, \emptyset)$,
- F_{e_0} comprises the states of the form $(-, \mathsf{left}, -) \in Q_{e_0}$,
- δ_{e_0} is defined as follows:
  - (continue left) suppose $(\rho\{q_{0,0}\}, \mathsf{left}, S) \in Q_{e_0}$ such that $\rho = S_1 \cdots S_n$ with $n \geq 0$ ($n = 0$ means that ρ is empty), $a \in \Sigma$, $\big(\bigcup\limits_{j \in [n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)\big) \cap F_0 = \emptyset$, and $\delta_0(S, a) \cap F_0 = \emptyset$, then

    $$((\rho\{q_{0,0}\}, \mathsf{left}, S), a, (\mathsf{red}(\delta_0(\rho\{q_{0,0}\}, a))\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a))) \in \delta_{e_0},$$

    Intuitively, in a state (ρ, left, S), if $\big(\bigcup\limits_{j \in [n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)\big) \cap F_0 = \emptyset$ and $\delta_0(S, a) \cap F_0 = \emptyset$, then $\mathcal{A}_{e_0}$ can choose to stay in the "left" mode. Moreover, no states occur more than once in $\mathsf{red}(\delta_0(\rho\{q_{0,0}\}, a))\{q_{0,0}\}$, since $q_{0,0}$ does not occur in $\mathsf{red}(\delta_0(\rho\{q_{0,0}\}, a))$, (from the assumption that there are no incoming transitions for $q_{0,0}$ in $\mathcal{A}_0$),
  - (start long) suppose $(\rho\{q_{0,0}\}, \mathsf{left}, S) \in Q_{e_0}$ such that $\rho = S_1 \cdots S_n$ with $n \geq 0$, $a \in \Sigma$, $\delta_0(S, a) \cap F_0 = \emptyset$, $\big(\bigcup\limits_{j \in [n]} \delta_0(S_j, a)\big) \cap F_0 = \emptyset$, and $\delta_0(\{q_{0,0}\}, a) \not\subseteq \delta_0(S, a) \cup \bigcup\limits_{j \in [n]} \delta_0(S_j, a)$, then

    $$\left((\rho\{q_{0,0}\}, \mathsf{left}, S), a, \left(\delta_0(\{q_{0,0}\}, a), \mathsf{long}, \delta_0(S, a) \cup \bigcup_{j \in [n]} \delta_0(S_j, a)\right)\right) \in \delta_{e_0}.$$

    Intuitively, from a state $(\rho\{q_{0,0}\}, \mathsf{left}, S)$ with $\rho = S_1 \cdots S_n$, when reading a letter a, if $\big(\bigcup\limits_{j \in [n]} \delta_0(S_j, a)\big) \cap F_0 = \emptyset$, $\delta_0(S, a) \cap F_0 = \emptyset$, and $\delta_0(\{q_{0,0}\}, a) \not\subseteq \delta_0(S, a) \cup \bigcup\limits_{j \in [n]} \delta_0(S_j, a)$, then $\mathcal{A}_{e_0}$ guesses that the current position is the first position of the leftmost and longest matching, it goes to the "long" mode, in addition, it keeps in the first component of the control state only the set of states of the thread started in the current position, and puts the union of the sets of the states of all the threads that have been started before, namely, $\bigcup\limits_{j \in [n]} \delta_0(S_j, a)$, into the third component to guarantee that none of these threads will reach a final state in the future (thus the guessing that the current position is the first position of the leftmost and longest matching is correct),

- (continue long) suppose $(S_1, \mathsf{long}, S) \in Q_{e_0}$, $\delta_0(S,a) \cap F_0 = \emptyset$, and $\delta_0(S_1,a) \not\subseteq \delta_0(S,a)$, then
$$((S_1, \mathsf{long}, S), a, (\delta_0(S_1,a), \mathsf{long}, \delta_0(S,a))) \in \delta_{e_0},$$
intuitively, $\mathcal{A}_{e_0}$ guesses that the current position is not the last position of the leftmost and longest matching and continues the "long" mode,
- (end long) suppose $(S_1, \mathsf{long}, S) \in Q_{e_0}$, $\delta_0(S_1,a) \cap F_0 \neq \emptyset$, and $\delta_0(S,a) \cap F_0 = \emptyset$, then
$$((S_1, \mathsf{long}, S), a, (\{q_{0,0}\}, \mathsf{left}, \delta_0(S,a) \cup \delta_0(S_1,a))) \in \delta_{e_0},$$
intuitively, when $\delta_0(S_1,a) \cap F_0 \neq \emptyset$ and $\delta_0(S,a) \cap F_0 = \emptyset$, $\mathcal{A}_{e_0}$ guesses that the current position is the last position of the leftmost and longest matching, resets the first component to $\{q_{0,0}\}$, goes to the "left" mode, and puts $\delta_0(S_1,a)$ to the third component to guarantee that the current thread will not reach a final state in the future (thus the guessing that the current position is the last position of the leftmost and longest matching is correct).
- (a matches e_0) suppose $(\rho\{q_{0,0}\}, \mathsf{left}, S) \in Q_{e_0}$ such that $\rho = S_1 \cdots S_n$ with $n \geq 0$, $a \in \Sigma$, $\big(\bigcup\limits_{j \in [n]} \delta_0(S_j, a)\big) \cap F_0 = \emptyset$, $\delta_0(\{q_{0,0}\}, a) \cap F_0 \neq \emptyset$, and $\delta_0(S,a) \cap F_0 = \emptyset$, then
$$\left((\rho\{q_{0,0}\}, \mathsf{left}, S), a, \left(\{q_{0,0}\}, \mathsf{left}, \delta_0(S,a) \cup \bigcup_{j \in [n]} \delta_0(S_j,a) \cup \delta_0(\{q_{0,0}\}, a)\right)\right) \in \delta_{e_0},$$
intuitively, from a state $(\rho\{q_{0,0}\}, \mathsf{left}, S)$ with $\rho = S_1 \cdots S_n$, when reading a letter a, if $\big(\bigcup\limits_{j \in [n]} \delta_0(S_j, a)\big) \cap F_0 = \emptyset$, $\delta_0(\{q_{0,0}\}, a) \cap F_0 \neq \emptyset$, and $\delta_0(S,a) \cap F_0 = \emptyset$, then $\mathcal{A}_{e_0}$ guesses that a is simply the leftmost and longest matching of e_0 (e.g. when $e_0 = a$), then it directly goes to the "left" mode (without going to the "long" mode), resets the first component of the control state to $\{q_{0,0}\}$, and puts the union of the sets of the states of all the threads that have been started, including the one started in the current position, namely, $\bigcup\limits_{j \in [n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)$, into the third component to guarantee that none of these threads will reach a final state in the future (where $\bigcup\limits_{j \in [n]} \delta_0(S_j, a)$ is used to validate the leftmost guessing and $\delta_0(\{q_{0,0}\}, a)$ is used to validate the longest guessing).

Let $Q_{\mathsf{left}} = \{(-, \mathsf{left}, -) \in Q_{e_0}\}$, $Q_{\mathsf{long}} = \{(-, \mathsf{long}, -) \in Q_{e_0}\}$, and $v = v_1 u_1 v_2 u_2 \cdots v_l u_l v_{l+1}$ such that u_j is the leftmost and longest matching of e_0 in $(v_1 u_1 \cdots v_{j-1} u_{j-1})^{-1} v$ for each $j \in [l]$, in addition, $v_{l+1} \not\in \Sigma^* e \Sigma^*$. Then there exists a *unique* accepting run r of $\mathcal{A}_{e_0}$ on v such that the state sequence in r is of the form
$$(\{q_{0,0}\}, \mathsf{left}, \emptyset)\ r_1\ (\{q_{0,0}\}, \mathsf{left}, -)\ r_2\ (\{q_{0,0}\}, \mathsf{left}, -) \cdots r_l\ (\{q_{0,0}\}, \mathsf{left}, -)\ r_{l+1},$$
where for each $j \in [l]$, $r_j \in \mathcal{L}((Q_{\mathsf{left}})^* \circ (Q_{\mathsf{long}})^*)$, and $r_{l+1} \in \mathcal{L}((Q_{\mathsf{left}})^*)$. Intuitively, each occurrence of the state subsequence from $\mathcal{L}((Q_{\mathsf{long}})^* \circ (\{q_{0,0}\}, \mathsf{left}, -))$, except the first one, witnesses the *leftmost and longest* matching of e_0 in v from the beginning or after the previous such a matching.

Since in the first component $\rho q_{0,0}$ of each state of $\mathcal{A}_{e_0}$, no states from $\mathcal{A}_0$ occur more than once, it is not hard to see that $|\mathcal{A}_{e_0}|$ is $2^{O(p(|\mathcal{A}_0|))}$ for some polynomial p.

Given $T_z \subseteq Q_1 \times Q_1$, we construct $\mathcal{B}_{\mathcal{A}_1,e_0,T_z}$ by the following three-step procedure.

1. Construct the product of $\mathcal{A}_1$ and $\mathcal{A}_{e_0}$.
2. Remove all transitions associated with states from $Q_1 \times Q_{\mathsf{long}}$, in addition, remove all transitions of the form $((q, (\rho\{q_{0,0}\}, \mathsf{left}, S)), a, (q', (\{q_{0,0}\}, \mathsf{left}, S')))$ such that $\delta_0(q_{0,0}, a) \cap F_0 \neq \emptyset$.
3. For each pair $(q, q') \in T_z$, do the following,
   - for each transition
   $$\left((\rho\{q_{0,0}\}, \mathsf{left}, S), a, \left(\delta_0(\{q_{0,0}\}, a), \mathsf{long}, \delta_0(S, a) \cup \bigcup_{j\in[n]} \delta_0(S_j, a)\right)\right) \in \delta_{e_0},$$
   add a transition
   $$\left((q, (\rho\{q_{0,0}\}, \mathsf{left}, S))\,, a, \left(q, \left(\delta_0(\{q_{0,0}\}, a), \mathsf{long}, \delta_0(S, a) \cup \bigcup_{j\in[n]} \delta_0(S_j, a)\right)\right)\right),$$
   - for each transition
   $$((S_1, \mathsf{long}, S), a, (\delta_0(S_1, a), \mathsf{long}, \delta_0(S, a))) \in \delta_{e_0},$$
   add a transition $((q, (S_1, \mathsf{long}, S)), a, (q, (\delta_0(S_1, a), \mathsf{long}, \delta_0(S, a))))\,,$
   - for each transition
   $$((S_1, \mathsf{long}, S), a, (\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \delta_0(S_1, a))) \in \delta_{e_0},$$
   add a transition $((q, (S_1, \mathsf{long}, S)), a, (q', (\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \delta_0(S_1, a))))$,
   - for each
   $$\left((\rho\{q_{0,0}\}, \mathsf{left}, S), a, \left(\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \bigcup_{j\in[n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)\right)\right) \in \delta_{e_0},$$
   add a transition
   $$\left((q, (\rho\{q_{0,0}\}, \mathsf{left}, S)), a, \left(q', \left(\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \bigcup_{j\in[n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)\right)\right)\right).$$

Since $|\mathcal{A}_{e_0}|$ is $2^{O(p(|\mathcal{A}_0|))}$, it follows that $|\mathcal{B}_{\mathcal{A}_1,e_0,T_z}|$ is $|\mathcal{A}_1| \cdot 2^{O(p(|\mathcal{A}_0|))}$. In addition, since $|\mathcal{A}_0| = O(|e_0|)$, we deduce that $|\mathcal{B}_{\mathcal{A}_1,e_0,T_z}|$ is $|\mathcal{A}_1| \cdot 2^{O(p(|e_0|))}$.

For the more general case that the $\mathsf{SL}[\mathsf{replaceAll}]$ formula C contains more than one occurrence of $\mathsf{replaceAll}(-,-,-)$ terms, we still nondeterministically remove the edges in the dependency graph G_C in a top-down manner and reduce the satisfiability of C to the satisfiability of a collection of regular constraints for source variables.

Complexity In each step of the reduction, suppose the two edges out of x are currently removed, let the two edges be from x to y and z and labeled by (l, e) and (r, e) respectively, then each element of $(\mathcal{T}, \mathcal{P})$ of $\mathcal{E}_i(x)$ may be transformed into an element $(\mathcal{T}', \mathcal{P}')$ of $\mathcal{E}_{i+1}(y)$ such that $|\mathcal{T}'| = |\mathcal{T}| \cdot 2^{O(p(|e|))}$, meanwhile, it may also be transformed into an element $(\mathcal{T}'', \mathcal{P}'')$ of $\mathcal{E}_{i+1}(y)$ such that $\mathcal{T}''$ has the same state space as $\mathcal{T}$. Thus, after the reduction, for each source variable x, $\mathcal{E}(x)$ may contain exponentially many elements, and each of them may have a state space of exponential size. To solve the nonemptiness problem of the intersection of all these regular constraints, the exponential space is sufficient. In addition, if the l-length of G_C is at most one, we can show that for each source variable x, $\mathcal{E}(x)$ corresponds to the intersection of polynomially many regular constraints, where each of them has a state space at most exponential size. To solve the nonemptiness of the intersection of these regular constraints, a polynomial space is sufficient. See Appendix E for a detailed analysis.

9 Undecidable extensions

In this section, we consider the language $\mathsf{SL}[\mathsf{replaceAll}]$ extended with either integer constraints, character constraints, or $\mathsf{IndexOf}$ constraints, and show that each of such extensions leads to undecidability. We will use variables of, in additional to the type Str, the Integer data type Int. The type Str consists of the string variables as in the previous sections. A variable of type Int, usually referred to as an *integer variable*, ranges over the set $\mathbb{N}$ of natural numbers. Recall that, in previous sections, we have used $x, y, z, \ldots$ to denote the variables of Str type. Hereafter we typically use $\mathfrak{l}, \mathfrak{m}, \mathfrak{n}, \ldots$ to denote the variables of Int. The choice of omitting negative integers is for simplicity. Our results can be easily extended to the case where Int includes negative integers.

We begin by defining the kinds of constraints we will use to extend $\mathsf{SL}[\mathsf{replaceAll}]$. First, we describe integer constraints, which express constraints on the length or number of occurrences of symbols in words.

Definition 9.1 (Integer constraints)**.** *An atomic integer constraint over Σ is an expression of the form $a_1 t_1 + \cdots + a_n t_n \leq d$ where $a_1, \cdots, a_n, d \in \mathbb{Z}$ are constant integers (represented in binary), and each* term *t_i is either*

1. *an integer variable $\mathfrak{n}$;*
2. *$|x|$ where x is a string variable; or*
3. *$|x|_a$ where x is string variable and $a \in \Sigma$ is a constant letter.*

Here, $|x|$ and $|x|_a$ denote the length of x and the number of occurrences of a in x, respectively.

An integer constraint *over Σ is a Boolean combination of atomic integer constraints over Σ.*

Character constraints, on the other hand, allow to compare symbols from different strings. The formal definitions are given as follows.

Definition 9.2 (Character constraints)**.** *An* atomic character constraint *over Σ is an equation of the form $x[t_1] = y[t_2]$ where*

- *x and y are either a string variable or a constant string in Σ^*, and*
- *t_1 and t_2 are either integer variables or constant positive integers.*

Here, the interpretation of $x[t_1]$ is the t_1-th letter of x. In case that x does not have the t_1-th letter or *y does not have the t_2-th letter, the constraint $x[t_1] = y[t_2]$ is false by convention.*

A character constraint *over Σ is a Boolean combination of atomic character constraints over Σ.*

We also consider the constraints involving the $\mathsf{IndexOf}$ function.

Definition 9.3 ($\mathsf{IndexOf}$ Constraints)**.** *An atomic* $\mathsf{IndexOf}$ *constraint over Σ is a formula of the form $t\ \mathfrak{o}\ \mathsf{IndexOf}(s_1, s_2)$, where*

- *t is an integer variable, or a positive integer (recall that here we assume that the first position of a string is 1), or the value 0 (denoting that there is no occurrence of s_1 in s_2),*
- *$\mathfrak{o} \in \{\geq, \leq\}$, and*
- *s_1, s_2 are either string variables or constant strings.*

We consider the first-occurrence *semantics of* $\mathsf{IndexOf}$. *More specifically, $t \geq \mathsf{IndexOf}(s_1, s_2)$ holds if t is no less than the first position in s_2 where s_1 occurs, similarly for $t \leq \mathsf{IndexOf}(s_1, s_2)$.*

An $\mathsf{IndexOf}$ *constraint over Σ is a Boolean combination of atomic* $\mathsf{IndexOf}$ *constraints over Σ.*

We will show that the extension of $\mathsf{SL}[\mathsf{replaceAll}]$ with integer constraints entails undecidability, by a reduction from (a variant of) the Hilbert's 10th problem, which is well-known to be undecidable [**?**]. For space reasons, all proofs appear in Appendix G. Intuitively, we want to find a solution to $f(x_1, \cdots, x_n) = g(x_1, \cdots, x_n)$ in the natural numbers, where f and g are polynomials with positive coefficients. We can use the length of string variables over a unary alphabet $\{a\}$ to represent integer variables, addition can be performed with concatenation, and multiplication of x and y with $\mathsf{replaceAll}(x, a, y)$. The integer constraint $|x| = |y|$ asserts the equality of f and g. Note that the use of concatenation can be further dispensed since, by Proposition 3.6, concatenation is expressible by $\mathsf{replaceAll}$ at the price of a slightly extended alphabet.

Theorem 9.4. *For the extension of* $\mathsf{SL}[\mathsf{replaceAll}]$ *with* integer constraints*, the satisfiability problem is undecidable, even if only a single integer constraint of the form $|x| = |y|$ or $|x|_a = |y|_a$ is used.*

Notice that the extension of $\mathsf{SL}[\mathsf{replaceAll}]$ with only one integer constraint of the form $|x| = |y|$ entails undecidability. We remark that the undecidability result here does *not* follow from the undecidability result for the extension of word equations with the letter-counting modalities in [**?**], since the formula by [**?**] is not straight-line.

By utilising a further result on Diophantine equations, we show that for the extension of $\mathsf{SL}[\mathsf{replaceAll}]$ with integer constraints, even if the $\mathsf{SL}[\mathsf{replaceAll}]$ formulae are simple (in the sense that their dependency graphs are of depth at most one), the satisfiability problem is still undecidable (note that no restrictions are put on the integer constraints in this case).

Theorem 9.5. *For the extension of* $\mathsf{SL}[\mathsf{replaceAll}]$ *with integer constraints, even if* $\mathsf{SL}[\mathsf{replaceAll}]$ *formulae are restricted to those whose dependency graphs are of depth at most one, the satisfiability problem is still undecidable.*

By essentially encoding $|x| = |y|$ with *character* or $\mathsf{IndexOf}$ *constraints*, we show:

Proposition 9.6. *For the extension of* $\mathsf{SL}[\mathsf{replaceAll}]$ *with either the character constraints or the* $\mathsf{IndexOf}$ *constraints, the satisfiability problem is undecidable.*

10 Related work

We now discuss some related work. We split our discussion into two categories: (1) theoretical results in terms of decidability and complexity; (2) practical (but generally incomplete) approaches used in string solvers. We emphasise work on replaceAll functions as they are our focus.

Theoretical Results We have discussed in Section 1 works on string constraints with the theory of strings with concatenation. This research programme builds on the question of solving satisfiability of *word equations*, i.e., a string equation $\alpha = \beta$ containing concatenation of string constants and variables. Makanin showed decidability [?], whose upper bound was improved to PSPACE in [?] using a word compression technique. A simpler algorithm was in recent years proposed in [?] using the recompression technique. The best lower bound for this problem is still NP, and closing this complexity gap is a long-standing open problem. Decidability (in fact, the PSPACE upper bound) can be retained in the presence of regular constraints (e.g. see [?]). This can be extended to existential theory of concatenation with regular constraints using the technique of [?]. The replace-all operator cannot be expressed by the concatenation operator alone. For this reason, our decidability of the fragment of SL[replaceAll] cannot be derived from the results from the theory of concatenation alone.

Regarding the extension with length constraints, it is still a long-standing open problem whether word equations with length constraints is decidable, though it is known that letter-counting (e.g. counting the number of occurrences of 0s and 1s separately) yields undecidability [?]. It was shown in [?] that the length constraints (in fact, letter-counting) can be added to the subclass of SL[replaceAll] where the pattern/replacement are constants, while preserving decidability. In contrast, if we allow variables on the replacement parameters of formulas in SL[replaceAll], we can easily encode the Hilbert's 10th problem with length (integer) constraints.

The replaceAll function can be seen as a special, yet expressive, string transformation function, aka string transducer. From this viewpoint, the closest work is [?], which we discuss extensively in the introduction. Here, we discuss two further recent transducer models: streaming string transducers [?] and symbolic transducers [?].

A streaming string transducer is a finite state machine where a finite set of string variables are used to store the intermediate results for output. The replaceAll(x, e, y) term can be modelled by an extension of streaming string transducers *with parameters*, that is, a streaming string transducer which reads an input string (interpreted as the value of x), uses y as a free string variable which is presumed to be read-only, and updates a string variable z, which stores the computation result, by a string term which may involve y. Nevertheless, to the best of our knowledge, this extension of streaming string transducers has not been investigated so far.

Symbolic transducers are an extension of Mealy machine to infinite alphabets by using a variable *cur* to represent the symbol in the current position, and replacing the input and output letters in transitions with unary predicates $\varphi(cur)$ and terms involving *cur* respectively. Symbolic transducers can model replaceAll functions *when the third parameter is a constant.* Inspired by symbolic transducers, it is perhaps an interesting future work to consider an extension of the replaceAll function by allowing predicates as patterns. For instance, one may consider the term replaceAll$(x, cur \equiv 0 \bmod 2, y)$ which replaces every even number in x with y.

Finally, the replaceAll function is related to Array Folds Logic introduced by Daca et al [?]. The authors considered an extension of the quantifier-free theory of integer arrays with counting. The main feature of the logic is the *fold* terms, borrowed from the folding concept in functional programming languages. Intuitively, a fold term applies a function to every element of the array to compute an output. If strings are treated as arrays over a finite domain (the alphabet), the replaceAll function can be seen as a fold term. Nevertheless, the replaceAll function goes beyond the fold terms considered in [?], since it outputs a string (an array), instead of an integer. Therefore, the results in [?] cannot be applied to our setting.

Practical Solvers A large amount of recent work develops practical string solvers including Kaluza [?], Hampi [?], Z3-str [?], CVC4 [?], Stranger [?], Norn [?], S3 and S3P [?, ?], and FAT [?]. Among them, only Stranger, S3, and S3P support replaceAll.

In the Stranger tool, an automata-based approach was provided for symbolic analysis of PHP programs, where two different semantics replaceAll were considered, namely, the leftmost and longest matching as well as the leftmost and shortest matching. Nevertheless, they focused on the abstract-interpretation based analysis of PHP programs and provided an *over-approximation* of all the possible values of the string variables at each program point. Therefore, their string constraint solving algorithm is *not* an exact decision procedure. In contrast, we provided a decision procedure for the straight-line fragment with the rather general replaceAll function, where the pattern parameter can be arbitrary regular expressions and the replacement parameter can be variables. In the latter case, we consider the leftmost and longest semantics mainly for simplicity, and the decision procedure can be adapted to the leftmost and shortest semantics easily.

The S3 and S3P tools also support the replaceAll function, where some progressive searching strategies were provided to deal with the non-termination problem caused by the recursively defined string operations (of which replaceAll is a special case). Nevertheless, the solvers are incomplete as reasoning about unbounded strings defined recursively is in general an undecidable problem.

We conclude with a discussion of related work and future work, focussing on (1) decidability, and (2) heuristics and implementation.

Decidability. *Length constraints* — i.e. an assertion $\varphi((x_1), \ldots, (x_n))$, where φ is a Presburger formula and (x_i) is an integer variable interpreted as the length of the string x_i — have been studied in the context of string solving. It is a major open problem whether the theory of concatenation with length constraints is decidable [?]. Several extensions of this theory are undecidable (e.g. with letter counting [?] and string-number conversion [?]). Several decidable restrictions, however, have been proposed including solved form [?] and acyclicity [?], both of which (like straight-line constraint) impose syntactic restrictions on the way in which string equality can be used in the constraints. It was shown in [?] the decidability of path feasibility for symbolic executions allowing and concatenation in the assignments, and regular constraints, and length constraints in the assertions. If we allow the functions $\mathsf{replaceAll}_p(sub, rep)$ in the assignments (instead of /concatenation) and length constraints as assertions, path feasibility becomes undecidable [?]. This also implies undecidability of allowing length constraints in our constraint language with parametric transducers. Fortunately, decidability can be easily recovered in some cases. One such case is when the length constraints φ has only one string variable x_1, e.g., $(x_1) > 7$. In this case, $\varphi((x_1))$ can be turned into a regular constraint $x_1 \in L$ for some L. [This is because the set of integer solutions is effectively a finite union of arithmetic progressions $\bigcup_{i=1}^{n}(a_i + b_i)$ (where $a_i + b_i := \{a_i + b_i n : n \in\}$), and each $(x_i) \in (a_i + b_i)$ is equivalent to the regular constraint $x_i \in^{a_i} ({}^{b_i})^*$.]

The complexity of the theory of concatenation with regular constraints is known to be PSPACE-complete [?, ?]. The complexity of the straight-line logic with concatenation, finite transducers, and regular constraints is EXPSPACE-complete [?]. The same complexity holds when we swap finite transducers with replaceall [?]. For functions, our logic strictly subsumes these two logics (e.g. since it can also express string reverse) and has precisely the same complexity EXPSPACE. One future research avenue is to identify *larger subclasses* of constraints with parametric transducers that are still solvable in the same complexity class.

In this paper, we have combined two powerful formalisms (two-way finite transducers and replaceall) into a single formalism. Since we are considering only two-way transducers that define functions, they are equivalent to two-way deterministic finite transducers, streaming transducers, and MSO-definable transductions [?, ?, ?]. On the other hand, one-way transducers that define functions are strictly more expressive than deterministic one-way transducers [?].

Heuristics and implementation. Theoretical algorithms (e.g. Makanin's algorithm [?]) typically do not directly lead to practical solvers. At the same time, the classes of constraints that are required in practice sometimes require string operations that are not covered by decidable string constraint languages. For these reasons, there is a large amount of work on heuristics for developing practical (often incomplete) string solvers, e.g., [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. Some practical heuristics include bounding string lengths (e.g. [?, ?, ?]), induction, overapproximations [?, ?], interpolation [?], and flat automata [?], to name a few. Focusing on semi-algorithms also allows highly expressive (but undecidable) string constraint languages, e.g., recursively defined functions [?, ?]. Recently, the study of decidability of string constraints have also resulted in automata-theoretic algorithms that are amenable to implementation, e.g., acyclic logic with concatenation, regular constraints, and length constraints [?], and straight-line logic with finite transducers (or replaceall), concatenation, and regular constraints [?, ?, ?]. We leave the development of practical heuristics for our more expressive constraint language for future work.

We mention some interesting benchmarks that are available for string constraints. The first one is Kaluza benchmarks [?], which contain string constraints with concatenation, regular constraints, and length constraints. It was shown in [?] that almost all the constraints are already in *solved form* (in particular, also straight-line). Some of these length constraints are also expressible as regular constraints. The second is SLOG benchmarks [?], which contain string constraints with concatenation, (a restricted class of) replaceall, and regular constraints. The third is SLOTH benchmarks [?], which contains string constraints with concatenation, finite transducers, and regular constraints. Most of these benchmarking examples are expressible in our decidable constraint language.

11 Conclusion

We have initiated a systematic investigation of the decidability of the satisfiability problem for the straight-line fragments of string constraints involving the replaceAll function and regular constraints. The straight-line restriction is known to be appropriate for applications in symbolic execution of string-manipulating programs [?]. Our main result is a decision procedure for a large fragment of the logic, wherein the pattern parameters are regular expressions (which covers a large proportion of the usage of the replaceAll function in practice). Concatenation is obtained for free since concatenation can be easily expressed in this fragment. We have shown that the decidability of this fragment cannot be substantially extended. This is achieved by showing that if either (1) the pattern parameters are allowed to be variables, or (2) the length constraints are incorporated in the fragment, then we get the undecidability. Our work clarified important fundamental issues surrounding the replaceAll functions in string constraint solving and provided a novel decision procedure which paved a way to a string solver that is able to fully support the replaceAll function. This would be the most immediate future work.

Acknowledgments T. Chen is supported by the Australian Research Council under Grant No. DP160101652 and the Engineering and Physical Sciences Research Council under Grant No. EP/P00430X/1. M. Hague is supported by the Engineering and Physical Sciences Research Council under Grant No. EP/K009907/1. A. Lin is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no 759969). Z. Wu is supported by the National Natural Science Foundation of China under Grant No. 61472474 and Grant No. 61572478, and the INRIA-CAS joint research project "Verification, Interaction, and Proofs".

Supplementary Material

"What Is Decidable about String Constraints with the ReplaceAll Function"

We provide below proofs and examples that were omitted from the main text due to space constraints.

A Proof of Proposition 4.1

We recall Proposition 4.1 and then give its proof.

PROPOSITION 4.1 *The satisfiability problem of* $\mathsf{SL}[\mathsf{replaceAll}]$ *is undecidable, if the second parameters of the* $\mathsf{replaceAll}$ *terms are allowed to be variables.*

Proof. We reduce from the Post Correspondence Problem (PCP). Recall that the input of the problem consists of two finite lists $\alpha_1, \ldots, \alpha_N$ and $\beta_1, \ldots, \beta_N$ of nonempty strings over Σ. A solution to this problem is a sequence of indices $(i_k)_{1 \leq k \leq K}$ with $K \geq 1$ and $1 \leq i_k \leq N$ for all k, such that $\alpha_{i_1} \ldots \alpha_{i_K} = \beta_{i_1} \ldots \beta_{i_K}$. The PCP problem is to decide whether such a solution exists or not.

Without loss of generality, suppose $\Sigma \cap [N] = \emptyset$ and $\$ \notin \Sigma \cup [N]$. Let $\Sigma' = \Sigma \cup [N] \cup \{\$\}$. We will construct an $\mathsf{SL}[\mathsf{replaceAll}]$ formula C over Σ' such that the PCP instance has a solution iff C is satisfiable. To this end, the formula C utilises the capability that the second parameter of the $\mathsf{replaceAll}$ terms may be variables.

Let $x_1, \cdots, x_N, y_1, \cdots, y_N, z$ be mutually distinct string variables. Then the formula $C = \varphi \wedge \psi$, where

$$\begin{aligned} \varphi &= \bigwedge_{i \in [N]} (x_i = \mathsf{replaceAll}(x_{i-1}, i, \alpha_i) \wedge y_i = \mathsf{replaceAll}(y_{i-1}, i, \beta_i)) \wedge z = \mathsf{replaceAll}(x_N, y_N, \$), \\ \psi &= x_0 \in (1 + \cdots + N)^+ \wedge z \in \$. \end{aligned}$$

It is not hard to see that φ is a straight-line relational constraint, thus C is an $\mathsf{SL}[\mathsf{replaceAll}]$ formula. Note that in $\mathsf{replaceAll}(x_N, y_N, \$)$, the second parameter is a variable. We show that C is satisfiable iff the PCP instance has a solution: C is satisfiable iff there is a string $i_1 \cdots i_K \in \mathcal{L}((1 + \cdots + N)^+)$ such that when x_0 is assigned with $i_1 \cdots i_K$, the value of z is \$. Since $z = \mathsf{replaceAll}(x_N, y_N, \$)$ and $x_N, y_N \in \Sigma^+$, we know that z is \$ iff the values of x_N and y_N are the same. Therefore, C is satisfiable iff there is a string $i_1 \cdots i_K \in \mathcal{L}((1 + \cdots + N)^+)$ such that when x_0 is assigned with $i_1 \cdots i_K$, the values of x_N and y_N are the same. Therefore, C is satisfiable iff there is a sequence of indices $i_1 \cdots i_K$ such that $\alpha_{i_1} \cdots \alpha_{i_K} = \beta_{i_1} \cdots \beta_{i_K}$, that is, the PCP instance has a solution. □

B Section 6: The Correctness of the decision procedure

We argue that the procedure in Section 6.2 is correct. Note that Proposition 6.1 removed a single $\mathsf{replaceAll}(-, -, -)$ to obtain only regular constraints. Each step of our decision procedure effectively eliminates a $\mathsf{replaceAll}(-, -, -)$. Similar to Proposition 6.1, each step maintains the satisfiability from the preceding step.

In more detail, from each G_i we can define a constraint C_i. This constraint is a conjunction of the following atomic constraints.

- For each variable x such that $(x, (\mathsf{l}, a), y)$ and $(x, (\mathsf{r}, a), z)$ are the edges in G_i, we assert in C_i that $x = \mathsf{replaceAll}(y, a, z)$.
- In addition, for each variable x such that $\mathcal{E}_i(x)$ is not empty, moreover, *either x is a source variable in G_C (not G_i) or there are (incoming or outgoing) edges connected to x in G_i*, let $e_i(x)$ be the regular expression equivalent to the conjunction of all constraints in $\mathcal{E}_i(x)$ (Note that the conjunction of multiple regular expressions still defines a regular language). We assert in C_i that $x \in e_i(x)$. Note that if x is not a source variable in G_C and there are no edges connected to x in G_i, then the regular constraints in $\mathcal{E}_i(x)$ are not included into C_i.

It is immediate that C_0 is equivalent to C. We require the following proposition, which gives us the correctness of the decision procedure by induction. Note that the final C_i when exiting the loop will be a conjunction of regular constraints on the source variables.

Proposition B.1. *For each i, let the l-edge and the r-edge from x to y and z respectively be the two edges removed from G_i to construct G_{i+1}. Then C_i is satisfiable iff there are sets $T_{j,z}$ such that C_{i+1} is satisfiable.*

We can see the above proposition by observing that, in each step, C_i is of the form

$$x = \mathsf{replaceAll}(y, a, z) \wedge x \in e_i(x) \wedge y \in e_i(y) \wedge z \in e_i(z) \wedge C'$$

where C' does not contain x, and C_{i+1} is of the form

$$y \in e_{i+1}(y) \wedge z \in e_{i+1}(z) \wedge C' .$$

Note that C' remains unchanged since only the two edges leaving x are removed from G_i and $\mathcal{E}_{i+1}(x') = \mathcal{E}_i(x')$ for all x' distinct from x, y, and z. First assume $y \neq z$. Supposing C_i is satisfiable, an argument similar to that of Proposition 6.1 shows that there are sets $T_{j,z}$ such that the same values of y and z also satisfy $e_{i+1}(y)$ and $e_{i+1}(z)$. Since C' is unchanged, all x' distinct from x, y, and z can also keep the same value. Thus, C_{i+1} is also satisfiable. In the other direction, suppose that there are sets $T_{j,z}$ such that C_{i+1} is satisfiable. Take a satisfying assignment to C_{i+1}. From the assignment to y and z we obtain as in Proposition 6.1 an assignment to x that satisfies $\mathsf{replaceAll}(y, a, z) \wedge x \in e_i(x)$. Furthermore, the assignments for y and z also satisfy $e_i(y)$ and $e_i(z)$ since $\mathcal{E}_i(y)$ and $\mathcal{E}_i(z)$ are subsets of $\mathcal{E}_{i+1}(y)$ and $\mathcal{E}_{i+1}(z)$. Finally, since C' is unchanged, the assignments to all other variables also transfer, giving us a satisfying assignment to C_i as required. In the case where $y = z$, the arguments proceed analogously to the case $y \neq z$.

C The product automaton $\mathcal{A}_1 \times \mathcal{A}_u$for $u = 010$

In Figure 7 we give the product automaton $\mathcal{A}_1 \times \mathcal{A}_u$ for $u = 010$. This is a straightforward product construction, but may be useful for reference when understanding Figure 6 which shows the automaton $\mathcal{B}_{\mathcal{A}_1,u,T_z}$ which is derived from the product.

D Complexity analysis in Section 7

We provide a more detailed analysis of the complexity of the algorithm for the constant string case, described in Section 7. A summary of this argument already appears in Section 7.

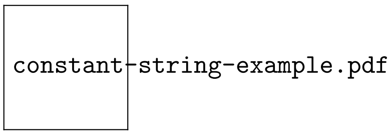

Figure 7: The NFA $\mathcal{A}_1 \times \mathcal{A}_u$ for $u = 010$

When constructing G_{i+1} from G_i, suppose the two edges from x to y and z respectively are currently removed, let the labels of the two edges be (l, u) and (r, u) respectively, then each element $(\mathcal{T}, \mathcal{P})$ of $\mathcal{E}_i(x)$ may be transformed into an element $(\mathcal{T}', \mathcal{P}')$ of $\mathcal{E}_{i+1}(y)$ such that $|\mathcal{T}'| = O(|u||\mathcal{T}|)$, meanwhile, it may also be transformed into an element $(\mathcal{T}'', \mathcal{P}'')$ of $\mathcal{E}_{i+1}(z)$ such that $\mathcal{T}''$ has the same state space as $\mathcal{T}$. Thus, for each source variable x, $\mathcal{E}(x)$ contains at most exponentially many elements, and each of them may have a state space of at most exponential size. For instance, for a path from x' to x where the constant strings $u_1, \cdots, u_n$ occur in the labels of edges, an element $(\mathcal{T}, \mathcal{P}) \in \mathcal{E}_0(x')$ may induce an element $(\mathcal{T}', \mathcal{P}')$ of $\mathcal{E}(x)$ such that $|\mathcal{T}'| \leq |\mathcal{T}||u_1| \cdots |u_n|$, which is exponential in the worst case. To solve the nonemptiness problem of the intersection of all these regular constraints, the exponential space is sufficient. Consequently, in this case, we still obtain an EXPSPACE upper-bound.

Let us now consider the special situation that the l-length of G_C is bounded by a constant c. Since $\mathsf{Idx}_{\mathsf{dmd}}(G_C) \leq \mathsf{Len}_{\mathsf{lft}}(G_C)$, we know that $\mathsf{Idx}_{\mathsf{dmd}}(G_C)$ is also bounded by c. Therefore, according to Proposition 4.5, there are at most polynomially different paths in G_C, we deduce that for each source variable x, $\mathcal{E}(x)$ contains at most polynomially many elements. In addition, since the number of l-edges in each path is bounded by c, during the execution of the decision procedure, the number of times when $(\mathcal{T}, \mathcal{P})$ of $\mathcal{E}_i(x)$ may be transformed into an element $(\mathcal{T}', \mathcal{P}')$ of $\mathcal{E}_{i+1}(y)$ such that $|\mathcal{T}'| = O(|u||\mathcal{T}|)$ is bounded by c. Therefore, for each source variable x and each element $(\mathcal{T}'', \mathcal{P}'')$ in $\mathcal{E}(x)$, $|\mathcal{T}''|$ is at most polynomial in the size of C. We then conclude that for each source variable x, $\mathcal{E}(x)$ corresponds to the intersection of polynomially many regular constraints such that each of them has a state space of polynomial size. Therefore, the nonemptiness of the intersection of all the regular constraints in $\mathcal{E}(x)$ can be solved in polynomial space. In this situation, we obtain a PSPACE upper-bound.

E Complexity analysis in Section 8

We provide a more detailed analysis of the complexity of the algorithm for the regular-expression case, described in Section 8. A summary of this argument already appears in Section 8.

In each step of the reduction, suppose the two edges out of x are currently removed, let the two edges be from x to y and z and labeled by (l, e) and (r, e) respectively, then each element of $(\mathcal{T}, \mathcal{P})$ of $\mathcal{E}_i(x)$ may be transformed into an element $(\mathcal{T}', \mathcal{P}')$ of $\mathcal{E}_{i+1}(y)$ such that $|\mathcal{T}'| = |\mathcal{T}| \cdot 2^{O(p(|e|))}$, meanwhile, it may also be transformed into an element $(\mathcal{T}'', \mathcal{P}'')$ of $\mathcal{E}_{i+1}(y)$ such that $\mathcal{T}''$ has the same state space as $\mathcal{T}$. Thus, after the reduction, for each source variable x, $\mathcal{E}(x)$ may contain exponentially many elements, and each of them may have a state space of exponential size, more precisely, if we start from a vertex x without predecessors, with an element $(\mathcal{T}, \mathcal{P})$ in $\mathcal{E}_0(x)$, and go to a source variable y through a path where k edges have been traversed and removed, let $e_1, \cdots, e_k$ be the regular expressions occurring in the labels of these edges, then the resulting element in $\mathcal{E}(y)$ has a state space of size $|\mathcal{T}| \cdot 2^{O(p(|e_1|))} \cdot 2^{O(p(|e_2|))} \cdot \cdots \cdot 2^{O(p(|e_k|))}$ in the worst case. To solve the nonemptiness problem of the intersection of all these regular constraints, the exponential space is sufficient. Consequently, for the most general case of regular expressions, we still obtain an EXPSPACE upper-bound.

On the other hand, for the situation that the l-length of G_C is at most one, we wan to show that the algorithm runs in polynomial space. Suppose the l-length of G_C is at most one. Then the diamond index of G_C is at most one as well. According to Proposition 4.5, there are only polynomially many paths in G_C. Nevertheless, for each source variable x, $\mathcal{E}(x)$ may contain an element $(\mathcal{T}, \mathcal{P})$ such that $|\mathcal{T}|$ is exponential. Since $|\mathcal{P}|$ may be exponential, $(\mathcal{T}, \mathcal{P})$ may correspond to the intersection of exponentially many regular constraints. However, we can show that $|\mathcal{P}|$ is at most polynomial, as a result of the fact that the l-length of G_C is at most one. The arguments proceed as follows: Suppose two edges from x to y, z respectively are removed, and an element $(\mathcal{T}', \mathcal{P}')$ of $\mathcal{E}_{i+1}(y)$ such that $|\mathcal{T}'|$ is exponential and $|\mathcal{P}'|$ is polynomial, is generated from an element of $(\mathcal{T}, \mathcal{P})$ of $\mathcal{E}_i(x)$. Then y must be a source variable in G_C. Otherwise, there is an l-edge out of y and the l-length of G_C is at least two, a contradiction. Therefore, y is a source variable in G_C, $(\mathcal{T}', \mathcal{P}')$ will not be used to generate the regular constraints for the other variables. In other words, y is a source variable in G_C, and $(\mathcal{T}', \mathcal{P}') \in \mathcal{E}(y)$ with $|\mathcal{P}'|$ polynomial. We then conclude that for each source variable x, $|\mathcal{E}(x)|$ is at most polynomial in the size of C and for each element $(\mathcal{T}, \mathcal{P}) \in \mathcal{E}(x)$, $|\mathcal{P}|$ is polynomial in the size of C. Therefore, for each source variable x, $\mathcal{E}(x)$ corresponds to the intersection of polynomially many regular constraints, where each of them has a state space at most exponential size. To solve the nonemptiness of the intersection of these regular constraints, the polynomial space is sufficient. We obtain a PSPACE upper-bound for the situation that the l-length of G_C is at most one.

F Examples in Section 8

Due to space constraints, we did not provide examples of the decision procedure for the regular-expression case. We provide some examples here.

Example F.1. *Let $e_0 = 0^*01(1^*+0^*)$. Then $\mathcal{A}_0$ and $\mathcal{A}_{e_0}$ are illustrated in Figure 8, where* sleft *and* slong *are the abbreviations of* left *and* long *respectively. Let us use the state $(\{q_{0,1}\}\{q_{0,0}\}, \mathsf{sleft}, \emptyset)$ to illustrate the construction. Since $\big(\delta_0(\{q_{0,1}\}, 0) \cup \delta_0(\{q_{0,0}\}, 0)\big) \cap F_0 = \{q_{0,1}\} \cap F_0 = \emptyset$, $\delta_0(\emptyset, 0) \cap F_0 = \emptyset$, and $\mathsf{red}(\delta_0(\{q_{0,1}\}, 0)\delta_0(\{q_{0,0}\}, 0)) = \{q_{0,1}\}$, we deduce that the transition*

$$((\{q_{0,1}\}\{q_{0,0}\}, \mathsf{sleft}, \emptyset), 0, (\{q_{0,1}\}\{q_{0,0}\}, \mathsf{sleft}, \emptyset)) \in \delta_{e_0} \ .$$

*On the other hand, it is impossible to go from the state $(\{q_{0,1}\}\{q_{0,0}\}, \mathsf{sleft}, \emptyset)$ to the "*long*" mode. This is due to the fact that $\delta_0(\{q_{0,0}\}, 0) = \{q_{0,1}\} \subseteq \delta_0(\{q_{0,1}\}, 0) = \{q_{0,1}\}$. In addition, there are no 1-transitions out of $(\{q_{0,1}\}\{q_{0,0}\}, \mathsf{sleft}, \emptyset)$. This is due to the fact that $\delta_0(\{q_{0,1}\}, 1) \cap F_0 = \{q_{0,2}, q_{0,3}\} \cap F_0 \neq \emptyset$.*

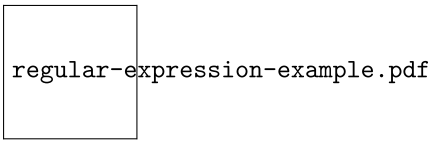

Figure 8: The NFA $\mathcal{A}_0$ and $\mathcal{A}_{e_0}$ for $e_0 = 0^*01(1^* + 0^*)$

Example F.2. *Let $C \equiv x = \mathsf{replaceAll}(y, e_0, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$, where e_1, e_2, e_3 are as in Example 6.2 (cf. Figure 2) and e_0 is as in Example F.1 (cf. Figure 8). Suppose $T_z =$*

$\{(q_0, q_0), (q_1, q_2)\}$. *Then the NFA* $\mathcal{B}_{\mathcal{A}_1,e_0,T_z}$ *is as illustrated in Figure 9, where the thick edges denote the added transitions. Let us use the state* $(q_1, (\{q_{0,0}\}, \mathsf{left}, \emptyset))$ *to exemplify the construction. The transition* $((q_1, (\{q_{0,0}\}, \mathsf{left}, \emptyset)), 1, (q_2, (\{q_{0,0}\}, \mathsf{left}, \emptyset)))$ *is in* $\mathcal{A}_1 \times \mathcal{A}_{e_0}$. *Since* $\delta_0(q_{0,0}, 1) \cap F_0 = \emptyset$, *this transition is not removed and is thus in* $\mathcal{B}_{\mathcal{A}_1,e_0,T_z}$. *On the other hand, since there are no* 0*-transitions out of* q_1 *in* $\mathcal{A}_1$, *there are no* 0*-transitions from* $(q_1, (\{q_{0,0}\}, \mathsf{left}, \emptyset))$ *to some state from* Q_{left} *in* $\mathcal{B}_{\mathcal{A}_1,e_0,T_z}$. *Moreover, because* $((\{q_{0,0}\}, \mathsf{left}, \emptyset), 0, (\{q_{0,1}\}, \mathsf{long}, \emptyset)) \in \delta_{e_0}$ *and* $(q_1, q_2) \in T_z$, *the transition* $((q_1, (\{q_{0,0}\}, \mathsf{left}, \emptyset)), 0, (q_1, (\{q_{0,1}\}, \mathsf{long}, \emptyset)))$ *is added. One may also note that there are no 0-transitions from* $(q_2, (\{q_{0,0}\}, \mathsf{left}, \emptyset))$ *to the state* $(q_2, (\{q_{0,1}\}, \mathsf{long}, \emptyset))$, *because there are no pairs* $(q2, -) \in T_z$. *It is not hard to see that* $010101 \in \mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,e_0,T_z})$. *In addition,* $10 \in \mathcal{L}(\mathcal{A}_3) \cap \mathcal{L}(\mathcal{A}_1(q_0, q_0)) \cap \mathcal{L}(\mathcal{A}_1(q_1, q_2))$. *Let* y *be* 010101 *and* z *be* 10. *Then* x *takes the value* $\mathsf{replaceAll}(010101, e_0, 10) = 10 \cdot \mathsf{replaceAll}(101, e_0, 10) = 10110$, *which is accepted by* $\mathcal{A}_1$. *Therefore,* C *is satisfiable.*

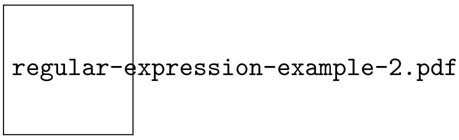

Figure 9: The NFA $\mathcal{B}_{\mathcal{A}_1,e_0,T_z}$

G Undecidability Proofs for Section 9

We provide the proofs of the theorems and propositions in Section 9 which show the undecidability of various extensions of our string constraints.

G.1 Proof of Theorem 9.4

We begin with the first Theorem, which is recalled below.

Proposition 9.4 *For the extension of* $\mathsf{SL}[\mathsf{replaceAll}]$ *with* integer constraints*, the satisfiability problem is undecidable, even if only a single integer constraint* $|x| = |y|$ *is used.*

Proof. The basic idea of the reduction is to simulate the two polynomials $f(x_1, \cdots, x_n)$ and $g(x_1, \cdots, x_n)$, where $x_1, \cdots, x_n$ range over the set of natural numbers, with two $\mathsf{SL}[\circ, \mathsf{replaceAll}]$ formulae C_f, C_g over a unary alphabet $\{a\}$, with the output string variables y_f, y_g respectively, and simulate the equality $f(x_1, \cdots, x_n) = g(x_1, \cdots, x_n)$ with the integer constraint $|y_f| = |y_g|$ (which is equivalent to $y_f = y_g$, since y_f, y_g represent strings over the unary alphabet $\{a\}$).

A polynomial $f(x_1, \cdots, x_n)$ or $g(x_1, \cdots, x_n)$ where $x_1, \cdots, x_n$ range over the set of natural numbers, can be simulated by an $\mathsf{SL}[\circ, \mathsf{replaceAll}]$ formula over an unary alphabet $\{a\}$ as follows: The natural numbers are represented by the strings over the alphabet $\{a\}$. A string variable is introduced for each subexpression of $f(x_1, \cdots, x_n)$. The numerical addition operator $+$ is simulated by the string operation $\circ$ and the multiplication operator $*$ is simulated by $\mathsf{replaceAll}$. Since it is easy to figure out how the simulation proceeds, we will only use an example to illustrate it and omit the details here. Let us consider $f(x_1, x_2) = x_1^2 + 2x_1x_2 + 5$. By abusing the notation, we also use x_1, x_2 as string variables in the simulation. We will introduce a string variable for each subexpression in $f(x_1, x_2)$, namely the variables $y_{x_1^2}, y_{x_1x_2}, y_{2x_1x_2}, y_{x_1^2+2x_1x_2}, y_{f(x_1,x_2)}$.

Then $f(x_1, x_2)$ is simulated by the $\mathsf{SL}[\circ, \mathsf{replaceAll}]$ formula

$$\begin{aligned} C_f \quad \equiv \quad & y_{x_1^2} = \mathsf{replaceAll}(x_1, a, x_1) \ \wedge y_{x_1x_2} = \mathsf{replaceAll}(x_1, a, x_2) \wedge \\ & y_{2x_1x_2} = \mathsf{replaceAll}(aa, a, y_{x_1x_2}) \ \wedge y_{x_1^2+2x_1x_2} = y_{x_1^2} \circ y_{2x_1x_2} \ \wedge \\ & y_{f(x_1,x_2)} = y_{x_1^2+2x_1x_2} \circ aaaaa \ \wedge x_1 \in a^* \ \wedge x_2 \in a^*. \end{aligned}$$

Then according to Proposition 3.6, C_f, C_g can be turned into equivalent $\mathsf{SL}[\mathsf{replaceAll}]$ formula C'_f, C'_g by introducing fresh letters.

Since C'_f and C'_g share only source variables $x_1, \cdots, x_n$, we know that $C'_f \wedge C'_g$ is still an $\mathsf{SL}[\mathsf{replaceAll}]$ formula. From the construction of C'_f, C'_g, it is evident that for every pair of polynomials $f(x_1, \cdots, x_n)$ and $g(x_1, \cdots, x_n)$, $f(x_1, \cdots, x_n) = g(x_1, \cdots, x_n)$ has a solution in natural numbers iff $C'_f \wedge C'_g \wedge |y_f| = |y_g|$ is satisfiable. The proof is complete. □

G.2 Undecidability of Depth-1 Dependency Graph

We recall the undecidability of a depth-1 dependency graph before providing the proof below.

Theorem9.5 *For the extension of* $\mathsf{SL}[\mathsf{replaceAll}]$ *with integer constraints, even if* $\mathsf{SL}[\mathsf{replaceAll}]$ *formulae are restricted to those whose dependency graphs are of depth at most one, the satisfiability problem is still undecidable.*

A *linear polynomial* (resp. quadratic polynomial) is a polynomial with degree at most one (resp. with degree at most two) where each coefficient is an integer.

Theorem G.1 ([?])**.** *The following problem is undecidable: Determine whether a system of equations of the following form has a solution in natural numbers,*

$$\begin{array}{ll} A_i = B_i, & i = 1, \cdots, k, \\ y_i F_i = G_i \wedge y_i H_i = I_i, & i = 1, \cdots, m, \end{array}$$

where A_i, B_i, F_i, G_i are linear polynomials on the variables $x_1, \cdots, x_n$ (Note that each variable y_i occurs in exactly two quadratic equations).

We can get a reduction from the problem in Theorem G.1 to the satisfiability of the extension of $\mathsf{SL}[\mathsf{replaceAll}]$ with integer constraints as follows: For each monomial $y_i x_j$ in the quadratic polynomials, we use an $\mathsf{SL}[\mathsf{replaceAll}]$ formula $z_{y_i x_j} = \mathsf{replaceAll}(y_i, a, x_j)$ to simulate $y_i x_j$, where $z_{y_i x_j}$ are freshly introduced string variables. Since each equation $y_i F_i = G_i$ or $y_i H_i = I_i$ can be seen as a linear combination of the terms $y_i x_j$ and x_j for $i \in [m]$ and $j \in [n]$, we can replace each variable x_j with $|x_j|$, and each term $y_i x_j$ with $|z_{y_i x_j}|$, thus transform them into the (linear) integer constraints $F'_i = G'_i$ or $H'_i = I'_i$. Similarly, after replacing each variable x_j with $|x_j|$, we transform each equation $A_i = B_i$ into an integer constraint $A'_i = B'_i$. Therefore, we get a formula

$$\begin{aligned} \bigwedge_{i \in [m], j \in [n]} z_{y_i x_j} = \mathsf{replaceAll}(y_i, a, x_j) \wedge \bigwedge_{i \in [m]} y_i \in a^* \ \wedge \bigwedge_{j \in [n]} x_j \in a^* \ \wedge \\ \bigwedge_{i \in [k]} A'_i = B'_i \wedge \bigwedge_{i \in [m]} (F'_i = G'_i \wedge H'_i = I'_i), \end{aligned}$$

where the dependency graph of the $\mathsf{SL}[\mathsf{replaceAll}]$ subformula is of depth at most one.

G.3 Undecidability of the Character Constraints

We provide part of the proof of Proposition 9.6, in particular, we show the undecidability of character constraints.

Proposition G.2. *For the extension of* $\mathsf{SL}[\mathsf{replaceAll}]$ *with character constraints, the satisfiability problem is undecidable.*

The arguments for Proposition G.2 proceed as follows. Recall that in the proof of Theorem 9.4, we get a formula $C_f \wedge C_g \wedge |y_f| = |y_g|$ such that $f(x_1, \cdots, x_n) = g(x_1, \cdots, x_n)$ has a solution in natural numbers iff $C_f \wedge C_g \wedge |y_f| = |y_g|$ is satisfiable. Let $\$ \neq a$. Suppose $z_f = y_f \circ \$$, and $z_g = y_g \circ \$$. Then $|y_f| = |y_g|$ can be captured by $z_f[\mathfrak{n}] = \$[1] \wedge z_g[\mathfrak{n}] = \$[1]$, where $\mathfrak{n}$ is a variable of type Int. More precisely, we have

$$C_f \wedge C_g \wedge |y_f| = |y_g| \text{ is satisfiable}$$
$$\text{iff}$$
$$C_f \wedge C_g \wedge z_f = y_f \circ \$ \wedge z_g = y_g \circ \$ \wedge z_f[\mathfrak{n}] = \$[1] \wedge z_g[\mathfrak{n}] = \$[1] \text{ is satisfiable.}$$

Therefore, we get a reduction from Hilbert's tenth problem to the satisfiability problem for the extension of $\mathsf{SL}[\mathsf{replaceAll}]$ with character constraints.

G.4 Undecidability of the IndexOf Constraints

We provide the final part of the proof of Proposition 9.6, in particular, we show the undecidability of IndexOf constraints.

Proposition G.3. *For the extension of* $\mathsf{SL}[\mathsf{replaceAll}]$ *with the* IndexOf *constraints, the satisfiability problem is undecidable.*

Proposition G.2 follows from the following observation and Theorem 9.4: For any two string variables x, y over a unary alphabet, $1 = \mathsf{IndexOf}(x, y)$ iff x is a prefix of y. Therefore, $|x| = |y|$ iff $1 = \mathsf{IndexOf}(x, y) \wedge 1 = \mathsf{IndexOf}(y, x)$. This implies that in the proof of Theorem 9.4, we can replace $|y_f| = |y_g|$ with $1 = \mathsf{IndexOf}(y_f, y_g) \wedge 1 = \mathsf{IndexOf}(y_g, y_f)$ and get a reduction from Hilbert's tenth problem to the satisfiability problem for the extension of $\mathsf{SL}[\mathsf{replaceAll}]$ with the IndexOf constraints. Note that $=$ can be simulated as a conjunction of $\leq$ and $\geq$.