Solving String Constraints With Regex-Dependent Functions Through Transducers With Priorities And Variables

Taolue Chen[1]*, Alejandro Flores-Lamas[2], Matthew Hague[2]†, Zhilei Han[3]‡, Denghang Hu[4], Shuanglong Kan[5], Anthony W. Lin[5]§, Philipp Rümmer[6]¶, and Zhilin Wu[4]

[1] Birkbeck, University of London, United Kingdom
[2] Royal Holloway, University of London, United Kingdom
[3] School of Software, Tsinghua University, China
[4] State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, China
[5] TU Kaiserslautern, Germany
[6] Uppsala University, Sweden

Abstract

Regular expressions are a classical concept in formal language theory. Regular expressions in programming languages (RegEx) such as JavaScript, feature non-standard semantics of operators (e.g. greedy/lazy Kleene star), as well as additional features such as capturing groups and references. While symbolic execution of programs containing RegExes appeals to string solvers natively supporting important features of RegEx, such a string solver is hitherto missing. In this paper, we propose the first string theory and string solver that natively provides such support. The key idea of our string solver is to introduce a new automata model, called *prioritized streaming string transducers* (PSST), to formalize the semantics of RegEx-dependent string functions. PSSTs combine *priorities*, which have previously been introduced in prioritized finite-state automata to capture greedy/lazy semantics, with *string variables* as in streaming string transducers to model capturing groups. We validate the consistency of the formal semantics with the actual JavaScript semantics by extensive experiments. Furthermore, to solve the string constraints, we show that PSSTs enjoy nice closure and algorithmic properties, in particular, the regularity-preserving property (i.e., pre-images of regular constraints under PSSTs are regular), and introduce a sound sequent calculus that exploits these properties and performs propagation of regular constraints by means of taking post-images or pre-images. Although the satisfiability of the string constraint language is generally undecidable, we show that our approach is complete for the so-called straight-line fragment. We evaluate the performance of our string solver on over 195 000 string constraints generated from an open-source RegEx library. The experimental results show the efficacy of our approach, drastically improving the existing methods (via symbolic execution) in both precision and efficiency.

*Orcid ID: 0000-0002-5993-1665
†Orcid ID: 0000-0003-4913-3800
‡Orcid ID: 0000-0001-9171-4997
§Orcid ID: 0000-0003-4715-5096
¶Orcid ID: 0000-0002-2733-7098

1 Introduction

In modern programming languages—such as JavaScript, Python, Java, and PHP—the string data type plays a crucial role. A quick look at the string libraries for these languages is enough to convince oneself how well supported string manipulations are in these languages, in that a wealth of string operations and functions are readily available for the programmers. Such operations include usual operators like concatenation, length, substring, but also complex functions such as match, replace, split, and parseInt. Unfortunately, it is well-known that string manipulations are error-prone and could even give rise to security vulnerabilities (e.g. cross-site scripting, a.k.a. XSS). One powerful method for identifying such bugs in programs is *symbolic execution* (possibly in combination with dynamic analysis), which analyses symbolic paths in a program by viewing them as constraints whose feasibility is checked by constraint solvers. Together with the challenging problem of string analysis, this interplay between program analysis and constraint solvers has motivated the highly active research area of *string solving*, resulting in the development of numerous string solvers in the last decade or so including Z3 [27], CVC4 [41], Z3-str/2/3/4 [61, 60, 13, 14], ABC [17], Norn [3], Trau [16, 2, 1], OSTRICH [24], S2S [40], Qzy [25], Stranger [58], Sloth [36, 4], Slog [57], Slent [56], Gecode+S [50], G-Strings [9], HAMPI [39], among many others.

One challenging problem in the development of string solvers is the need to support an increasing number of real-world string functions, especially because the initial stage of the development of string solvers typically assumed only simple functions (in particular, concatenation, regular constraints, and sometimes also length constraints). For example, the importance of supporting functions like the replaceAll function (i.e. replace with global flag) in a string solver was elaborated in [22]; ever since, quite a number of string solvers support this operator. Unfortunately, the gap between the string functions that are supported by current string solvers and those supported by modern programming languages is still too big. As convincingly argued in [44] in the context of constraint solving, the widely used *Regular Expressions* in modern programming languages (among others, JavaScript, Python, etc.)—which we call *RegEx* in the sequel—are one important and frequently occurring feature in programs that are difficult for existing SMT theories over strings to model and solve, especially because their syntaxes and semantics substantially differ from the notion of regular expressions in formal language theory [38]. Indeed, many important string functions in programming languages—such as exec, test, search, match, replace, and split in JavaScript, as well as match, findall, search, sub, and split in Python—can and often do exploit RegEx, giving rise to path constraints that are difficult (if not impossible) to precisely capture in existing string solving frameworks. We illustrate these difficulties in the following two examples.

Example 1.1. *We briefly mention the challenges posed by the replace function in JavaScript; a slightly different but more detailed example can be found in Section 2. Consider the Javascript code snippet*

```
var namesReg = /([A-Za-z]+) ([A-Za-z]+)/g;
var newAuthorList = authorList.replace(nameReg, "£2, £1");
```

Assuming `authorList` *is given as a list of* `;`*-separated author names — first name, followed by a last name — the above program would convert this to last name, followed by first name format. For instance,* `"Don Knuth; Alan Turing"` *would be converted to* `"Knuth, Don; Turing, Alan"`*. A natural post condition for this code snippet one would like to check is the existence of at least one "," between two occurrences of ";".*

Example 1.2. *We consider the match function in JavaScript, in combination with replace. Consider the code snippet in Figure 1. The function* `normalize` *removes leading and trailing zeros from a decimal string with the input* `decimal`*. For instance,* `normalize("0.250") == "0.25"`*,* `normalize("02.50") == "2.5"`*,* `normalize("025.0") == "25"`*, and finally we have* `normalize("0250") == "250"`*. As the reader might have guessed, the function match actually returns an array of strings, corresponding to those that are matched in the* capturing groups *(two in our example) in the RegEx using the* greedy *semantics of the Kleene star/plus operator. One might be interested in checking, for instance, that there is a way to generate a the string* `"0.0007"`*, but not the string* `"00.007"`*.*

```
function normalize(decimal) {
  const decimalReg = /^(\d+)\.?(\d*)$/;
  var   decomp     = decimal.match(decimalReg);
  var   result     = "";
  if (decomp) {
    var integer    = decomp[1].replace(/^0+/, "");
    var fractional = decomp[2].replace(/0+$/, "");
    if (integer    !== "") result = integer; else result = "0";
    if (fractional !== "") result = result + "." + fractional;
  }
  return result;
}
```

Figure 1: Normalize a decimal by removing the leading and trailing zeros

The above examples epitomize the difficulties that have arisen from the interaction between RegEx and string functions in programs. Firstly, RegEx uses deterministic semantics for pattern matching (like greedy semantics in the above example, but the so-called *lazy* matching is also possible), and allows features that do not exist in regular expressions in formal language theory, e.g., capturing groups (those in brackets) in the above example. Secondly, string functions in programs can exploit RegEx in an intricate manner, e.g., by means of references $1 and $2 in Example 1.1. Hitherto, no existing string solvers can support any of these features. This is despite the fact that *idealized versions* of regular constraints and the replace functions are allowed in modern string solvers (e.g. see [2, 36, 41, 55, 59, 24]), i.e., features that can be found in the above examples like capturing groups, greedy/lazy matching, and references are not supported. This limitation of existing string solvers was already mentioned in the recent paper [44].

In view of the aforementioned limitation of string solvers, what solutions are possible? One recently proposed solution is to map the path constraints generated by string-manipulating programs that exploit RegEx into constraints in the SMT theories supported by existing string solvers. In fact, this was done in recent papers [44], where the path constraints are mapped to constraints in the theory of strings with concatenation and regular constraints in Z3 [27]. Unfortunately, this mapping is an *approximation*, since such complex string manipulations are generally *inexpressible* in any string theories supported by existing string solvers. To leverage this, CEGAR (counter-example guided abstraction and refinement) is used in [44], while ensuring that an *under-approximation* is preserved. This results in a rather severe price in both precision and performance: the refinement process may not terminate even for extremely simple programs (e.g. the above examples).

Therefore, the current state-of-affairs is unsatisfactory because even the introduction of very simple RegEx expressions in programs (e.g. the above examples) results in path constraints that can *not* be solved by existing symbolic executions in combination with string solvers. In this paper, we would like to firstly advocate that string solvers should *natively* support important features of RegEx in their SMT theories. Existing work (e.g. the reduction to Z3 provided by [44]) shows that this is a monumental theoretical and programming task, not to mention the loss in precision and the performance penalty. Secondly, we present *the first* string theory and string solver that natively provide such a support.

Contributions. In this paper, we provide *the first* string theory and string solver that natively support RegEx. Not only can our theory/solver easily express and solve Example 1.1 and Example 1.2 — which hitherto no existing string solvers and string analysis can handle — our experiments using a library of 98,117 real-world regular expressions indicate that our solver substantially outperforms the existing method [44] in terms of the number of solved problems and runtime. We provide more details of our contributions below.

Our string theory provides for the first time a native support of the match and the replace functions, which use JavaScript[1] RegEx in the input arguments. Here is a quick summary of our string constraint language (see Section 3 for more details):

$$\begin{aligned}\varphi \quad \overset{\text{def}}{=} \quad & x = y \mid z = x \cdot y \mid y = \mathsf{extract}_{i,e}(x) \mid y = \mathsf{replace}_{\mathsf{pat},\mathsf{rep}}(x) \mid \\ & y = \mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x) \mid x \in e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi\end{aligned}$$

where e, pat are RegExes, $i \in \mathbb{N}$, x, y, z are variables, and rep is called the replacement string and might refer to strings matched in capturing groups, as in Example 1.1. Apart from the standard concatenation operator $\cdot$, we support $\mathsf{extract}$, which extracts the string matched by the ith capturing group in the RegEx e (note that match can be simulated by several calls to $\mathsf{extract}$). We also support $\mathsf{replace}$ (resp. $\mathsf{replaceAll}$), which replaces the first occurrence (resp. all occurrences) of substrings in x matched by pat by rep. Our solver/theory also covers the most important features of RegEx (including greedy/lazy matching, capturing groups, among others) that make up 74.97% of the RegEx expressions of [44] across 415,487 NPM packages.

A crucial step in the development of our string solver is a formalization of the semantics of the $\mathsf{extract}$, $\mathsf{replace}$, and $\mathsf{replaceAll}$ functions in an automata-theoretic model that is amenable to analysis (among others, closure properties; see below). To this end, we introduce a new transducer model called *Prioritized Streaming String Transducers (PSSTs)*, which is inspired by two automata/transducer models: prioritized finite-state automata [11] and streaming string transducers [5, 6]. PSSTs allow us to precisely capture the non-standard semantics of RegEx operators (e.g. greedy/lazy Kleene star) by priorities and deal with capturing groups by string variables. We show that $\mathsf{extract}$, $\mathsf{replace}$, and $\mathsf{replaceAll}$ can all be expressed as PSSTs. More importantly, we have performed an extensive experiment validating our formalization against JavaScript semantics.

Next, by means of a sound sequent calculus, our string solver (implemented in the standard DPLL(T) setting of SMT solvers [47]) will exploit crucial closure and algorithmic properties satisfied by PSSTs. In particular, the solver attempts to (1) *propagate* regular constraints (i.e. the constraints $x \in e$) in the formula around by means of the string functions $\cdot$, $\mathsf{replace}$, $\mathsf{replaceAll}$, and $\mathsf{extract}$, and (2) either detect conflicting regular constraints, or find a satisfiable assignment. A single step of the regular-constraint propagation computes either the *post*-image or the *pre*-image of the above functions. In particular, it is crucial that each step of our

[1] JavaScript was chosen because it is relevant to string solving [37, 48], due to vulnerabilities in JavaScripts caused by string manipulations. Our method can be easily adapted to RegEx semantics in other languages.

constraint propagation preserves regularity of the constraints. Since the *post*-image does not always preserve regularity, we only propagate by taking *post*-image when regularity is preserved. On the other hand, one of our crucial results is that taking *pre*-image always preserves regularity: regular constraints are *effectively closed under taking pre-image of functions captured in PSSTs*. Finally, despite the fact that our above string theory is undecidable (which follows from [42]), we show that our string solving algorithm is guaranteed to terminate (and therefore is also complete) under the assumption that the input formula syntactically satisfies the so-called *straight-line restriction*.

We implement our decision procedure on top of the open-source solver OSTRICH [24], and carry out extensive experiments to evaluate the performance. For the benchmarks, we generate two collections of JavaScript programs (with 98,117 programs in each collection), from a library of real-world regular expressions [26], by using two simple JavaScript program templates containing match and replace functions, respectively. Then we generate all the four (resp. three) path constraints for each match (resp. replace) JavaScript program and put them into one SMT-LIB script. OSTRICH is able to answer all four (resp. three) queries in 97% (resp. 91.5%) of the match (resp. replace) scripts, with the average time 1.57s (resp. 6.62s) per file. Running ExpoSE [44] with the same time budget on the same benchmarks, we show that OSTRICH offers a 8x–18x speedup in comparison to ExpoSE, while being able to cover substantially more paths (9.6% more for match, 49.9% more for replace), making OSTRICH the first string solver that is able to handle RegExes precisely and efficiently.

Organization. In Section 2, more details of Example 1.1 are worked out to illustrate our approach. The string constraint language supporting RegExes is presented in Section 3. The semantics of the RegEx-dependent string functions are formally defined via PSSTs in Section 4. The sequent calculus for solving the string constraints is introduced in Section 5. The implementation of the string solver and experiments are described in Section 6. The related work is given in Section 7. Finally, Section 8 concludes this paper.

2 A Detailed Example

In this section, we provide a detailed example to illustrate our string solving method. Consider the JavaScript program in Figure 2; this example is similar to Example 1.1 from the Introduction. The function "authorNameDBLPtoACM" in Figure 2 transforms an author list in the DBLP BibTeX style to the one in the ACM BibTeX style. For instance, if a paper is authored by Alice M. Brown and John Smith, then the author list in the DBLP BibTeX style is "Alice M. Brown and John Smith", while it is "Brown, Alice M. and Smith, John" in the ACM BibTeX style.

The input of the function "authorNameDBLPtoACM" is authorList, which is expected to follow the pattern specified by the regular expression autListReg. Intuitively, autListReg stipulates that authorList joins the strings of full names as a concatenation of a given name, middle names, and a family name, separated by the blank symbol (denoted by \s). Each of the given, middle, family names is a concatenation of a capital alphabetic letter (denoted by [A-Z]) followed by a sequence of letters (denoted by \w) or a dot symbol (denoted by .). Between names, the word "and" is used as the separator. The symbols ˆ and $ denote the beginning and the end of a string input respectively.

The DBLP name format of each author is specified by the regular expression nameReg in Figure 2, which describes the format of a full name.

```
function authorNameDBLPtoACM(authorList)
{
  var autListReg
    = /^[A-Z](\w*|.)(\s[A-Z](\w*|.))*(\sand\s[A-Z](\w*|.)(\s[A-Z](\w*|.))*)*$/;
  if (autListReg.test(authorList)) {
    var nameReg = /([A-Z](?:\w*|.)(?:\s[A-Z](?:\w*|.))*)(\s[A-Z](?:\w*|.))/g;
    return authorList.replace(nameReg, "$2, $1");
  }
  else return authorList;
}
```

Figure 2: Change the author list from the DBLP format to the ACM format

```
  var autListReg =
      /^[A-Z](\w*|.)(\s[A-Z](\w*|.))*(\sand\s[A-Z](\w*|.)(\s[A-Z](\w*|.))*)*$/;
  assume(autListReg.test(authorList));
  var nameReg = /([A-Z](?:\w*|.)(?:\s[A-Z](?:\w*|.))*)(\s[A-Z](?:\w*|.))/g;
  var result = authorList.replace(nameReg, "$2, $1");
  assume(/\sand[^,]*\sand/.test(result));
```

Figure 3: Symbolic execution of a path of the JavaScript program in Fig. 2

- There are two capturing groups in nameReg, one for recording the concatenation of the given name and middle names, and the other for recording the family name. Note that the symbols ?: in (?:\s[A-Z](?:\w*—.)) denote the non-capturing groups, i.e. matching the subexpression, but not remembering the match.

- The *greedy* semantics of the Kleene star * is utilized here to guarantee that the subexpression (?:\s[A-Z](?:\w*—\.))* matches all the middle names (since there may exist multiple middle names) and thus nameReg matches the full name. For instance, the first match of nameReg in "Alice M. Brown and John Smith" is "Alice M. Brown", instead of "Alice M.". In comparison, if the semantics of * is assumed to be non-greedy, then (?:\s[A-Z](?:\w*—\.))* can be matched to the empty string, thus nameReg is matched to "Alice M.", which is *not* what we want. Therefore, the greedy semantics of * is essential for the correctness of "authorNameDBLPtoACM".

- The global flag "g" is used in nameReg so that the name format of each author is transformed.

The name format transformation is via the replace function, i.e. authorList.replace(nameReg, "$2, $1"), where $1 and $2 refer to the match of the first and second capturing group respectively.

A natural post-condition of authorNameDBLPtoACM is that there exists at least one occurrence of the comma symbol between every two occurrences of "and". This post-condition has to be established by the function on *every* execution path. As an example, consider the path shown in Fig. 3, in which the branches taken in the program are represented as assume statements. The negated post-condition is enforced by the regular expression in the last assume. For this path, the post-condition can be proved by showing that the program in Fig. 3 is infeasible: there does not exist an initial value authorList so that no assumption fails and the program executes to the end.

To enable symbolic execution of the JavaScript programs like in Fig. 3, one needs to model both the greedy semantics of the Kleene star and store the matches of capturing groups. For this purpose, we introduce prioritized streaming string transducers (PSST, cf. Section 4) by which replace(nameReg, "$2, $1") is represented as a PSST $\mathcal{T}$, where the *priorities* are used to model the greedy semantics of $*$ and the *string variables* are used to record the matches of the capturing groups as well as the return value. Then the symbolic execution of the program in Fig. 3 can be equivalently turned into the satisfiability of the following string constraint,

$$\mathsf{authorList} \in \mathsf{autListReg} \wedge \mathsf{result} = \mathcal{T}(\mathsf{authorList}) \wedge \mathsf{result} \in \mathsf{postConReg}, \tag{1}$$

where postConReg = `/^.*\sand[^,]*\sand.*$/`, and autListReg is as in Fig. 2.

Our solver is able to show that (1) is unsatisfiable. On the calculus level (introduced in more details in Section 5), the main inference step applied for this purpose is the computation of the *pre-image* of postConReg under the function $\mathcal{T}$; in other words, we compute the language of all strings that are mapped to incorrect strings (containing two "and"s without a comma in between) by $\mathcal{T}$. This inference step relies on the fact that the pre-images of regular languages under PSSTs are regular (see Lemma 5.5). Denoting the pre-image of postConReg by $\mathcal{B}$, formula (1) is therefore equivalent to

$$\mathsf{authorList} \in \mathcal{B} \wedge \mathsf{authorList} \in \mathsf{autListReg} \wedge \mathsf{result} = \mathcal{T}(\mathsf{authorList}) \wedge \mathsf{result} \in \mathsf{postConReg}. \tag{2}$$

To show that this formula (and thus (1)) is unsatisfiable, it is now enough to prove that the languages defined by $\mathcal{B}$ and autListReg are disjoint.

3 A String Constraint Language Natively Supporting RegEx

In this section, we define a string constraint language natively supporting RegEx. Throughout the paper, $\mathbb{Z}^+$ denotes the set of positive integers, and $\mathbb{N}$ denotes the set of natural numbers. Furthermore, for $n \in \mathbb{Z}^+$, let $[n] := \{1, \ldots, n\}$. We use Σ to denote a finite set of letters, called *alphabet*. A *string* over Σ is a finite sequence of letters from Σ. We use Σ^* to denote the set of strings over Σ, ε to denote the empty string, and Σ^ε to denote $\Sigma \cup \{\varepsilon\}$. A string w' is called a *prefix* (resp. *suffix*) of w if $w = w'w''$ (resp. $w = w''w'$) for some string w''.

We start with the syntax of RegEx which is essentially that used in JavaScript. (We do not include backreferences though.)

Definition 3.1 (Regular expressions, RegEx)**.**

$$e \overset{def}{=} \emptyset \mid \varepsilon \mid a \mid (e) \mid [e + e] \mid [e \cdot e] \mid [e^?] \mid [e^{??}] \mid$$
$$[e^*] \mid [e^{*?}] \mid [e^+] \mid [e^{+?}] \mid [e^{\{m_1,m_2\}}] \mid [e^{\{m_1,m_2\}?}]$$

where $a \in \Sigma$, $n \in \mathbb{Z}^+$, $m_1, m_2 \in \mathbb{N}$ with $m_1 \leq m_2$.

For $\Gamma = \{a_1, \ldots, a_k\} \subseteq \Sigma$, we write Γ for $[[\cdots[a_1 + a_2] + \cdots] + a_k]$ and thus $[\Gamma^*] \equiv [[[\cdots[a_1 + a_2] + \cdots] + a_k]^*]$. Similarly for $[\Gamma^{*?}]$, $[\Gamma^+]$, and $[\Gamma^{+?}]$. We write $|e|$ for the length of e, i.e., the number of symbols occurring in e. Note that square brackets [] are used for the operator precedence and the parentheses () are used for *capturing groups*.

The operator $[e^*]$ is the *greedy* Kleene star, meaning that e should be matched as many times as possible. In contrast, the operator $[e^{*?}]$ is the *lazy* Kleene star, meaning e should be matched as few times as possible. The Kleene plus operators $[e^+]$ and $[e^{+?}]$ are similar to $[e^*]$

and $[e^{*?}]$ but e should be matched at least once. Moreover, as expected, the repetition operators $[e^{\{m_1,m_2\}}]$ require the number of times that e is matched is between m_1 and m_2 and $[e^{\{m_1,m_2\}?}]$ is the lazy variant. Likewise, the optional operator has greedy and lazy variants $[e^?]$ and $[e^{??}]$, respectively.

For two RegEx e and e', we say that e' is a *subexpression* of e, if one of the following conditions holds: 1) $e' = e$, 2) $e = [e_1 \cdot e_2]$ or $[e_1 + e_2]$, and e' is a subexpression of e_1 or e_2, 3) $e = [e_1^?], [e_1^{??}], [e_1^*], [e_1^+], [e_1^{*?}], [e_1^{+?}], [e_1^{\{m_1,m_2\}}], [e_1^{\{m_1,m_2\}?}]$ or (e_1), and e' is a subexpression of e_1. We use $S(e)$ to denote the set of subexpressions of e.

We shall formalize the semantics of RegEx, in particular, for a given regular expression and an input string, how the string is matched against the regular expression, in Section 4.2.

In the rest of this section, we define the string constraint language STR.

The syntax of STR is defined by the following rules.

$$\begin{aligned}\varphi \quad \overset{\text{def}}{=} \quad & x = y \mid z = x \cdot y \mid y = \mathsf{extract}_{i,e}(x) \mid y = \mathsf{replace}_{\mathsf{pat},\mathsf{rep}}(x) \mid \\ & y = \mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x) \mid x \in e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi\end{aligned}$$

where

- $\cdot$ is the string concatenation operation which concatenates two strings,
- $e \in$RegEx and $\mathsf{pat} \in$RegEx,
- for the $\mathsf{extract}$ function, $i \in \mathbb{N}$,
- for the $\mathsf{replace}$ and $\mathsf{replaceAll}$ operation, $\mathsf{rep} \in \mathsf{REP}$, where REP is defined as a concatenation of letters from Σ, the references $\$i$ ($i \in \mathbb{N}$), as well as $\$^{\leftarrow}$ and $\$^{\rightarrow}$. (Intuitively, $\$0$ denotes the matching of pat, $\$i$ with $i > 0$ denotes the matching of the i-th capturing group, $\$^{\leftarrow}$ and $\$^{\rightarrow}$ denote the prefix before resp. suffix after the matching of pat.)

The $\mathsf{extract}_{i,e}(x)$ function extracts the match of the i-th capturing group in the successful match of e to x for $x \in \mathscr{L}(e)$ (otherwise, the return value of the function is undefined). Note that $\mathsf{extract}_{i,e}(x)$ returns x if $i = 0$. Moreover, if the i-th capturing group of e is *not* matched, even if $x \in \mathscr{L}(e)$, then $\mathsf{extract}_{i,e}(x)$ returns a special symbol null, denoting the fact that its value is undefined. For instance, when $[[a^+] + ([a^*])]$ is matched to the string aa, $[a^+]$, instead of $([a^*])$, will be matched, since $[a^+]$ precedes $([a^*])$. Therefore, $\mathsf{extract}_{1,[[a^+]+([a^*])]}(aa) = \mathsf{null}$.

Remark 1. *The match function in programming languages, e.g.* $\mathsf{str.match(reg)}$ *function in JavaScript, finds the first match of* reg *in* str*, assuming that* reg *does not contain the global flag. We can use* $\mathsf{extract}$ *to express the first match of* reg *in* str *by adding* $[\Sigma^{*?}]$ *and* $[\Sigma^*]$ *before and after* reg *respectively. More generally, the value of the i-th capturing group in the first match of a RegEx* reg *in* str *can be specified as* $\mathsf{extract}_{i+1,\mathsf{reg}'}(\mathsf{str})$*, where* $\mathsf{reg}' = [[[\Sigma^{*?}] \cdot (\mathsf{reg})] \cdot [\Sigma^*]]$*. The other string functions involving regular expressions, e.g.* exec *and* test*, without global flags, are similar to* match*, thus can be encoded by* $\mathsf{extract}$ *as well.*

The function $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$ is parameterized by the *pattern* $\mathsf{pat} \in$ *RegEx* and the *replacement string* $\mathsf{rep} \in \mathsf{REP}$. For an input string x, it identifies all matches of pat in x and replaces them with strings specified by rep. More specifically, $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$ finds the first match of pat in x and replaces the match with rep, let x' be the suffix of x after the first match of pat, then it finds the first match of pat in x' and replace the match with rep, and so on. A reference $\$i$ where $i > 0$ is instantiated by the matching of the i-th capturing group. There are three special references[2] $0, $\$^{\leftarrow}$, and $\$^{\rightarrow}$. These are instantiated by the matched text, the text occurring before the match, and the text occurring after the match respectively. In particular, if the input word is uvw where v has been matched and will be replaced, then $0 takes the value v, $\$^{\leftarrow}$ takes the value u, and $\$^{\rightarrow}$ takes the value w. When there are multiple matches in a replaceAll, the values of $\$^{\leftarrow}$ and $\$^{\rightarrow}$ are always with respect to the original input string x.

The $\mathsf{replace}_{\mathsf{pat},\mathsf{rep}}(x)$ function is similar to $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$, except that it replaces only the first (leftmost) match of pat.

A STR formula φ is said to be *straight-line*, if 1) it contains neither negation nor disjunction, 2) the equations in φ can be ordered into a sequence, say $x_1 = t_1, \ldots, x_n = t_n$, such that $x_1, \ldots, x_n$ are mutually distinct, moreover, for each $i \in [n]$, x_i does *not* occur in $t_1, \ldots, t_{i-1}$. Let $\mathsf{STR}_{\mathsf{SL}}$ denote the set of straight-line STR formulas.

As a crucial step for solving the string constraints in STR, we shall define the formal semantics of the extract, replace, and replaceAll functions in the next section.

4 Semantics of string functions via PSST

Our goal in this section is to define the formal semantics of the string functions involving RegEx used in STR, that is, extract, replace and replaceAll. To this end, we need to first define the semantics of RegEx-string matching. One of the key novelties here is to utilize an extension of finite-state automata with transition priorities and string variables, called prioritized streaming string transducers (abbreviated as PSST). It turns out that PSST provides a convenient means to capture the non-standard semantics of RegEx operators and to store the matches of capturing groups in RegEx, which paves the way to define the semantics of string functions (and the string constraint language).

4.1 Prioritized streaming string transducers (PSST)

PSSTs can be seen as an extension of finite-state automata with transition priorities and string variables. We first recall the definition of classic finite-state automata.

Definition 4.1 (Finite-state Automata)**.** *A* (nondeterministic) finite-state automaton *(FA) over a finite alphabet Σ is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times \Sigma^{\varepsilon} \times Q$ is the transition relation.*

For an input string w, a *run* of $\mathcal{A}$ on w is a sequence $q_0 a_1 q_1 \ldots a_n q_n$ such that $w = a_1 \cdots a_n$ and $(q_{j-1}, a_j, q_j) \in \delta$ for every $j \in [n]$. The run is said to be *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. The set of strings accepted by $\mathcal{A}$, i.e., the language *recognized* by $\mathcal{A}$, is denoted by $\mathscr{L}(\mathcal{A})$. The *size* $|\mathcal{A}|$ of $\mathcal{A}$ is the cardinality of δ, the set of transitions.

For a finite set Q, let $\overline{Q} = \bigcup_{n \in \mathbb{N}} \{(q_1, \ldots, q_n) \mid \forall i \in [n], q_i \in Q \wedge \forall i, j \in [n], i \neq j \to q_i \neq q_j\}$. Intuitively, $\overline{Q}$ is the set of sequences of non-repetitive elements from Q. In particular, the empty sequence $() \in \overline{Q}$. Note that the length of each sequence from $\overline{Q}$ is bounded by $|Q|$. For a sequence $P = (q_1, \ldots, q_n) \in \overline{Q}$ and $q \in Q$, we write $q \in P$ if $q = q_i$ for some $i \in [n]$. Moreover, for $P_1 = (q_1, \ldots, q_m) \in \overline{Q}$ and $P_2 = (q'_1, \ldots, q'_n) \in \overline{Q}$, we say $P_1 \cap P_2 = \emptyset$ if $\{q_1, \ldots, q_m\} \cap \{q'_1, \ldots, q'_n\} = \emptyset$.

Definition 4.2 (Prioritized Streaming String Transducers)**.** *A* prioritized streaming string transducer *(*PSST*) is a tuple $\mathcal{T} = (Q, \Sigma, X, \delta, \tau, E, q_0, F)$, where*

[2] The corresponding syntax for $0, $\$^{\leftarrow}$ and $\$^{\rightarrow}$ in JavaScript are $&, $‘, and $′.

- *Q is a finite set of states,*

- *Σ is the input and output alphabet,*

- *X is a finite set of string variables,*

- *$\delta \in Q \times \Sigma \to \overline{Q}$ defines the non-ε transitions as well as their priorities (from highest to lowest),*

- *$\tau \in Q \to \overline{Q} \times \overline{Q}$ such that for every $q \in Q$, if $\tau(q) = (P_1; P_2)$, then $P_1 \cap P_2 = \emptyset$, (Intuitively, $\tau(q) = (P_1; P_2)$ specifies the ε-transitions at q, with the intuition that the ε-transitions to the states in P_1 (resp. P_2) have higher (resp. lower) priorities than the non-ε-transitions out of q.)*

- *E associates with each transition a string-variable assignment function, i.e., E is partial function from $Q \times \Sigma^{\varepsilon} \times Q$ to $X \to (X \cup \Sigma)^*$ such that its domain is the set of tuples (q, a, q') satisfying that either $a \in \Sigma$ and $q' \in \delta(q, a)$ or $a = \varepsilon$ and $q' \in \tau(q)$,*

- *$q_0 \in Q$ is the initial state, and*

- *F is the output function, which is a partial function from Q to $(X \cup \Sigma)^*$.*

For $\tau(q) = (P_1; P_2)$, we will use $\pi_1(\tau(q))$ and $\pi_2(\tau(q))$ to denote P_1 and P_2 respectively. The size of $\mathcal{T}$, denoted by $|\mathcal{T}|$, is defined as $\sum\limits_{(q,a,q') \in \mathsf{dom}(E)} \sum\limits_{x \in X} |E((q, a, q'))(x)|$, where $|E((q, a, q'))(x)|$ is the length of $E(q, a, q')(x)$, i.e., the number of symbols from $X \cup \Sigma$ in it. A PSST $\mathcal{T}$ is said to be *copyless* if for each transition (q, a, q') in $\mathcal{T}$ and each $x \in X$, x occurs in $(E(q, a, q')(x'))_{x' \in X}$ at most once. A PSST $\mathcal{T}$ is said to be *copyful* if it is not copyless. For instance, if $X = \{x_1, x_2\}$ and $E(q, a, q')(x_1) = x_1$ and $E(q, a, q')(x_2) = x_1 a$ for some transition (q, a, q'), then x_1 occurs twice in $(E(q, a, q')(x'))_{x' \in X}$, thus $\mathcal{T}$ is copyful.

A run of $\mathcal{T}$ on a string w is a sequence $q_0 a_1 s_1 q_1 \dots a_m s_m q_m$ such that

- for each $i \in [m]$,
  - either $a_i \in \Sigma$, $q_i \in \delta(q_{i-1}, a_i)$, and $s_i = E(q_{i-1}, a_i, q_i)$,
  - or $a_i = \varepsilon$, $q_i \in \tau(q_{i-1})$ and $s_i = E(q_{i-1}, \varepsilon, q_i)$,

- for every subsequence $q_i a_{i+1} s_{i+1} q_{i+1} \dots a_j s_j q_j$ such that $i < j$ and $a_{i+1} = \dots = a_j = \varepsilon$, it holds that each ε-transition occurs at most once in it, namely, for every $k, l : i \leq k < l < j$, $(q_k, q_{k+1}) \neq (q_l, q_{l+1})$.

Note that it is possible that $\delta(q, a) = ()$, that is, there is no a-transition out of q. From the assumption that each ε-transition occurs at most once in a sequence of ε-transitions, we deduce that given a string w, the length of a run of $\mathcal{T}$ on w, i.e. the number of transitions in it, is $O(|w||\mathcal{T}|)$.

For any pair of runs $R = q_0 a_1 s_1 \dots a_m s_m q_m$ and $R' = q_0 a'_1 s'_1 \dots a'_n s'_n q'_n$ such that $a_1 \dots a_m = a'_1 \dots a'_n$, we say that R is of a higher priority over R' if

- either R' is a prefix of R (in this case, the transitions of R after R' are all ε-transitions),

- or there is an index j satisfying one of the following constraints:
  - $q_0 a_1 q_1 \dots q_{j-1} a_j = q_0 a'_1 q'_1 \dots q'_{j-1} a'_j$, $q_j \neq q'_j$, $a_j \in \Sigma$, and we have that $\delta(q_{j-1}, a_j) = (\dots, q_j, \dots, q'_j, \dots)$,

- $q_0 a_1 q_1 \dots q_{j-1} a_j = q_0 a'_1 q'_1 \dots q'_{j-1} a'_j$, $q_j \neq q'_j$, $a_j = \varepsilon$, and one of the following holds: (i) $\pi_1(\tau(q_{j-1})) = (\dots, q_j, \dots, q'_j, \dots)$, (ii) $\pi_2(\tau(q_{j-1})) = (\dots, q_j, \dots, q'_j, \dots)$, or (iii) $q_j \in \pi_1(\tau(q_{j-1}))$ and $q'_j \in \pi_2(\tau(q_{j-1}))$,
- $q_0 a_1 q_1 \dots q_{j-1} = q_0 a'_1 q'_1 \dots q'_{j-1}$, $a_j = \varepsilon$, $a'_j \in \Sigma$, $q_j \in \pi_1(\tau(q_{j-1}))$, and $q'_j \in \delta(q_{j-1}, a'_j)$,
- $q_0 a_1 q_1 \dots q_{j-1} = q_0 a'_1 q'_1 \dots q'_{j-1}$, $a_j \in \Sigma$, $a'_j = \varepsilon$, $q_j \in \delta(q_{j-1}, a_j)$, and $q'_j \in \pi_2(\tau(q_{j-1}))$.

An *accepting* run of $\mathcal{T}$ on w is a run of $\mathcal{T}$ on w, say $R = q_0 a_1 s_1 \dots a_m s_m q_m$, such that 1) $F(q_m)$ is defined, 2) R is of the highest priority among those runs satisfying 1). The output of $\mathcal{T}$ on w, denoted by $\mathcal{T}(w)$, is defined as $\eta_m(F(q_m))$, where $\eta_0(x) = \varepsilon$ for each $x \in X$, and $\eta_i(x) = \eta_{i-1}(s_i(x))$ for every $1 \leq i \leq m$ and $x \in X$. Note that here we abuse the notation $\eta_m(F(q_m))$ and $\eta_{i-1}(s_i(x))$ by taking a function η from X to Σ^* as a function from $(X \cup \Sigma)^*$ to Σ^*, which maps each $x \in X$ to $\eta(x)$ and each $a \in \Sigma$ to a. If there is no accepting run of $\mathcal{T}$ on w, then $\mathcal{T}(w) = \bot$, that is, the output of $\mathcal{T}$ on w is undefined. The string relation defined by $\mathcal{T}$, denoted by $\mathcal{R}_{\mathcal{T}}$, is $\{(w, \mathcal{T}(w)) \mid w \in \Sigma^*, \mathcal{T}(w) \neq \bot\}$.

Example 4.3. *The* PSST $\mathcal{T} = (Q, \Sigma, X, \delta, \tau, E, q_0, F)$ *to extract the match of the first capturing group for the regular expression* `(\d+)(\d*)` *is illustrated in Fig. 4, where x_1 and x_2 store the matches of the two capturing groups. More specifically, in $\mathcal{T}$ we have $\Sigma = \{0, \cdots, 9\}$, $X = \{x_1, x_2\}$, $F(q_4) = x_1$ denotes the final output, and δ, τ, E are illustrated in Fig. 4, where the dashed edges denote the ε-transitions of lower priorities than the non-ε-transitions and the symbol ℓ denotes the currently scanned input letter. For instance, for the state q_2, $\delta(q_2, \ell) = (q_2)$ for $\ell \in \{0, \dots, 9\}$, $\tau(q_2) = ((); (q_3))$, $E(q_2, \ell, q_2)(x_1) = x_1\ell$, $E(q_2, \ell, q_2)(x_2) = x_2$, $E(q_2, \varepsilon, q_3)(x_1) = x_1$, and $E(q_2, \varepsilon, q_3)(x_2) = \varepsilon$. Note that the identity assignments, e.g. $E(q_2, \varepsilon, q_3)(x_1) = x_1$, are omitted in Fig. 4 for readability. For the input string w= "2050", the accepting run of $\mathcal{T}$ on w is*

$$q_0 \xrightarrow[x_1:=\varepsilon]{\varepsilon} q_1 \xrightarrow[x_1:=x_1 2]{2} q_2 \xrightarrow[x_1:=x_1 0]{0} q_2 \xrightarrow[x_1:=x_1 5]{5} q_2 \xrightarrow[x_1:=x_1 0]{0} q_2 \xrightarrow[x_2:=\varepsilon]{\varepsilon} q_3 \xrightarrow{\varepsilon} q_4,$$

where the value of x_1 and x_2 when reaching the state q_4 are "2050" and ε respectively.

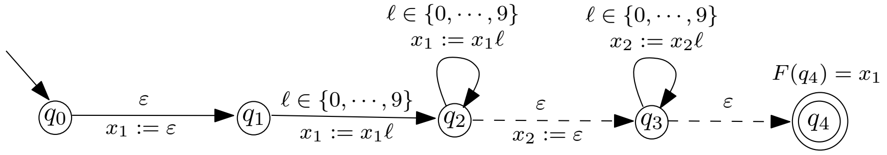


Figure 4: The PSST $\mathcal{T}$: Extract the matching of the first capturing group in (\d+)(\d*)

4.2 Semantics of RegEx-String Matching

We now define the formal semantics of RegEx. Traditionally they are interpreted as a regular language which can be defined inductively. In our case, where RegEx are mainly used in string functions, what matters is the intermediate result when parsing a string against the given RegEx. As a result, we shall present an operational (as opposed to traditional denotational) account of the RegEx-string matching by constructing PSSTs out of regular expressions.

Note that in [10, 11], a construction from RegEx to prioritized finite transducers (PFT) was given. The construction therein is a variant of the classical Thompson construction from regular expressions to nondeterministic finite automata [53]. In particular, the size of the constructed PFT is linear in the size of the given RegEx. One may be tempted to think that the construction in [10, 11] can be easily adapted to construct PSSTs out of regular expressions. Nevertheless, the construction in [10, 11] does *not* work for so called *problematic regular expressions*, i.e., those regular expressions that contain the subexpressions e^* or $e^{*?}$ with $\varepsilon \in \mathscr{L}(e)$. Moreover, the construction therein did not consider the repetition operators $[e_1^{\{m_1,m_2\}}]$ or $[e_1^{\{m_1,m_2\}?}]$. Our construction, which is considerably different from that in [10, 11], works for arbitrary regular expressions. In particular, the size of the constructed PSST can be *exponential* in the size of the given regular expression in the worst case. Moreover, we validate by extensive experiments that our construction is consistent with the actual RegEx-string matching in JavaScript.

For technical convenience, we assume that F in a PSST is a set of final states, instead of an output function, in the sequel. The main idea of the construction is to split the set of final states, F, into two disjoint subsets F_1 and F_2, with the intention that F_1 and F_2 are responsible for accepting the empty string resp. non-empty strings. Therefore, the PSSTs constructed below are of the form $(Q, \Sigma, X, \delta, \tau, E, q_0, (F_1, F_2))$. The necessity of this splitting will be illustrated in Example 4.6.

Furthermore, to deal with the situation that some capturing group may not be matched to any string and its value is undefined, we introduce a special symbol null and assume that the initial values of all the string variables are null. For simplicity, in the definition of a PSST, if $\delta(q, a, q') = ()$ or $\tau(q, \varepsilon, q') = ((); ())$, they will not be stated explicitly. Moreover, we will omit all the assignments $E(q, a, q')(x)$ such that $E(q, a, q')(x) = x$.

For PSSTs of the form $(Q, \Sigma, X, \delta, \tau, E, q_0, (F_1, F_2))$, we introduce a notation to be used in the construction, namely, the concatenation of two PSSTs.

Definition 4.4 (Concatenation of two PSSTs)**.** *For $i \in \{1, 2\}$, let $\mathcal{T}_i$ be a PSST such that $\mathcal{T}_i = (Q_i, \Sigma, X_i, \delta_i, \tau_i, E_i, q_{i,0}, (F_{i,1}, F_{i,2}))$. Then the* concatenation *of $\mathcal{T}_1$ and $\mathcal{T}_2$, denoted by $\mathcal{T}_1 \cdot \mathcal{T}_2$, is defined as follows (see Fig. 5): Let $\mathcal{T}_2' = (Q_2', \Sigma, X_2, \delta_2', \tau_2', E_2', q_{2,0}', (F_{2,1}', F_{2,2}'))$ be a fresh copy of $\mathcal{T}_2$, but with the string variables of $\mathcal{T}_2$ kept unchanged. Then*

$$\mathcal{T} = (Q_1 \cup Q_2 \cup Q_2', \Sigma, X_1 \cup X_2, \delta, \tau, q_{1,0}, (F_{2,1}, F_{2,2} \cup F_{2,1}' \cup F_{2,2}'))$$

where

- *δ comprises the transitions in δ_1, δ_2, and δ_2',*
- *τ comprises the transitions in τ_1, τ_2, τ_2', and the following transitions,*
  - *for every $f_{1,1} \in F_{1,1}$, $\tau(f_{1,1}) = ((q_{2,0}); ())$,*
  - *for every $f_{1,2} \in F_{1,2}$, $\tau(f_{1,2}) = ((q_{2,0}'), ())$,*
- *E inherits all the assignments in E_1, E_2, and E_2', and includes the following assignments: for every $f_{1,1} \in F_{1,1}$, $f_{1,2} \in F_{1,2}$, and $x' \in X_2$, $E(f_{1,1}, \varepsilon, q_{2,0})(x') = E(f_{1,2}, \varepsilon, q_{2,0}')(x') =$ null. (Intuitively, the values of all the variables in X_2 are reset when entering $\mathcal{T}_2$ and $\mathcal{T}_2'$.)*

Note that in the above definition, it is possible that $X_1 \cap X_2 \neq \emptyset$. We remark that if $F_{1,1} = \emptyset$ or $F_{2,1} = \emptyset$, then *one copy* of $\mathcal{T}_2$, instead of two copies, is sufficient for the concatenation.

We shall recursively construct a PSST $\mathcal{T}_e$ for each RegEx e, such that the initial state has no incoming transitions and each of its final states has no outgoing transitions. Moreover, all the transitions out of the initial state are ε-transitions. We assume that in $\mathcal{T}_e$, a string variable $x_{e'}$ is introduced for each subexpression e' of e.

The construction is technical and below we only select to present some representative cases. The other cases are given in the long version of this paper [23].

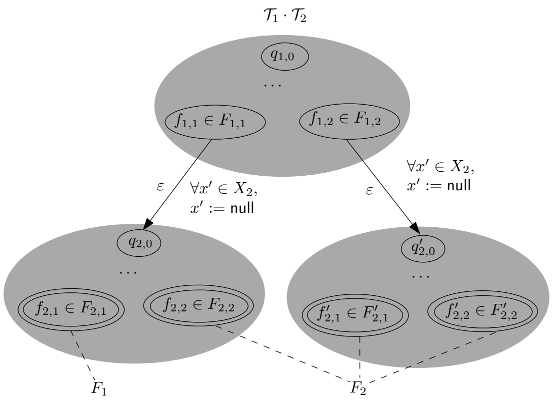


Figure 5: $\mathcal{T}_1 \cdot \mathcal{T}_2$: Concatenation of $\mathcal{T}_1$ and $\mathcal{T}_2$

Case $e = (e_1)$ $\mathcal{T}_e$ is adapted from $\mathcal{T}_{e_1} = (Q_{e_1}, \Sigma, X_{e_1}, \delta_{e_1}, \tau_{e_1}, E_{e_1}, q_{e_1,0}, (F_{e_1,1}, F_{e_1,2}))$ by adding the string variable x_e and the assignments for x_e, that is, $X_e = X_{e_1} \cup \{x_e\}$ and for each transition (q, a, q') in $\mathcal{T}_{e_1}$ with $a \in \Sigma^\varepsilon$, we have $E_e(q, a, q')(x_e) = E_{e_1}(q, a, q')(x_{e_1})$.

Case $e = [e_1 + e_2]$ **(see Fig. 6)** For $i \in \{1, 2\}$, let it be the case that we have $\mathcal{T}_{e_i} = (Q_{e_i}, \Sigma, X_{e_i}, \delta_{e_i}, \tau_{e_i}, E_{e_i}, q_{e_i,0}, (F_{e_i,1}, F_{e_i,2}))$. Moreover, assume $X_{e_1} \cap X_{e_2} = \emptyset$. Then

$$\mathcal{T}_e = (Q_{e_1} \cup Q_{e_2} \cup \{q_{e,0}\}, \Sigma, X_{e_1} \cup X_{e_2} \cup \{x_e\}, \delta_e, \tau_e, E_e, q_{e,0}, (F_{e_1,1} \cup F_{e_2,1}, F_{e_1,2} \cup F_{e_2,2}))$$

where

- δ_e comprises the transitions in δ_{e_1} and δ_{e_2},
- τ_e comprises the transitions in τ_{e_1} and τ_{e_2}, as well as the transition $\tau_e(q_{e,0}) = ((q_{e_1,0}); (q_{e_2,0}))$,
- E_e inherits E_{e_1}, E_{e_2}, plus the assignments $E_e(q_{e,0}, \varepsilon, q_{e_1,0})(x_e) = E_e(q_{e,0}, \varepsilon, q_{e_2,0})(x_e) = \varepsilon$, as well as $E_e(q, a, q')(x_e) = x_e a$ for every transition (q, a, q') in $\mathcal{T}_{e_1}$ and $\mathcal{T}_{e_2}$ (where $a \in \Sigma^\varepsilon$).

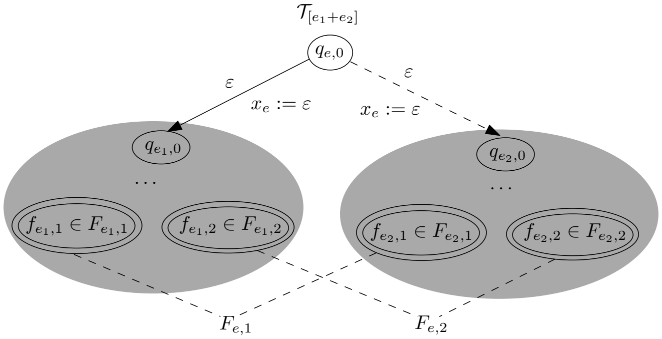

$\mathcal{T}_{[e_1+e_2]}$
$q_{e,0}$
ε
$x_e := \varepsilon$
ε
$x_e := \varepsilon$
$q_{e_1,0}$
...
$f_{e_1,1} \in F_{e_1,1}$
$f_{e_1,2} \in F_{e_1,2}$
$q_{e_2,0}$
...
$f_{e_2,1} \in F_{e_2,1}$
$f_{e_2,2} \in F_{e_2,2}$
$F_{e,1}$
$F_{e,2}$

Figure 6: The PSST $\mathcal{T}_{[e_1+e_2]}$

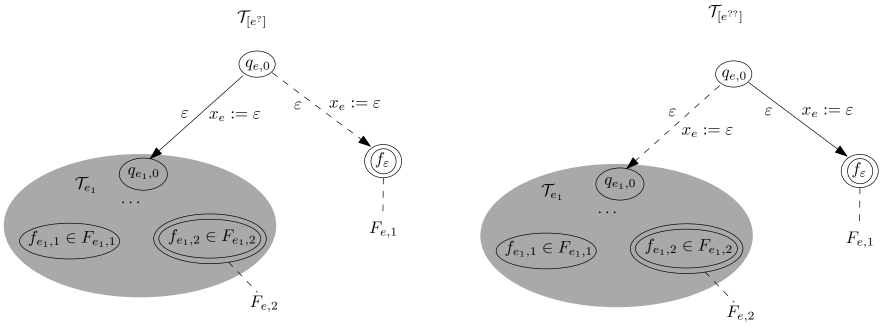

$\mathcal{T}_{[e^?]}$
$q_{e,0}$
ε $x_e := \varepsilon$
ε $x_e := \varepsilon$
$\mathcal{T}_{e_1}$
$q_{e_1,0}$
$\ldots$
$f_{e_1,1} \in F_{e_1,1}$
$f_{e_1,2} \in F_{e_1,2}$
$F_{e,2}$
f_ε
$F_{e,1}$
$\mathcal{T}_{[e^{??}]}$
$q_{e,0}$
ε $x_e := \varepsilon$
ε $x_e := \varepsilon$
$\mathcal{T}_{e_1}$
$q_{e_1,0}$
$\ldots$
$f_{e_1,1} \in F_{e_1,1}$
$f_{e_1,2} \in F_{e_1,2}$
$F_{e,2}$
f_ε
$F_{e,1}$

Figure 7: The PSST $\mathcal{T}_{[e_1^?]}$ and $\mathcal{T}_{[e_1^{??}]}$

Case $e = [e_1 \cdot e_2]$ For $i \in \{1,2\}$, let $\mathcal{T}_{e_i} = (Q_{e_i}, \Sigma, X_{e_i}, \delta_{e_i}, \tau_{e_i}, E_{e_i}, q_{e_i,0}, (F_{e_i,1}, F_{e_i,2}))$. Moreover, let us assume that $X_{e_1} \cap X_{e_2} = \emptyset$. Then $\mathcal{T}_e$ is obtained from $\mathcal{T}_{e_1} \cdot \mathcal{T}_{e_2}$ (the concatenation of $\mathcal{T}_{e_1}$ and $\mathcal{T}_{e_2}$, see Fig. 5) by adding a string variable x_e, a fresh state $q_{e,0}$ as the initial state, the ε-transition $\tau_e(q_{e,0}) = ((q_{e_1,0}); ())$, and the assignments $E_e(q_{e,0}, \varepsilon, q_{e_1,0})(x_e) = \varepsilon$, $E_e(p,a,q)(x_e) = x_e a$ for every transition (p,a,q) in $\mathcal{T}_{e_1}$, $\mathcal{T}_{e_2}$, and $\mathcal{T}'_{e_2}$ (where $a \in \Sigma^\varepsilon$).

Case $e = [e_1^?]$ **(see Fig. 7)** Let $\mathcal{T}_{e_1} = (Q_{e_1}, \Sigma, X_{e_1}, \delta_{e_1}, \tau_{e_1}, E_{e_1}, q_{e_1,0}, (F_{e_1,1}, F_{e_1,2}))$. Then

$$\mathcal{T}_e = (Q_{e_1} \cup \{q_{e,0}, f_\varepsilon\}, \Sigma, X_{e_1} \cup \{x_e\}, \delta_e, \tau_e, E_e, q_{e,0}, (\{f_\varepsilon\}, F_{e_1,2}))$$

where

- δ_e is exactly δ_{e_1},
- τ_e comprises the transitions in τ_{e_1}, as well as the transition $\tau_e(q_{e,0}) = ((q_{e_1,0}, f_\varepsilon); ())$,
- E_e inherits E_{e_1} and includes the assignments $E_e(q_{e,0}, \varepsilon, q_{e_1,0})(x_e) = E_e(q_{e,0}, \varepsilon, f_\varepsilon)(x_e) = \varepsilon$, as well as $E_e(q,a,q')(x_e) = x_e a$ for every transition (q,a,q') in $\mathcal{T}_{e_1}$ (where $a \in \Sigma^\varepsilon$).

Note that $F_{e_1,1}$ is not included into $F_{e,1}$ here.

Case $e = [e_1^{??}]$ **(see Fig. 7)** In this case, $\mathcal{T}_{[e_1^{??}]}$ is almost the same as $\mathcal{T}_{[e_1^?]}$. The only difference is that the priorities of the two ε-transitions out of $q_{e,0}$ are swapped, namely, $\tau_e(q_{e,0}) = ((f_\varepsilon, q_{e_1,0}); ())$ here.

Case $e = [e_1^*]$ **(see Fig. 8)** Let $\mathcal{T}_{e_1} = (Q_{e_1}, \Sigma, X_{e_1}, \delta_{e_1}, \tau_{e_1}, E_{e_1}, q_{e_1,0}, (F_{e_1,1}, F_{e_1,2}))$. Then

$$\mathcal{T}_e = (Q_{e_1} \cup \{q_{e,0}, f_{e,1}, f_{e,2}\}, \Sigma, X_e, \delta_e, E_e, \tau_e, q_{e,0}, (\{f_{e,1}\}, \{f_{e,2}\}))$$

where

- δ_e is exactly δ_{e_1},
- τ_e comprises the transitions in τ_{e_1}, as well as the transitions $\tau_e(q_{e,0}) = ((q_{e_1,0}, f_{e,1}); ())$, $\tau_e(f_{e_1,1}) = ((q_{e_1,0}); ())$ for every $f_{e_1,1} \in F_{e_1,1}$, and $\tau_e(f_{e_1,2}) = ((q_{e_1,0}, f_{e,2}); ())$ for every $f_{e_1,2} \in F_{e_1,2}$,

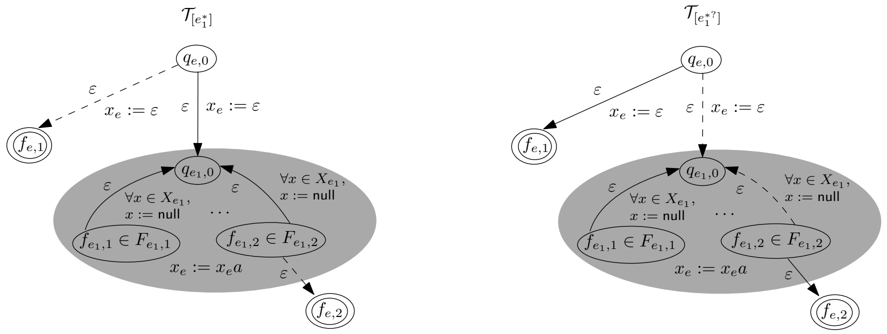


Figure 8: The PSST $\mathcal{T}_{[e_1^*]}$ and $\mathcal{T}_{[e_1^{*?}]}$

- E_e inherits E_{e_1} plus the assignments $E_e(q_{e,0}, \varepsilon, f_{e,1})(x_e) = E_e(q_{e,0}, \varepsilon, q_{e_1,0})(x_e) = \varepsilon$, $E_e(f_{e_1,1}, \varepsilon, q_{e_1,0})(x) = E_e(f_{e_1,2}, \varepsilon, q_{e_1,0})(x) = \mathsf{null}$ for every $f_{e_1,1} \in F_{e_1,1}$, $f_{e_1,2} \in F_{e_1,2}$, and $x \in X_{e_1}$, as well as $E_e(q, a, q')(x_e) = x_e a$ for every transition (q, a, q') in $\mathcal{T}_{e_1}$ with $a \in \Sigma^\varepsilon$. (Intuitively, the values of all the string variables in X_{e_1} are reset when starting a new iteration of e_1.)

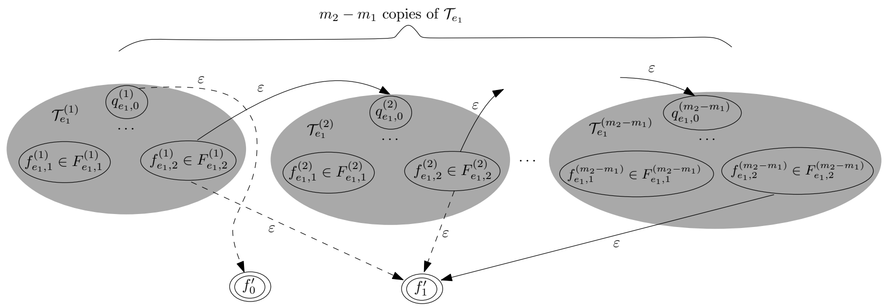
Case $e = [e_1^{*?}]$ **(see Fig. 8)** The construction is almost the same as $e = [e_1^*]$. The only difference is that the priorities of the ε-transitions out of $q_{e,0}$ resp. $f_{e_1,2} \in F_{e_1,2}$ are swapped.

Case $e = [e_1^+]$ We first construct $\mathcal{T}_{e_1}$ and $\mathcal{T}^-_{[e_1^*]}$, where $\mathcal{T}^-_{[e_1^*]}$ is obtained from $\mathcal{T}_{[e_1^*]}$ by dropping the string variable $x_{[e_1^*]}$. Therefore, $\mathcal{T}_{e_1}$ and $\mathcal{T}^-_{[e_1^*]}$ have the same set of string variables, X_{e_1}. Then we construct $\mathcal{T}_e$ by adding into $\mathcal{T}_{e_1} \cdot \mathcal{T}^-_{[e_1^*]}$ a fresh state $q_{e,0}$ as the initial state, and the transitions $\tau_e(q_{e,0}) = ((q_{e_1,0}); ())$, as well as the assignments $E_e(q_{e,0}, \varepsilon, q_{e_1,0})(x_e) = \varepsilon$, $E_e(q, a, q')(x_e) = x_e a$ for every transition (q, a, q') in $\mathcal{T}_{e_1} \cdot \mathcal{T}^-_{[e_1^*]}$.

Case $e = [e_1^{\{m_1, m_2\}}]$ **for** $1 \leq m_1 < m_2$ **(see Fig. 9)** We first construct $\mathcal{T}_{e_1}^{\{m_1\}}$ as the concatenation of m_1 copies of $\mathcal{T}_{e_1}$ (Recall Definition 4.4 for the concatenation of PSSTs). Note that $\mathcal{T}_{e_1}^{\{m_1\}}$ is different from $\mathcal{T}_{e_1^{m_1}}$, the PSST constructed from $e_1^{m_1}$, the concatenation of the expression e_1 for m_1 times. In particular, the set of string variables in $\mathcal{T}_{e_1}^{\{m_1\}}$ is X_{e_1}, which is different from that of $\mathcal{T}_{e_1^{m_1}}$.

Then we construct the PSST $\mathcal{T}_{e_1}^{\{1, m_2 - m_1\}}$ (see Fig. 9), which consists of $m_2 - m_1$ copies of $\mathcal{T}_{e_1}$, denoted by $(\mathcal{T}_{e_1}^{(i)})_{i \in [m_2 - m_1]}$, as well as the ε-transition from $q^{(1)}_{e_1,0}$ to a fresh state f'_0 (of the lowest priority), and the ε-transitions from each $f^{(i)}_{e_1,2} \in F^{(i)}_{e_1,2}$ with $1 \leq i < m_2 - m_1$ to $q^{(i+1)}_{e_1,0}$ (of the highest priority) and a fresh state f'_1 (of the lowest priority). The final states of $\mathcal{T}_{e_1}^{\{1, m_2 - m_1\}}$ are $(\{f'_0\}, \{f'_1\})$. (Intuitively, each $\mathcal{T}_{e_1}^{(i)}$ accepts only nonempty strings, thus $f^{(i)}_{e_1,1} \in F^{(i)}_{e_1,1}$ contains no outgoing transitions in $\mathcal{T}_{e_1}^{\{1, m_2 - m_1\}}$.) Note that the set of string variables in $\mathcal{T}_{e_1}^{\{1, m_2 - m_1\}}$ is still X_{e_1}.

Finally, we construct $\mathcal{T}_e$ from $\mathcal{T}_{e_1}^{\{m_1\}} \cdot \mathcal{T}_{e_1}^{\{1, m_2 - m_1\}}$, the concatenation of $\mathcal{T}_{e_1}^{\{m_1\}}$ and $\mathcal{T}_{e_1}^{\{1, m_2 - m_1\}}$, by adding a fresh state $q_{e,0}$, a string variable x_e, the ε-transition $\tau_e(q_{e,0}) = ((q_{e_1,0}); ())$ (assum-



$m_2 - m_1$ copies of $\mathcal{T}_{e_1}$
$\mathcal{T}_{e_1}^{(1)}$
$q_{e_1,0}^{(1)}$
$f_{e_1,1}^{(1)} \in F_{e_1,1}^{(1)}$
$f_{e_1,2}^{(1)} \in F_{e_1,2}^{(1)}$
$\mathcal{T}_{e_1}^{(2)}$
$q_{e_1,0}^{(2)}$
$f_{e_1,1}^{(2)} \in F_{e_1,1}^{(2)}$
$f_{e_1,2}^{(2)} \in F_{e_1,2}^{(2)}$
$\mathcal{T}_{e_1}^{(m_2-m_1)}$
$q_{e_1,0}^{(m_2-m_1)}$
$f_{e_1,1}^{(m_2-m_1)} \in F_{e_1,1}^{(m_2-m_1)}$
$f_{e_1,2}^{(m_2-m_1)} \in F_{e_1,2}^{(m_2-m_1)}$
ε
f'_0
f'_1

Figure 9: The PSST $\mathcal{T}_{e_1}^{\{1,m_2-m_1\}}$

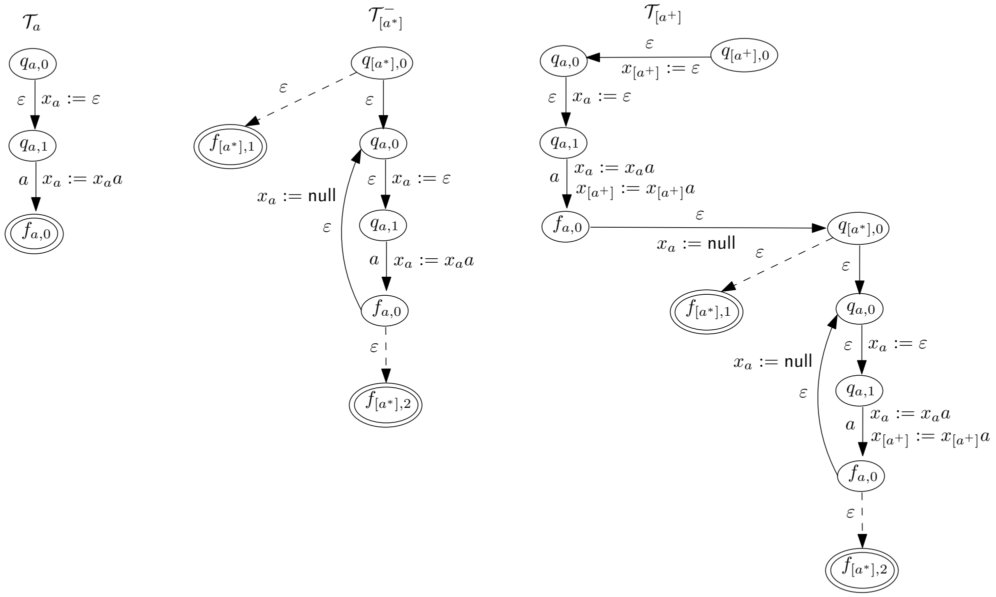


Figure 10: The PSST $\mathcal{T}_e$ for $e = [a^+]$

ing that $q_{e_1,0}$ is the initial state of $\mathcal{T}_{e_1}^{\{m_1\}}$), and also the assignments $E_e(q_{e_0}, \varepsilon, q_{e_1,0})(x_e) = \varepsilon$, as well as $E_e(q, a, q')(x_e) = x_e a$ for each transition (q, a, q') in $\mathcal{T}_{e_1}^{\{m_1\}} \cdot \mathcal{T}_{e_1}^{\{1,m_2-m_1\}}$.

Example 4.5. *Consider RegEx $e = [a^+]$. We first construct $\mathcal{T}_a$ and $\mathcal{T}_{a^*}^-$ (recall that $\mathcal{T}_{a^*}^-$ is obtained from $\mathcal{T}_{a^*}$ by removing the string variable $x_{[a^*]}$, see Fig. 10). Then we construct $\mathcal{T}_e$ from $\mathcal{T}_a \cdot \mathcal{T}_{a^*}^-$ by adding the initial state $q_{[a^+],0}$, the string variable $x_{[a^+]}$, as well as the assignments for $x_{[a^+]}$ (see Fig. 10). Note here only one copy of $\mathcal{T}_{a^*}^-$ is used in $\mathcal{T}_a \cdot \mathcal{T}_{a^*}^-$, since ε is not accepted by $\mathcal{T}_a$.*

The following example illustrates the necessity of splitting final states into two disjoint subsets.

Example 4.6. *Consider RegEx $e = [([a^{*?}])^*]$. If we execute "aaa".match(/(a*?)*/) in node.js, then the result is the array ["aaa", "a"], which means (a*?)* is matched to "aaa" and (a*?) is matched to a. If we did not split the set of final states into two disjoint subsets, we would have*

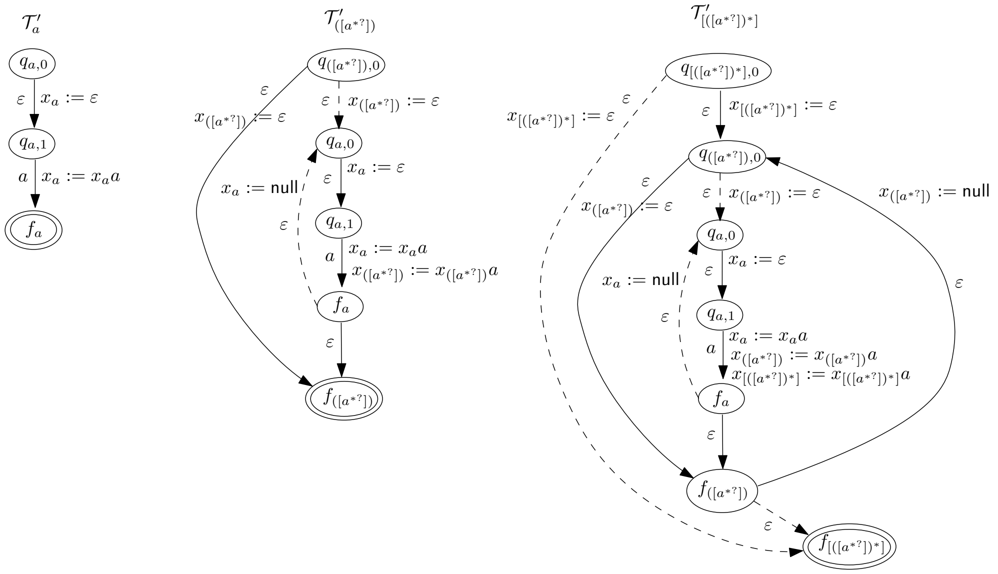


Figure 11: The PSST $\mathcal{T}'_e$ for $e = [([a^{*?}])^*]$ with a single set of final states

obtained a PSST $\mathcal{T}'_e$ *as illustrated in Fig. 11, to simulate the matching of e against words. The accepting run of* $\mathcal{T}'_e$ *on* $w = aaa$ *is*

$$q_{[([a^{*?}])^*]} \xrightarrow{\varepsilon} q_{([a^{*?}]),0} \xrightarrow{\varepsilon} f_{([a^{*?}])} \xrightarrow{\varepsilon} q_{([a^{*?}]),0} \xrightarrow{\varepsilon} q_{a,0} \xrightarrow{\varepsilon} q_{a,1} \xrightarrow{a} f_a \xrightarrow{\varepsilon} f_{([a^{*?}])} \xrightarrow{\varepsilon} q_{([a^{*?}]),0} \xrightarrow{\varepsilon} q_{a,0} \xrightarrow{\varepsilon} q_{a,1} \xrightarrow{a} f_a \xrightarrow{\varepsilon} f_{([a^{*?}])} \xrightarrow{\varepsilon} q_{([a^{*?}]),0} \xrightarrow{\varepsilon} q_{a,0} \xrightarrow{\varepsilon} q_{a,1} \xrightarrow{a} f_a \xrightarrow{\varepsilon} f_{([a^{*?}])} \xrightarrow{\varepsilon} q_{([a^{*?}]),0} \xrightarrow{\varepsilon} f_{([a^{*?}])} \xrightarrow{\varepsilon} f_{[([a^{*?}])^*]},$$

where $x_e = aaa$ *and* $x_{([a^{*?}])} = \varepsilon$*, namely, e is matched to "aaa" and* $([a^{*?}])$ *is matched to* ε*. Therefore, the semantics of e defined by* $\mathcal{T}'_e$ *is* inconsistent *with semantics of /(a*?)*/ in node.js. Intuitively, the semantics of /(a*?)*/ in node.js requires that either it is matched to* ε *in whole and the subexpression a*? is* not *matched at all, or it is matched to a concatenation of* non-empty *strings each of which matches a*?. This semantics can be captured by (adapted)* PSST*s where the set of final states is split into two disjoint subsets.*

Validation experiments for the formal semantics We have defined RegEx-string matching by constructing PSSTs. In the sequel, we conduct experiments to validate the formal semantics against the actual JavaScript RegEx-string matching.

Let $\mathscr{O}$ denote the set of RegEx operators: alternation $+$, concatenation $\cdot$, optional ?, lazy optional ??, Kleene star $*$, lazy Kleene star $*$?, Kleene plus $+$, lazy Kleene plus $+$?, repetition $\{m_1, m_2\}$, and lazy repetition $\{m_1, m_2\}$?. Moreover, let $\mathscr{O}^2$ (resp. $\mathscr{O}^3$) denote the set of pairs (resp. triples) of operators from $\mathscr{O}$. Aiming at a good coverage of different syntactical ingredients of RegEx, we generate regular expressions for every element of $\mathscr{O}^{\leq 3} = \mathscr{O} \cup \mathscr{O}^2 \cup \mathscr{O}^3$. As arguments of these operators, we consider the following character sets: $\mathbb{S} = \{$a, ..., z$\}$, $\mathbb{C} = \{$A, ..., Z$\}$, $\mathbb{D} = \{0, \ldots, 9\}$, and $\mathbb{O}$, the set of ASCII symbols not belonging to $\mathbb{S} \cup \mathbb{C} \cup \mathbb{D}$. Intuitively, these character sets correspond to JavaScript character classes [a-z], [A-Z], [0-9], and [ˆa-zA-Z0-9] (where ˆ denotes complement). Moreover, for the regular expression generated for each element of $\mathscr{O}^{\leq 3}$, we set the subexpression corresponding to its first component as the capturing group. For instance, for the pair $(*?, *)$, we generate the RegEx $[([\mathbb{S}^{*?}])^*]$. In the end, we generate $10 + 10 * 10 + 10 * 10 * 10 = 1110$ RegExes.

For each generated RegEx e, we construct a PSST $\mathcal{T}_e$, whose output corresponds to the matching of the first capturing group in e. Moreover, we generate from $\mathcal{T}_e$ an input string w as well as the corresponding output w'. We require that the length of w is no less than some threshold (e.g., 10), in order to avoid the empty string and facilitate a meaningful comparison with the actual semantics of JavaScript regular-expression matching. Let reg be the JavaScript regular expression corresponding to e. Then we execute the following JavaScript program $\mathcal{P}_{e,w}$,

```
var x = w; console.log(x.match(reg)[1]);
```

and confirm that its output is equal to w', thus validating that the formal semantics of RegEx-string matching defined by PSSTs is consistent with the actual semantics of JavaScript match function. For instance, for the RegEx expression $[([\mathbb{S}^{*?}])^*]$, we generate from the $\mathcal{T}_e$ the input string $w = aaaaaaaaaa$, together with the output a. Then we generate the JavaScript program from reg and w, execute it, and obtain the same output a.

In all the generated RegExs, we confirm the consistency of the formal semantics of RegEx-string matching defined by PSSTs with the actual JavaScript semantics, namely, for each RegEx e, the output of the PSST $\mathcal{T}_e$ on w is equal to the output of the JavaScript program $\mathcal{P}_{e,w}$.

4.3 Modeling string functions by PSSTs

The $\mathsf{extract}$, $\mathsf{replace}$ and $\mathsf{replaceAll}$ functions can be accurately modeled using PSSTs. That is, we can reduce satisfiability of our string logic to satisfiability of a logic containing only concatenation, PSST transductions, and membership of regular languages.

Lemma 4.7. *The satisfiability of* STR *reduces to the satisfiability of boolean combinations of formulas of the form* $z = x \cdot y$*,* $y = \mathcal{T}(x)$*, and* $x \in \mathcal{A}$*, where* $\mathcal{T}$ *is a PSST and* $\mathcal{A}$ *is an FA.*

First, observe that regular constraints (aka membership queries) $x \in e$ can be reduced to FA membership queries $x \in \mathcal{A}$ using standard techniques. Features such a greediness and capture groups do not affect whether a word matches a RegEx, they only affect *how* a string matches it. Thus, for regular constraints, these features can be ignored and a standard translation from regular expressions to finite automata can be used.

The $\mathsf{extract}_{i,e}$ function can be defined by a PSST $\mathcal{T}_{i,e}$ obtained from the PSST $\mathcal{T}_e$ (see Section 4.2) by removing all string variables, except $x_{e'}$, where e' is the subexpression of e corresponding to the ith capturing group, and setting the output expression of the final states as $x_{e'}$.

We give a sketch of the encoding of $\mathsf{replaceAll}$ here. Full formal details are given in the long version of the paper [23]. The encoding of $\mathsf{replace}$ is almost identical to that of $\mathsf{replaceAll}$.

A call $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$ replaces every match of pat by a value determined by the replacement string rep. Recall, rep may contain references $\$i$, $\$^{\leftarrow}$, or $\$^{\rightarrow}$. The first step in our reduction to PSSTs is to eliminate the special references $\$0$, $\$^{\leftarrow}$, and $\$^{\rightarrow}$. In essence, this simplification uses PSST transductions to insert the contextual information needed by $\$^{\leftarrow}$ and $\$^{\rightarrow}$ alongside each substring that will be replaced. Then, the call to $\mathsf{replaceAll}$ can be rewritten to include this information in the match, and use standard references ($\$i$) in the replacement string. The reference $\$0$ can be eliminated by wrapping each pattern with an explicit capturing group.

We show informally how to construct the PSST for $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}$ where all the references in rep are of the form $\$i$ with $i > 0$. The full reduction is given in the long version of the paper [23].

Let $\mathsf{rep} = w_1 \$ i_1 w_2 \cdots w_k \$ i_k w_{k+1}$. For each $j \in [k]$, we introduce a *fresh* string variable y_j. Let us use $\mathsf{rep}[(y_1, \cdots, y_k)/(\$ i_1, \cdots, \$ i_k)]$ to denote the sequence $w_1 y_1 w_2 \cdots w_k y_k w_{k+1}$. For instance, if $\mathsf{rep} = a\$1a\$2a\$1a$, then $\mathsf{rep}[(y_1, y_2, y_3)/(1, 2, 1)] = a y_1 a y_2 a y_3 a$. Moreover, let $e'_{i_1}, \ldots, e'_{i_k}$ be the subexpressions of pat corresponding to the i_1th, …, i_kth capturing groups. Note here we use mutually distinct fresh variables $y_1, \cdots, y_k$ for $\$ i_1, \cdots, \$ i_k$, even if i_j and $i_{j'}$ may be equal for $j \neq j'$. We make this choice for the purpose of satisfying the copyless property [5] of PSSTs, which leads to improved complexity results in some cases (discussed in the sequel). If we tried to use the same variable for the different occurrences of the same reference – then the resulting transition in the encoding below would not be copyless. Moreover, the construction below guarantees that the values of different variables for the multiple occurrences of the same reference are actually the same.

Suppose $\mathcal{T}_{\mathsf{pat}} = (Q_{\mathsf{pat}}, \Sigma, X_{\mathsf{pat}}, \delta_{\mathsf{pat}}, \tau_{\mathsf{pat}}, E_{\mathsf{pat}}, q_{\mathsf{pat},0}, (F_{\mathsf{pat},1}, F_{\mathsf{pat},2}))$. Then $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ is obtained from $\mathcal{T}_{\mathsf{pat}}$ by adding the fresh string variables $y_1, \cdots, y_k$ and a fresh state q'_0 such that (see Fig. 12)

- $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ goes from q'_0 to $q_{\mathsf{pat},0}$ via an ε-transition of higher priority than the non-ε-transitions, in order to search the first match of pat starting from the current position,

- when $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ stays at q'_0, it keeps appending the current letter to the end of x_0, which stores the output of $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$,

- starting from $q_{\mathsf{pat},0}$, $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ simulates $\mathcal{T}_{\mathsf{pat}}$ and stores the matches of capturing groups of pat into the string variables (in particular, the matches of the i_1th, …, i_kth capturing groups into the string variables $x_{e'_{i_1}}, \cdots, x_{e'_{i_k}}$ respectively), moreover, for each $j \in [k]$, y_j is updated in the same way as $x_{e'_{i_j}}$ (in particular, for each transition (q, a, q') in $\mathcal{T}_{\mathsf{pat}}$ such that $E_{\mathsf{pat}}(q, a, q')(x_{e'_{i_j}}) = x_{e'_{i_j}} a$, we have $E_{\mathsf{pat}}(q, a, q')(y_j) = y_j a$),

- when the first match of pat is found, $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ goes from $f_{\mathsf{pat},1} \in F_{\mathsf{pat},1}$ or $f_{\mathsf{pat},2} \in F_{\mathsf{pat},2}$ to q'_0 via an ε-transition, it then appends the replacement string, which is given by $\mathsf{rep}[(y_1, \cdots, y_k)/(\$ i_1, \cdots, \$ i_k)]$, to the end of x_0, resets the values of all the string variables, except x_0, to null, and keeps searching for the next match of pat.

It may be observed that the PSST will be copyless. That is, the value of a variable is not copied to two or more variables during a transition. In all but the last case, variables are only copied to themselves, via assignments of the form $x_{e'} := x_{e'} a$, $x_{e'} := x_{e'}$, $x_{e'} := \varepsilon$, or $x_{e'} := \mathsf{null}$. In the final case, when a replacement is made, the assignments are $x_0 := x_0 w_1 y_1 w_2 \cdots w_k y_k w_{k+1}$ and $x' := \mathsf{null}$ for all the variables $x' \in X_{\mathsf{pat}} \cup \{y_1, \cdots, y_k\}$. Again, only one copy of the value of each variable is retained.

Copyful PSSTs are only needed when removing $\$^{\leftarrow}$ and $\$^{\rightarrow}$ from the replacement strings. To see this, consider the prefix preceding the first replacement in a string. If $\$^{\leftarrow}$ appears in the replacement string, this prefix will be copied an unbounded number of times (once for each matched and replaced substring). Conversely, references of the form $\$ i$ are "local" to a single match. By having a separate variable for each occurence of $\$ i$ in the replacement string, we can avoid having to make copies of the values of the variables.

5 A Propagation-Based Calculus for String Constraints

We now introduce our calculus for solving string constraints in STR (see Table 1), state its correctness, and observe that it gives rise to a decision procedure for the fragment $\mathsf{STR}_{\mathsf{SL}}$ of

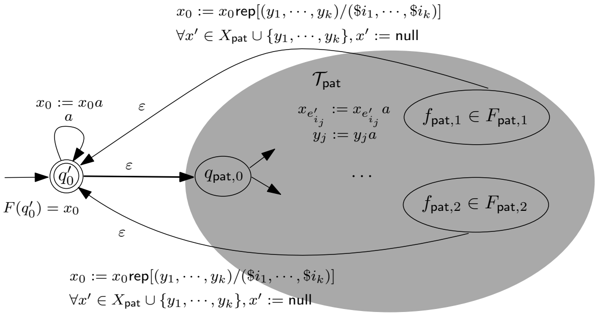

$x_0 := x_0\mathsf{rep}[(y_1, \cdots, y_k)/(\$i_1, \cdots, \$i_k)]$
$\forall x' \in X_{\mathsf{pat}} \cup \{y_1, \cdots, y_k\}, x' := \mathsf{null}$
$\mathcal{T}_{\mathsf{pat}}$
$x_0 := x_0 a$
a
ε
$x_{e'_{i_j}} := x_{e'_{i_j}} a$
$y_j := y_j a$
$f_{\mathsf{pat},1} \in F_{\mathsf{pat},1}$
q'_0
ε
$q_{\mathsf{pat},0}$
$\cdots$
$F(q'_0) = x_0$
$f_{\mathsf{pat},2} \in F_{\mathsf{pat},2}$
ε
$x_0 := x_0\mathsf{rep}[(y_1, \cdots, y_k)/(\$i_1, \cdots, \$i_k)]$
$\forall x' \in X_{\mathsf{pat}} \cup \{y_1, \cdots, y_k\}, x' := \mathsf{null}$

Figure 12: The PSST $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$

Table 1: Rules of the one-sided sequent calculus. A term e^c denotes the complement of a regular expression e, i.e., $\mathcal{L}(e^c) = \Sigma^* \setminus \mathcal{L}(e)$.

$$\wedge\ \frac{\Gamma, \varphi, \psi}{\Gamma, \varphi \wedge \psi} \qquad \neg\vee\ \frac{\Gamma, \neg\varphi, \neg\psi}{\Gamma, \neg(\varphi \vee \psi)} \qquad \vee\ \frac{\Gamma, \varphi \qquad \Gamma, \psi}{\Gamma, \varphi \vee \psi} \qquad \neg\wedge\ \frac{\Gamma, \neg\varphi \qquad \Gamma, \neg\psi}{\Gamma, \neg(\varphi \wedge \psi)} \qquad \neg\neg\ \frac{\Gamma, \varphi}{\Gamma, \neg\neg\varphi}$$

$$\notin\ \frac{\Gamma, x \in e^c}{\Gamma, x \notin e} \qquad \neq\ \frac{\Gamma, x \neq y, y = f(x_1, \ldots, x_n)}{\Gamma, x \neq f(x_1, \ldots, x_n)} \ \text{ where } y \text{ is fresh} \qquad \textsc{Cut}\ \frac{\Gamma, x \in e \qquad \Gamma, x \in e^c}{\Gamma}$$

$$=\text{-}\textsc{Prop}\ \frac{\Gamma, x \in e, x = y, y \in e}{\Gamma, x \in e, x = y} \qquad \neq\text{-}\textsc{Subsume}\ \frac{\Gamma, x \in e_1, y \in e_2}{\Gamma, x \in e_1, x \neq y, y \in e_2} \ \text{ if } \mathcal{L}(e_1) \cap \mathcal{L}(e_2) = \emptyset$$

$$=\text{-}\textsc{Prop-Elim}\ \frac{\Gamma, x \in e, y \in e}{\Gamma, x \in e, x = y} \ \text{ if } |\mathcal{L}(e)| = 1 \qquad \neq\text{-}\textsc{Prop-Elim}\ \frac{\Gamma, x \in e, y \in e^c}{\Gamma, x \in e, x \neq y} \ \text{ if } |\mathcal{L}(e)| = 1$$

$$\textsc{Close}\ \frac{}{\Gamma, x \in e_1, \ldots, x \in e_n} \qquad \text{if } \mathcal{L}(e_1) \cap \cdots \cap \mathcal{L}(e_n) = \emptyset$$

$$\textsc{Subsume}\ \frac{\Gamma, x \in e_1, \ldots, x \in e_n}{\Gamma, x \in e, x \in e_1, \ldots, x \in e_n} \qquad \text{if } \mathcal{L}(e_1) \cap \cdots \cap \mathcal{L}(e_n) \subseteq \mathcal{L}(e)$$

$$\textsc{Intersect}\ \frac{\Gamma, x \in e}{\Gamma, x \in e_1, \ldots, x \in e_n} \qquad \text{if } \begin{array}{l} n > 1 \text{ and} \\ \mathcal{L}(e_1) \cap \cdots \cap \mathcal{L}(e_n) = \mathcal{L}(e) \end{array}$$

$$\textsc{Fwd-Prop}\ \frac{\Gamma, x \in e, x = f(x_1, \ldots, x_n), x_1 \in e_1, \ldots, x_n \in e_n}{\Gamma, x = f(x_1, \ldots, x_n), x_1 \in e_1, \ldots, x_n \in e_n} \qquad \text{if } \mathcal{L}(e) = f(\mathcal{L}(e_1), \ldots, \mathcal{L}(e_n))$$

$$\textsc{Fwd-Prop-Elim}\ \frac{\Gamma, x \in e, x_1 \in e_1, \ldots, x_n \in e_n}{\Gamma, x = f(x_1, \ldots, x_n), x_1 \in e_1, \ldots, x_n \in e_n} \qquad \text{if } \begin{array}{l} \mathcal{L}(e) = f(\mathcal{L}(e_1), \ldots, \mathcal{L}(e_n)) \\ \text{and } |\mathcal{L}(e)| = 1 \end{array}$$

$$\textsc{Bwd-Prop}\ \frac{\left\{\Gamma, x \in e, x = f(x_1, \ldots, x_n), x_1 \in e_1^i, \ldots, x_n \in e_n^i\right\}_{i=1}^k}{\Gamma, x \in e, x = f(x_1, \ldots, x_n)} \ \text{ if } \begin{array}{l} f^{-1}(\mathcal{L}(e)) = \\ \bigcup_{i=1}^k \left(\mathcal{L}(e_1^i) \times \cdots \times \mathcal{L}(e_n^i)\right) \end{array}$$

straightline formulas. The calculus is based on the principle of propagating regular language constraints by computing images and pre-images of string functions. We deliberately keep the calculus minimalist and focus on the main proof rules; for an implementation, the calculus has to be complemented with a suitable strategy for applying the rules, as well as standard SMT optimizations such as non-chronological back-tracking and conflict-driven learning. An

$$\wedge^*\;\dfrac{\textsc{Fwd-Prop}\;\dfrac{\textsc{Fwd-Prop}\;\dfrac{\textsc{Close}\;\dfrac{}{x \in a^+\Sigma^*, x = y \cdot z, y \in a^+, z \in \Sigma^*, x \in b^+(a^c)^*, x = \mathsf{replaceAll}_{a,b}(x)}}{x \in a^+\Sigma^*, x = y \cdot z, y \in a^+, z \in \Sigma^*, x = \mathsf{replaceAll}_{a,b}(x)}}{x = y \cdot z, y \in a^+, z \in \Sigma^*, x = \mathsf{replaceAll}_{a,b}(x)}}{x = y \cdot z \wedge y \in a^+ \wedge z \in \Sigma^* \wedge x = \mathsf{replaceAll}_{a,b}(x)}$$

Figure 13: Proof of unsatisfiability for (3) in Example 5.1

$$\wedge^*\;\dfrac{\textsc{Bwd-Prop}\;\dfrac{\textsc{Cut}\;\dfrac{\text{FPE}\;\dfrac{\text{FPE}\;\dfrac{\textsc{Subsume}^*\;\dfrac{x \in a, z \in a, y \in \epsilon, r \in b}{x \in a, z \in a, y \in \epsilon, r \in b, \ldots}}{z \in a, y \in \epsilon, x \in a, r = \mathsf{replaceAll}_{a,b}(x), \ldots}}{z \in a, y \in \epsilon, x = y \cdot z, \ldots} \qquad \dfrac{\vdots}{z \in a^c, \ldots}}{y \in \epsilon, z \in a^+, x = y \cdot z, x \in a^+, \ldots} \qquad \dfrac{\vdots}{y \in a^+, z \in a^*, \ldots}}{x = y \cdot z, x \in a^+, r = \mathsf{replaceAll}_{a,b}(x)}}{x = y \cdot z \wedge x \in a^+ \wedge r = \mathsf{replaceAll}_{a,b}(x)}$$

Figure 14: Proof of satisfiability for (4) in Example 5.2. FPE stands for Fwd-Prop-Elim

implementation also has to choose a suitable effective representation of RegEx membership constraints, for instance using finite-state automata.[3] In particular, we use the fact that—for membership—RegEx can be complemented. We denote the complement of e in a membership constraint by e^c. Our calculus is parameterized in the set of considered string functions; in this paper, we work with the set $\{\cdot, \mathsf{extract}, \mathsf{replace}, \mathsf{replaceAll}\}$ consisting of concatenation, extraction, and replacement, but this set can be extended by other functions for which images and/or pre-images can be computed (see Section 5.2).

5.1 Sequents and Examples

The calculus operates on *one-sided sequents,* and can be interpreted as a sequent calculus in the sense of Gentzen [34] in which all formulas are located in the antecedent (to the left of the turnstile $\vdash$). A one-sided sequent is a finite set $\Gamma \subseteq \mathsf{STR}$ of string constraints. For sake of presentation, we write sequents as lists of formulas separated by comma, and $\Gamma, \varphi_1, \ldots, \varphi_n$ for the union $\Gamma \cup \{\varphi_1, \ldots, \varphi_n\}$. We say that a sequent Γ is *unsatisfiable* if $\bigwedge \Gamma$ is unsatisfiable. Our calculus is refutational and has the purpose of either showing that some initial sequent Γ is unsatisfiable, or that it is satisfiable by constructing a solution for it. A solution is a sequent $x_1 \in w_1, x_2 \in w_2, \ldots, x_n \in w_n$ that defines the values of string variables using RegExes that only consist of single words.

Example 5.1. *We first illustrate the calculus by showing unsatisfiability of the constraint*[4]*:*

$$x = y \cdot z \wedge y \in a^+ \wedge z \in \Sigma^* \wedge x = \mathsf{replaceAll}_{a,b}(x) \tag{3}$$

[3] Recall features such as greediness do not need to be modeled for simple membership queries as they do not change the accepted language.

[4] Note here for convenience, in the regular constraints $x \in e$, we write e as in classical regular expressions and do not strictly follow the syntax of STR, since in this case, only the language defined by e matters.

To this end, we construct a proof tree that has (3) *as its root, by applying proof rules until all proof goals have been closed (Fig. 13). The proof is growing upward, and is built by first eliminating the conjunctions $\wedge$, resulting in a list of formulas. Next, we apply the rule* Fwd-Prop *for* forward-propagation *of a regular expression constraint. Given that $y \in a^+, z \in \Sigma^*$, from the equation $x = y \cdot z$ we can conclude that $x \in a^+\Sigma^*$. From $x \in a^+\Sigma^*$ and $x = \mathsf{replaceAll}_{a,b}(x)$, we can next conclude that $x \in b^+(a^c)^*$, i.e., x starts with b and cannot contain the letter a. Finally, the proof can be closed because the languages $a^+\Sigma^*$ and $b^+(a^c)^*$ are disjoint.*

Example 5.2. *We next consider the case of a satisfiable formula in* $\mathsf{STR}_{\mathsf{SL}}$*:*

$$x = y \cdot z \wedge x \in a^+ \wedge r = \mathsf{replaceAll}_{a,b}(x) \tag{4}$$

Fig. 14 shows how a solution can be constructed for this formula. The strategy is to first derive constraints for the variables y, z whose value is not determined by any equation. Given that $x \in a^+$, from the equation $x = y \cdot z$ we can derive that either $y \in \epsilon, z \in a^+$ or $y \in a^+, z \in a^*$, using rule Bwd-Prop*. We focus on the left branch $y \in \epsilon, z \in a^+$. Since propagation is not able to derive further information for y, z, and no contradiction was detected, at this point we can conclude satisfiability of* (4)*. To construct a solution, we pick an arbitrary value for z satisfying the constraint $z \in a^+$, and use* Cut *to add the formula $z \in a$ to the branch. Again following the left branch, we can then use* Fwd-Prop-Elim *to evaluate $x = y \cdot z$ and add the formula $x \in a$, and after that $r \in b$ due to $r = \mathsf{replaceAll}_{a,b}(x)$. Finally,* Subsume *is used to remove redundant RegEx constraints from the proof goal. The resulting sequent (top-most sequent on the left-most branch) is a witness for satisfiability of* (4)*.*

5.2 Proofs and Proof Rules

More formally, proof rules are relations between a finite list of sequents (the premises), and a single sequent (the conclusion). Proofs are finite trees growing upward, in which each node is labeled with a sequent, and each non-leaf node is related to the node(s) directly above it through an instance of a proof rule. A proof branch is a path from the proof root to a leaf. A branch is closed if a closure rule (a rule without premises) has been applied to its leaf, and open otherwise. A proof is closed if all of its branches are closed.

The proof rules of the calculus are shown in Table 1. The first row shows standard proof rules to handle Boolean operators; see, e.g., [35]. Rule $\not\in$ turns negated membership predicates into positive ones through complementation, and rule $\neq$ negative function applications into positive ones. As a result, only disequalities between string variables remain. The rule Cut can be used to introduce case splits, and is mainly needed to extract solutions once propagation has converged (as shown in Example 5.2).

The next four rules handle equations between string variables. Rule $=$-Prop propagates RegEx constraints from the left-hand side to the right-hand side of an equation; $=$-Prop-Elim in addition removes the equation in the case where the propagated constraint has a unique solution. The rule $\neq$-Prop-Elim similarly turns a singleton RegEx for the left-hand side of a disequality into a RegEx constraint on the right-hand side. As a convention, we allow application of $=$-Prop, $=$-Prop-Elim, and $\neq$-Prop-Elim in both directions, left-to-right and right-to-left of equalities/disequalities. Finally, $\neq$-Subsume eliminates disequalities that are implied by the RegEx constraints of a proof goal.

The rule Close closes proof branches that contain contradictory RegEx constraints, and is the only closure rule needed in our calculus. Subsume removes RegEx constraints that are implied by other constraints in a sequent, and INTERSECT replaces multiple RegExes with a single constraint.

The last three rules handle applications of functions $f \in \{\cdot, \mathsf{extract}, \mathsf{replace}, \mathsf{replaceAll}\}$ through propagation. Rule FWD-PROP defines forward propagation, and adds a RegEx constraint $x \in e$ for the value of a function by propagating constraints about the arguments. The RegEx e encodes the image of the argument RegExes under f:

Definition 5.3 (Image)**.** *For an n-ary string function $f : \Sigma^* \times \cdots \times \Sigma^* \to \Sigma^*$ and languages $L_1, \ldots, L_n \subseteq \Sigma^*$, we define the* image *of $L_1, \ldots, L_n$ under f as $f(L_1, \ldots, L_n) = \{f(w_1, \ldots, w_n) \in \Sigma^* \mid w_1 \in L_1, \ldots, w_n \in L_n\}$.*

Forward propagation is often useful to prune proof branches. It is easy to see, however, that the images of regular languages under the functions considered in this paper are not always regular; for instance, $\mathsf{replace}_{\mathsf{pat},\$0\$0}$ can map regular languages to context-sensitive languages. In such cases, the side condition of FWD-PROP cannot be satisfied by any RegEx e, and the rule is not applicable.

Rule FWD-PROP-ELIM handles the special case of forward propagation producing a singleton language. In this case, the function application is not needed for further reasoning and can be eliminated. This rule is mainly used during the extraction of solutions (as shown in Example 5.2).

Rule BWD-PROP defines the dual case of backward propagation, and derives RegEx constraints for function arguments from a constraint about the function value. The argument constraints encode the *pre-image* of the propagated language:

Definition 5.4 (Pre-image)**.** *For an n-ary string function $f : \Sigma^* \times \cdots \times \Sigma^* \to \Sigma^*$ and a language $L \subseteq \Sigma^*$, we define the* pre-image *of L under f as the relation $f^{-1}(L) = \{(w_1, \ldots, w_n) \in (\Sigma^*)^n \mid f(w_1, \ldots, w_n) \in L\}$.*

A **key result** of the paper is that pre-images of regular languages under the functions considered in the paper can always be represented in the form $\bigcup_{i=1}^{k}(\mathcal{L}(e_1^i) \times \cdots \times \mathcal{L}(e_n^i))$, i.e., they are *recognizable languages* [20]. This implies that BWD-PROP is applicable whenever a RegEx constraint for the result of a function application exists, and prepares the ground for the decidability result in the next section. For concatenation, recognizability was shown in [3, 24]. This paper contributes the corresponding result for all functions defined by PSSTs:

Lemma 5.5 (Pre-image of regular languages under PSSTs)**.** *Given a PSST $\mathcal{T} = (Q_T, \Sigma, X, \delta_T, \tau_T, E_T, q_{0,T}, F_T)$ and an FA $\mathcal{A} = (Q_A, \Sigma, \delta_A, q_{0,A}, F_A)$, we can compute an FA $\mathcal{B} = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$ in exponential time such that $\mathscr{L}(\mathcal{B}) = \mathcal{R}_{\mathcal{T}}^{-1}(\mathscr{L}(\mathcal{A}))$.*

The proof of Lemma 5.5 is given in the long version of the paper [23]. Moreover, we have already shown in Lemma 4.7 that extract, replace, and replaceAll can be reduced to PSSTs. We can finally observe that the calculus is sound:

Lemma 5.6 (Soundness)**.** *The sequent calculus defined by Table 1 is sound: (i) the root of a closed proof is an unsatisfiable sequent; and (ii) if a proof has an open branch that ends with a solution $x_1 \in w_1, x_2 \in w_2, \ldots, x_n \in w_n$, then the assignment $\{x_1 \mapsto w_1, x_2 \mapsto w_2, \ldots, x_n \mapsto w_n\}$ is a satisfying assignment of the root sequent.*

Proof. By showing that each of the proof rules in Table 1 is an equivalence transformation: the conclusion of a proof rule is equivalent to the disjunction of the premises. □

5.3 Decision Procedure for $\mathsf{STR}_{\mathsf{SL}}$

One of the main results of this paper is the decidability of the $\mathsf{STR}_{\mathsf{SL}}$ fragment of straightline formulas including concatenation, extract, replace, and replaceAll:

Theorem 5.7. *Satisfiability of* $\mathsf{STR}_{\mathsf{SL}}$ *formulas is decidable.*

Proof. We define a terminating strategy to apply the rules in Table 1 to formulas in the $\mathsf{STR}_{\mathsf{SL}}$ fragment. The resulting proofs will either be closed, proving unsatisfiability, or have at least one satisfiable goal containing a solution:

- *Phase 1:* apply the Boolean rules (first row of Table 1) to eliminate Boolean operators.
- *Phase 2:* apply rule Bwd-Prop to all regex constraints and all function applications on all proof branches. Whenever contradictory regex constraints occur in a proof goal, use Close to close the branch. Also apply =-Prop to systematically propagate constraints across equations. This phase terminates because $\mathsf{STR}_{\mathsf{SL}}$ formulas are acyclic.

  If all branches are closed as a result of Phase 2, the considered formula is unsatisfiable; otherwise, we can conclude satisfiability, and Phase 3 will extract a solution.
- *Phase 3:* select an open branch of the proof. On this branch, determine the set I of input variables, which are the string variables that do not occur as left-hand side of equations or function applications. For every $x \in I$, use rule Cut to introduce an assignment $x \in w$ that is consistent with the regex constraints on x. Then systematically apply Subsume, =-Prop, Fwd-Prop-Elim to evaluate remaining formulas and produce a solution.

□

Complexity analysis. Because the pre-image computation for each PSST incurs an exponential blow-up in the size of the input automaton $\mathcal{A}$, the aforementioned decision procedure has a non-elementary complexity in the worst-case. In fact, this is optimal and a matching lower-bound is given in the long version of the paper [23].

When $\$^{\leftarrow}$ and $\$^{\rightarrow}$ are not used, the PSSTs in the reduction are copyless, and the exponential blow-up in the size of the input FA $\mathcal{A}$ can be avoided. That is, the pre-image automaton $\mathcal{B}$ such that $\mathscr{L}(\mathcal{B}) = \mathcal{R}_{\mathcal{T}}^{-1}(\mathscr{L}(\mathcal{A}))$ is exponential only in the size of $\mathcal{T}$ and not the size of $\mathcal{A}$. Hence, the exponentials do not stack on top of each other during the backwards analysis and the non-elementary blow-up is not necessary. Since the PSST $\mathcal{T}$ may be exponential in the size of the underlying regular expression, we may compute automata that are up to double exponential in size. The states of these automata can be stored in exponential space and the transition relation can be computed on the fly, giving an exponential space algorithm. More details are given in the long version of this paper [23].

Moreover, since the number of PSSTs is usually small in the path constraints of string-manipulating programs, the performance of the decision procedure is actually good on the benchmarks we tested, with the average running time per query a few seconds (see Section 6).

6 Implementation and Experiments

We extend the open-source solver OSTRICH [24] to support for STR based on the calculus. In particular, it can decide the satisfiability of $\mathsf{STR}_{\mathsf{SL}}$ formulas. The extension can handle most of the other operations of the SMT-LIB theory of Unicode strings.[5]

[5] http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml

6.1 Implementation

Our solver extends classical regular expressions in SMT-LIB with the indexed re.capture and re.reference operators, which denote capturing groups and references to them. We also add re.*?, re.+?, re.opt? and re.loop? as the lazy counterparts of Kleene star, plus operator, optional operator and loop operator.

Three new string operators are introduced to make use of these extended regular expressions: str.replace_cg, str.replace_cg_all, and str.extract. The operators str.replace_cg and str.replace_cg_all are the counterparts of the standard str.replace_re and replace_re_all operators, and allow capturing groups in the match pattern and references in the replacement pattern. E.g., the following constraint swaps the first name and the last name, as in Example 1.1:

```
(= w (str.replace_cg_all v
      (re.++ ((_ re.capture 1)
                 (re.+ (re.union (re.range "A" "Z") (re.range "a" "z"))))
             (str.to.re " ")
             ((_ re.capture 2)
                 (re.+ (re.union (re.range "A" "Z") (re.range "a" "z")))))
      (re.++ (_ re.reference 2) (_ re.reference 1))))
```

The replacement string is written as a regular expression only containing the operators re.++, str.to_re, and re.reference. The use of string variables in the replacement parameter is not allowed, since the resulting transformation could not be mapped to a PSST.

The indexed operator str.extract implements extract$_{i,e}$ in STR. For instance,

```
  ((_ str.extract 1)
      (re.++ (re.*? re.allchar)((_ re.capture 1) (re.+ (re.range "a" "z")) re.all))
      x)
```

extracts the left-most, longest sub-string of lower-case characters from a string x.

Our implementation is able to handle *anchors* as well, although for reasons of presentation we did not introduce them as part of our formalism. Anchors match certain positions of a string without consuming any input characters. In most programming languages, it is common to use ^ and $ in regular expressions to signify the start and end of a string, respectively. We add re.begin-anchor and re.end-anchor for them. Our implementation correctly models the semantics of anchors and is able to solve constraints containing these operators.

OSTRICH implements the procedure in Theorem 5.7, and focuses on SL formulas. The three string operators mentioned above will be converted into an equivalent PSST (see the full version of the paper [23]). OSTRICH then iteratively applies the propagation rules from Section 5 to derive further RegEx constraints, and eventually either detect a contradiction, or converge and find a fixed-point. For straight-line formulas, the existence of a fixed-point implies satisfiability, and a solution can then be constructed as described in Section 5. In addition, similar to other SMT solvers, OSTRICH applies simplification rules (e.g., Fwd-Prop-Elim, =-Prop, Subsume, Close, etc in Table 1) to formulas before invoking the SL procedure. This enables OSTRICH to solve some formulas outside of the SL fragment, but is not a complete procedure for non-SL formulas.

6.2 Experimental evaluation

Our experiments have the purpose of answering the following main questions:
R1: How does OSTRICH compare to other solvers that can handle real-world regular expressions, including greedy/lazy quantifiers and capturing groups?

```
(declare-fun x () String)
(define-fun  y () String (str.replace_cg_all x <re1> <repl>))
(push 1)
(assert (str.in.re x (re.++ re.all <re1> re.all)))
(assert (str.in.re y (re.++ re.all <re2> re.all)))
(check-sat) (get-model)
(pop 1) (push 1)
(assert (str.in.re x (re.++ re.all <re1> re.all)))
(assert (not (str.in.re y
          (re.++ re.all <re2> re.all))))
(check-sat) (get-model)
(pop 1) (push 1)
(assert (not (str.in.re x (re.++ re.all <re1> re.all))))
(check-sat) (get-model)
(pop 1)
```

```
function fun(x) {
   if(/<re1>/.test(x)) {
      var y = x.replace(/<re1>/g, <repl>);
      if(/<re2>/.test(y))
        console.log("1");
      else
        console.log("2");
   }
   else
       console.log("3");
}

var S$ = require("S$");
var x = S$.symbol("x", "");
fun(x);
```

Figure 15: Harnesses with replace-all: SMT-LIB for OSTRICH (left), and JavaScript for ExpoSE (right).

R2: How does OSTRICH perform in the context of symbolic execution, the primary application of string constraint solving?

*For **R1**:* There are no standard string benchmarks involving RegExes, and we are not aware of other constraint solvers supporting capturing groups, neither among the SMT nor the CP solvers. The closest related work is the algorithm implemented in ExpoSE, which applies Z3 [27] for solving string constraints, but augments it with a refinement loop to approximate the RegEx semantics.[6] For **R1**, we compared OSTRICH with ExpoSE+Z3 on 98,117 RegExes taken from [26].

For each regular expression, we created four harnesses: two in SMT-LIB, as inputs for OSTRICH, and two in JavaScript, as inputs for ExpoSE+Z3. The two harnesses shown in Fig. 15 use one of the regular expressions from [26] (`<re1>`) in combination with the replace-all function to simulate typical string processing; `<re2>` is the fixed pattern `[a-z]+`, and `<repl>` the replacement string `"$1"`. The three paths of the JavaScript function `fun` correspond to the three queries in the SMT-LIB script, so that a direct comparison can be made between the results of the SMT-LIB queries and the set of paths covered by ExpoSE+Z3. The other two harnesses are similar to the ones in Fig. 15, but use the match function instead of replace-all, and contain four queries and four paths, respectively.

The results of this experiment are shown in Table 2. OSTRICH is able to answer all four queries in 95,175 of the match benchmarks (97%), and all three queries in 89,794 of the replace-all benchmarks (91.5%). The errors in 1,134 cases (resp., 1,135 cases) are mainly due to back-references in `<re1>`, which are not handled by OSTRICH. ExpoSE+Z3 can cover 228,888 paths of the match problems in total (91.2% of the number of sat results of OSTRICH), although the runtime of ExpoSE+Z3 is on average 18x higher than that of OSTRICH. For replace, ExpoSE+Z3 can cover 173,007 paths (66.7%), showing that this class of constraints is harder; the runtime of ExpoSE+Z3 is on average 8x higher than that of OSTRICH. Overall, even taking into account that ExpoSE+Z3 has to analyze JavaScript code, as opposed to the SMT-LIB given to OSTRICH, the experiments show that OSTRICH is a highly competitive solver for RegExes.

*For **R2**:* For this experiment, we integrated OSTRICH into the symbolic execution tool Aratha [8]. We compare Aratha+OSTRICH with ExpoSE+Z3 on the regression test suite of

[6] We considered replacing Z3 with OSTRICH in ExpoSE for the experiments. However, ExpoSE integrates Z3 using its C API, and changing to OSTRICH, with native support for capturing groups, would have required the rewrite of substantial parts of ExpoSE.

	OSTRICH						**ExpoSE+Z3**				
	# queries solved within 60s						# paths covered within 60s				
	0	1	2	3	4	#Err	0	1	2	3	4
Match	422	249	751	386	95,175	1,134	3,333	9,274	36,916	48,594	0
(98,117	Average time: 1.57s						Average time: 28.0s				
benchm.)	Total #sat: 250,947, #unsat: 132,662						Total #paths covered: 228,888				
Replace	4,170	2,463	555	89,794	—	1,135	5,281	18,221	69,059	5,556	—
(98,117	Average time: 6.62s						Average time: 55.0s				
bench.)	Total #sat: 259,354, #unsat: 13,601						Total #paths covered: 173,007				

Table 2: The number of queries answered by OSTRICH, and number of paths covered by ExpoSE+Z3, in **R1**. Experiments were done on an AMD Opteron 2220 SE machine, running 64-bit Linux and Java 1.8. Runtime per benchmark was limited to 60s wall-clock time, 2GB memory, and the number of tests executed concurrently by ExpoSE+Z3 to 1. Average time is wall-clock time per benchmark, timeouts count as 60s.

	Aratha+OSTRICH						**ExpoSE+Z3**					
	# paths covered within 120s						# paths covered within 120s					
	0	1	2	3	≥4	#Err	0	1	2	3	≥4	#T.O.
ExpoSE	14	9	9	2	15	2	14	9	9	2	15	6
(49 programs)	Average time: **4.66**s						Average time: 57.44s					
	Total #paths covered:124						Total #paths covered:121					
Match	3	7	12	6	0	0	3	8	12	5	0	6
(28 programs)	Average time: **5.19**s						Average time: 60.26s					
	Total #paths covered: 49						Total #paths covered: 47					
Replace	12	20	6	0	0	0	15	21	2	0	0	23
(38 programs)	Average time: **4.14**s						Average time: 95.34s					
	Total #paths covered: 32						Total #paths covered: 25					

Table 3: Results of Expose+Z3 and Aratha+OSTRICH on Javascript programs for **R2**. Experiments were done on an Intel(R)-Core(TM)-i5-8265U-CPU-@1.60GHz cpu, running 64-bit Linux and Java 1.8. Runtime was limited to 120s wall-clock time. Average time is wall-clock time needed per benchmark, and counts timeouts as 120s. #Err is the number of non-straight-line path constraints that OSTRICH fails to deal with and #T.O is the number of timeouts. Note that some paths may have already been covered before T.O.

ExpoSE [43], as well as a collection of other JavaScript programs containing match or replace functions extracted from Github. In Table 3, we can see that Aratha+OSTRICH can within 120s cover slightly more paths than ExpoSE+Z3. Aratha+OSTRICH can discover feasible paths much more quickly than ExpoSE+Z3, however: on all three families of benchmarks, Aratha+OSTRICH terminates on average in less than 10s, and it discovers all paths within 20s. ExpoSE+Z3 needs the full 120s for 35 of the programs ("T.O." in the table), and it finds new paths until the end of the 120s. Since ExpoSE+Z3 handles the replace-all operation by unrolling, it is not able to prove infeasibility of paths involving such operations, and will therefore not terminate on some programs. Overall, the experiments indicate that OSTRICH is more efficient than the CEGAR-augmented symbolic execution for dealing with RegExes.

7 Related Work

Modelling and Reasoning about RegEx. Variants and extensions of regular expressions to capture their usage in programming languages have received attention in both theory and practice. In formal language theory, regular expressions with capturing groups and backreferences were considered in [18, 19] and also more recently in [30, 49, 12, 31], where the expressibility issues and decision problems were investigated. Nevertheless, some basic features of these regular expression, namely, the non-commutative union and the greedy/lazy semantics of Kleene star/plus, were not addressed therein. In the software engineering community, some empirical studies were recently reported for these regular expressions, including portability across different programing languages [26] and DDos attacks [51], as well as how programmers write them in practice [46].

Prioritized finite-state automata and transducers were proposed in [11]. Prioritized finite-state transducers add indexed brackets to the input string in order to identify the matches of capturing groups. It is hard—if not impossible—to use prioritized finite-state transducers to model replace(all) function, e.g., swapping the first and last name as in Example 1.1. In contrast, PSSTs store the matches in string variables, which can then be referred to, allowing us to conveniently model the match and replace(all) function. Streaming string transducers were used in [62] to solve the straight-line string constraints with concatenation, finite-state transducers, and regular constraints.

String Constraint Solving. As we discussed Section 1, there has been much research focussing on string constraint solving algorithms, especially in the past ten years. Solvers typically use a combination of techniques to check the satisfiability of string constraints, including word-based methods, automata-based methods, and unfolding-based methods like the translation to bit-vector constraints. We mention among others the following string solvers: Z3 [27], CVC4 [41], Z3-str/2/3/4 [61, 60, 13, 14], ABC [17], Norn [3], Trau [16, 2, 1], OSTRICH [24], S2S [40], Qzy [25], Stranger [58], Sloth [36, 4], Slog [57], Slent [56], Gecode+S [50], G-Strings [9], HAMPI [39], and S3 [54]. Most modern string solvers provide support of concatenation and regular constraints. The push (e.g. see [32, 33, 39, 48, 42, 54]) towards incorporating other functions—e.g. length, string-number conversions, replace, replaceAll—in a string theory is an important theme in the area, owing to the desire to be able to reason about complex real-world string-manipulating programs. These functions, among others, are now part of the SMT-LIB Unicode Strings standard.[7]

To the best of our knowledge, there is currently no solver that supports RegEx features like greedy/lazing matching or capturing groups (apart from our own solver OSTRICH). This was remarked in [44], where the authors try to amend the situation by developing ExpoSE — a dynamic symbolic execution engine — that maps path constraints in JavaScript to Z3. The strength of ExpoSE is in a thorough modelling of RegEx features, some of which (including backreferences) we do not cover in our string constraint language and string solver OSTRICH. However, the features that we do not cover are also rare in practice, according to [44] — in fact, around 75% of all the RegEx expressions found in their benchmarks across 415,487 NPM packages can be covered in our fragment. The strength of OSTRICH against ExpoSE is in a substantial improvement in performance (by 30–50 fold) and precision. ExpoSE does not terminate even for simple examples (e.g. for Example 1.1 and Example 1.2), which can be solved by our solver within a few seconds.

For string constraint solving in general, we refer the readers to the recent survey [7]. In this work, we consider a string constraint language which is undecidable in general, and propose

[7] See http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml

a propagation-based calculus to solve the constraints. However, we also identified a straight-line fragment including concatenation, extract, replace(All) which turns to be decidable. Our decision procedure extends the backward-reasoning approach in [24], where only standard one-way and two-way finite-state transducers were considered.

8 Conclusion

The challenge of reasoning about string constraints with regular expressions stems from functions like match and replace that exploit features like capturing groups, not to mention the subtle deterministic (greedy/lazy) matching. Our results provide the first string solving method that natively supports and effectively handles RegEx, which is a large order of magnitude faster than the symbolic execution engine ExpoSE [44] tailored to constraints with regular expressions, which is at the moment the only available method for reasoning about string constraints with regular expressions. Our solver OSTRICH relies on two ingredients: (i) Prioritized Streaming String Transducers (used to capture subtle non-standard semantics of RegEx, while being amenable to analysis), and (ii) a sequent calculus that exploits nice closure and algorithmic properties of PSST, and performs a kind of propagation of regular constraints by means of taking post-images or pre-images. We have also carried out thorough empirical studies to validate our formalization of RegEx as PSST with respect to JavaScript semantics, as well as to measure the performance of our solver. Finally, although the satisfiability of the constraint language is undecidable, we have also shown that our solver terminates (and therefore is complete) for the straight-line fragment.

Several avenues for future work are obvious. Firstly, it would be interesting to see how ExpoSE could be used in combination with our solver OSTRICH. This would essentially lift OSTRICH to a symbolic execution engine (i.e. working at the level of programs).

Secondly, we could incorporate other features of RegEx that are not in our framework, e.g., lookahead and backreferences. To handle lookahead, we may consider alternating variants of PSSTs. Alternating automata [21] are effectively able to branch and run parallel checks on the input. We will need to model the subtle interplay between lookahead and references. Backreferences could be handled by allowing some inspection of variable contents during transducer runs. There is some precedent for this in higher-order automata [45, 28], whose stacks non-trivially store and use data. However, the pre-image of string functions supporting RegEx with backreferences will not be regular in general, and emptiness of intersection of RegEx with backreferences is undecidable [19]. Decidability can be recovered in some cases [31]. We may study these cases or look for incomplete algorithms.

Finally, since strings do not live in isolation in a real-world program, there is a real need to also extend our work with other data types, in particular the integer data type.

Acknowledgments We thank Johannes Kinder and anonymous referees for their helpful feedback. T. Chen is supported by Birkbeck BEI School under Grant No. ARTEFACT, National Natural Science Foundation of China under Grant No. 62072309, The State Key Laboratory of Novel Software Technology, Nanjing University under Grant No. KFKT2018A16. M. Hague and A. Flores-Lamas are supported by the Engineering and Physical Sciences Research Council under Grant No. EP/T00021X/1. S. Kan and 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). P. Rümmer is supported by the Swedish Research Council (VR) under grant 2018-04727, by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), by the Wallenberg project UPDATE, and by grants from Microsoft and Amazon Web Services. Z. Wu is supported by the National Natural Science Foundation of China under Grant No. 61872340.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Flatten and conquer: a framework for efficient analysis of string constraints. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 602–617, 2017.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Trau: SMT solver for string constraints. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–5. IEEE, 2018.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *CAV*, pages 150–166, 2014.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukás Holík, and Petr Janku. Chain-free string constraints. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, pages 277–293, 2019.

[5] Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*, pages 1–12, 2010.

[6] Rajeev Alur and Jyotirmoy V. Deshmukh. Nondeterministic streaming string transducers. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.

[7] Roberto Amadini. A survey on string constraint solving. *CoRR*, abs/2002.02376, 2020.

[8] Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Constraint programming for dynamic symbolic execution of javascript. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2019.

[9] Roberto Amadini, Graeme Gange, Peter J. Stuckey, and Guido Tack. A novel approach to string constraint solving. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017.

[10] Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. In Zoltán Ésik and Zoltán Fülöp, editors, *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014*, volume 151 of *EPTCS*, pages 109–123, 2014.

[11] Martin Berglund and Brink van der Merwe. On the semantics of regular expression parsing in the wild. *Theoretical Computer Science*, 679:69 – 82, 2017.

[12] Martin Berglund and Brink van der Merwe. Regular expressions with backreferences re-examined. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*, pages 30–41. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017.

[13] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59, 2017.

[14] Berzish, Murphy. *Z3str4: A Solver for Theories over Strings.* PhD thesis, 2021.

[15] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem.* Perspectives in

Mathematical Logic. Springer, 1997.

[16] Diep Bui and contributors. Z3-trau. `https://github.com/diepbp/z3-trau`, 2019.

[17] Tevfik Bultan and contributors. Abc string solver. `https://github.com/vlab-cs-ucsb/ABC`, 2015.

[18] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003.

[19] Benjamin Carle and Paliath Narendran. On extended regular expressions. In Adrian-Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings*, volume 5457 of *Lecture Notes in Computer Science*, pages 279–289. Springer, 2009.

[20] Olivier Carton, Christian Choffrut, and Serge Grigorieff. Decision problems among the main subfamilies of rational relations. *ITA*, 40(2):255–275, 2006.

[21] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[22] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *PACMPL*, 2(POPL):3:1–3:29, 2018.

[23] Taolue Chen, Matthew Hague, Zhilei Han, Denghang Hu, Alejandro Flores-Lamas, Anthony Lin, Shanglong Kan, Philipp Ruemmer, and Zhilin Wu. Solving string constraints with regex-dependent functions through transducers with priorities and variables. *CoRR*, 2021.

[24] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *PACMPL*, 3(POPL), January 2019.

[25] Arlen Cox and Jason Leasure. Model checking regular language constraints, 2017.

[26] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 443–454, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[28] Joost Engelfriet. Iterated stack automata and complexity classes. *Inf. Comput.*, 95(1):21–75, 1991.

[29] Emmanuel Filiot and Pierre-Alain Reynier. Copyful streaming string transducers. In Matthew Hague and Igor Potapov, editors, *Reachability Problems - 11th International Workshop, RP 2017, London, UK, September 7-9, 2017, Proceedings*, volume 10506 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 2017.

[30] Dominik D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory Comput. Syst.*, 53(2):159–193, 2013.

[31] Dominik D. Freydenberger and Markus L. Schmid. Deterministic regular expressions with back-references. *J. Comput. Syst. Sci.*, 105:1–39, 2019.

[32] Vijay Ganesh and Murphy Berzish. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. *CoRR*, abs/1605.09442, 2016.

[33] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. Word equations with length constraints: What's decidable? In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, pages 209–226, 2012.

[34] Gerhard Gentzen. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation, "Investigations into Logical Deduction," in [52].

[35] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University

Press, 2009.

[36] Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL):4:1–4:32, 2018.

[37] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security Symposium*, 2011.

[38] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[39] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, 2012.

[40] Quang Loc Le and Mengda He. A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 350–372. Springer, 2018.

[41] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV*, pages 646–662, 2014.

[42] Anthony W. Lin and Pablo Barceló. String solving with word equations and transducers: Towards a logic for analysing mutation XSS. POPL '16, pages 123–136. ACM, 2016.

[43] Blake Loring, Duncan Mitchell, and Johannes Kinder. Expose: practical symbolic execution of standalone javascript. In Hakan Erdogmus and Klaus Havelund, editors, *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, pages 196–199. ACM, 2017.

[44] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 425–438. ACM, 2019.

[45] A. N. Masilov. Multilevel magazine automata. *Probl. Peredachi Inf.*, 12(1):55–62, 1976.

[46] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 415–426. IEEE Press, 2019.

[47] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[48] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 513–528, 2010.

[49] Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inf. Comput.*, 249:1–17, 2016.

[50] Joseph D. Scott, Pierre Flener, Justin Pearson, and Christian Schulte. Design and implementation of bounded-length sequence variables. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 51–67. Springer, 2017.

[51] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 361–376, USA, 2018. USENIX Association.

[52] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. North-Holland, Amsterdam, 1969.

[53] Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[54] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS*, pages 1232–1243, 2014.

[55] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. Progressive reasoning over recursively-defined strings. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 218–240. Springer, 2016.

[56] Hung-En Wang, Shih-Yu Chen, Fang Yu, and Jie-Hong R. Jiang. A symbolic model checking approach to the analysis of string and length constraints. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 623–633. ACM, 2018.

[57] Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. String analysis via automata manipulation with logic circuit representation. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2016.

[58] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS*, pages 154–157, 2010. Benchmark can be found at http://www.cs.ucsb.edu/~vlab/stranger/.

[59] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Form. Methods Syst. Des.*, 44(1):44–70, 2014.

[60] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 235–254. Springer, 2015.

[61] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: a Z3-based string solver for web application analysis. In *ESEC/SIGSOFT FSE*, pages 114–124, 2013.

[62] Qizhen Zhu, Hitoshi Akama, and Yasuhiko Minamide. Solving string constraints with streaming string transducers. *Journal of Information Processing*, 27:810–821, 2019.

A Appendix

A.1 Construction of PSST from RegEx

Case $e = \emptyset$ (see Figure 16) $\mathcal{T}_\emptyset = (\{q_{\emptyset,0}\}, \Sigma, \{x_\emptyset\}, \delta_\emptyset, \tau_\emptyset, E_\emptyset, q_{\emptyset,0}, (\emptyset, \emptyset))$, where there are no transitions out of $q_{\emptyset,0}$, namely, $\delta_\emptyset(q_{\emptyset,0}, a) = ()$ for every $a \in \Sigma$, $\tau_\emptyset(q_{\emptyset,0}) = ((); ())$, and $E_\emptyset$ is vacuous here.

Case $e = \varepsilon$ (see Figure 16) $\mathcal{T}_\varepsilon = (\{q_{\varepsilon,0}, f_{\varepsilon,0}\}, \Sigma, \{x_\varepsilon\}, \delta_\varepsilon, \tau_\varepsilon, E_\varepsilon, q_{\varepsilon,0}, (\{f_{\varepsilon,0}\}, \emptyset))$, where $\tau_\varepsilon(q_{\varepsilon,0}) = ((f_{\varepsilon,0}); ())$, and $E_\varepsilon(q_{\varepsilon,0}, \varepsilon, f_{\varepsilon,0})(x) = \varepsilon$. Note $F_2 = \emptyset$ here.

Case $e = a$ (see Figure 16) $\mathcal{T}_a = (\{q_{a,0}, q_{a,1}, f_{a,0}\}, \Sigma, \{x_a\}, \delta_a, \tau_a, E_a, q_{a,0}, (\emptyset, \{f_{a,0}\}))$, where $\tau_a(q_{a,0}) = ((q_{a,1}); ())$, $\delta_a(q_{a,1}, a) = (f_{a,0})$, $E_a(q_{a,0}, \varepsilon, q_{a,1})(x_a) = \varepsilon$, and $E_a(q_{a,1}, a, f_{a,0})(x_a) = x_a a$. Note $F_1 = \emptyset$ here.

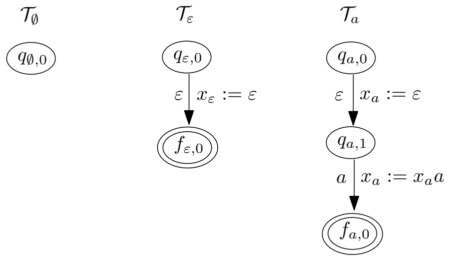


Figure 16: The PSST $\mathcal{T}_\emptyset$, $\mathcal{T}_\varepsilon$, and $\mathcal{T}_a$

Case $e = [e_1^{+?}]$ Then $\mathcal{T}_e$ is constructed from $\mathcal{T}_{e_1}$ and $\mathcal{T}^-_{[e_1^{*?}]}$, similarly to the aforementioned construction of $\mathcal{T}_{[e_1^+]}$.

Case $e = [e_1^{\{m_1,m_2\}?}]$ for $1 \leq m_1 < m_2$ (see Figure 17) Then $\mathcal{T}_e$ is constructed as the concatenation of $\mathcal{T}_{e_1}^{\{m_1\}}$ and $\mathcal{T}_{e_1}^{\{1,m_2-m_1\}?}$, where $\mathcal{T}_{e_1}^{\{1,m_2-m_1\}?}$ is illustrated in Figure 17, which is the same as $\mathcal{T}_{e_1}^{\{1,m_2-m_1\}}$ in Figure 9, except that the priorities of the ε-transition from $q^{(1)}_{e_1,0}$ to f'_0 has the highest priority and the priorities of the ε-transitions out of each $f^{(i)}_{e_1,2} \in F^{(i)}_{e_1,2}$ to f'_1 are swapped.

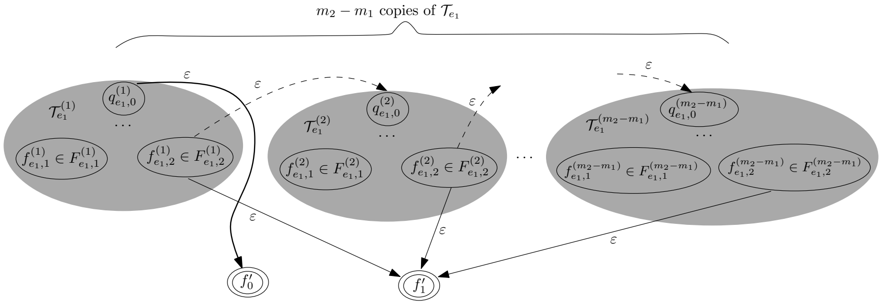

$m_2 - m_1$ copies of $\mathcal{T}_{e_1}$
$\mathcal{T}_{e_1}^{(1)}$
$q^{(1)}_{e_1,0}$
$\cdots$
$f^{(1)}_{e_1,1} \in F^{(1)}_{e_1,1}$
$f^{(1)}_{e_1,2} \in F^{(1)}_{e_1,2}$
ε
ε
$\mathcal{T}_{e_1}^{(2)}$
$q^{(2)}_{e_1,0}$
$\cdots$
$f^{(2)}_{e_1,1} \in F^{(2)}_{e_1,1}$
$f^{(2)}_{e_1,2} \in F^{(2)}_{e_1,2}$
ε
$\cdots$
ε
$\mathcal{T}_{e_1}^{(m_2-m_1)}$
$q^{(m_2-m_1)}_{e_1,0}$
$\cdots$
$f^{(m_2-m_1)}_{e_1,1} \in F^{(m_2-m_1)}_{e_1,1}$
$f^{(m_2-m_1)}_{e_1,2} \in F^{(m_2-m_1)}_{e_1,2}$
ε
ε
ε
f'_0
f'_1

Figure 17: The PSST $\mathcal{T}_{e_1}^{\{1,m_2-m_1\}?}$

A.2 From extract, replace and replaceAll to PSSTs

Lemma 4.7. The satisfiability of STR reduces to the satisfiability of boolean combinations of formulas of the form $z = x \cdot y$, $y = \mathcal{T}(x)$, and $x \in \mathcal{A}$, where $\mathcal{T}$ is a PSST and $\mathcal{A}$ is an FA.

The proof is in two steps: first we remove $\$0$, $\$^{\leftarrow}$, and $\$^{\rightarrow}$, then we encode the remaining string functions with PSSTs.

A.2.1 Removing Special References

The first step in our proof is to remove the special references $\$0$, $\$^{\leftarrow}$, and $\$^{\rightarrow}$ from the replacement strings. These can be replaced in a series of steps, leaving only PSST transductions and replacement strings with only simple references ($\$i$). We will just consider replaceAll as replace is almost identical.

First, to remove $\$0$, suppose we have a statement $y := \mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$ with $\$0$ in rep. We simply substitute $y := \mathsf{replaceAll}_{\mathsf{pat}',\mathsf{rep}'}(x)$ where $\mathsf{pat}' = (\mathsf{pat})$, and $\mathsf{rep}' = \mathsf{rep}[\$1/\$0, \$2/\$1, \ldots, \$(k+1)/\$k]$. That is, we make the complete match an explicit (first) capture, which shifts the indexes of the remaining capturing groups by 1.

Now suppose we have a statement $y := \mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$ with $\$^{\leftarrow}$ or $\$^{\rightarrow}$ in rep. We replace it with the following statements, explained below, where $y_1, \ldots, y_5$ are fresh variables.

$$\begin{aligned}
y_1 &:= \mathsf{replaceAll}_{(\mathsf{pat}),\langle\$1\rangle}(x);\\
y_2 &:= \mathcal{T}_{\langle}(y_1);\\
y_3 &:= \mathcal{T}_{\mathrm{rev}}(y_2);\\
y_4 &:= \mathcal{T}_{\rangle}(y_3);\\
y_5 &:= \mathcal{T}_{\mathrm{rev}}(y_4);\\
y &:= \mathsf{replaceAll}_{\mathsf{pat}',\mathsf{rep}'}(y_5)
\end{aligned}$$

The first step is to mark the matched parts of the string with $\langle$ and $\rangle$ brackets (where $\langle$ and $\rangle$ are not part of the main alphabet). This is achieved by the first replaceAll.

Next, we use a PSST $\mathcal{T}_{\langle}$ that passes over the marked word. This is a copyful PSST that simply stores the word read so far into a variable X, except for the $\langle$ and $\rangle$ characters. It also has an output variable O, to which it also copies each character directly, except $\langle$. When it encounters $\langle$ it appends to O the string $\langle X \langle$. That is, it puts the entire string preceding each $\langle$ into the output, surrounded by $\langle$ at the start and end. This is copyful since X will be copied to both X and O in this step. For example, suppose the input string were $ab\langle c\rangle d\langle e\rangle f$, the output of $\mathcal{T}_{\langle}$ would be $ab\langle\underline{ab}\langle c\rangle d\langle\underline{abcd}\langle e\rangle f$. We have underlined the strings inserted for readability.

The next step is to do the same for $\rangle$. To achieve this we first reverse the string so that a PSST can read the end of the string first. A similar transduction to $\mathcal{T}_{\langle}$ is performed before the string is reversed again. In our example the resulting string is $ab\langle\underline{ab}\langle c\rangle\underline{def}\rangle d\langle\underline{abcd}\langle e\rangle\underline{f}\rangle f$.

Finally, we have $y := \mathsf{replaceAll}_{\mathsf{pat}',\mathsf{rep}'}(y_5)$ where $\mathsf{pat}' = \langle(\Sigma^{*?})\langle\mathsf{pat}\rangle(\Sigma^{*?})\rangle$, and

$$\mathsf{rep}' = \mathsf{rep}[\$1/\$^{\leftarrow}, \$2/\$1, \ldots \$(k+1)/\$k, \$(k+2)/\$^{\rightarrow}] \ .$$

That is, by inserting the preceding and succeeding text directly next to each match, we can use simple references $\$i$ instead of $\$^{\leftarrow}$ and $\$^{\rightarrow}$.

A.2.2 Encoding string functions as PSSTs

Once $\$0$, $\$^{\leftarrow}$, and $\$^{\rightarrow}$ are removed from the replacement strings, then the string functions can be replaced by PSSTs.

Lemma A.1. *For each string function $f = \mathsf{extract}_{i,e}$, $\mathsf{replace}_{\mathsf{pat},\mathsf{rep}}$, or $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}$ without $\$^{\leftarrow}$ or $\$^{\rightarrow}$ in the replacements strings, a PSST $\mathcal{T}_f$ can be constructed such that*

$$\mathcal{R}_f = \{(w, w') \mid w' = f(w)\}.$$

Proof. The $\mathsf{extract}_{i,e}$ can function be defined by a PSST $\mathcal{T}_{i,e}$ obtained from the PSST $\mathcal{T}_e$ (see Section 4.2) by removing all the string variables, except the string variable $x_{e'}$, where e' is the subexpression of e corresponding to the ith capturing group, and setting the output expression of the final states as $x_{e'}$.

Next, we give the construction of the PSST for $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}$ where all the references in rep are of the form $\$i$.

Recall $\mathsf{rep} = w_1 \$ i_1 w_2 \cdots w_k \$ i_k w_{k+1}$. Let $e'_{i_1}, \ldots, e'_{i_k}$ denote the subexpressions of pat corresponding to the i_1th, …, i_kth capturing groups of pat. Then $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}} = (Q_{\mathsf{pat}} \cup \{q'_0\}, \Sigma, X', \delta', \tau', E', q'_0, F')$ where

- $q'_0 \notin Q_{\mathsf{pat}}$,
- $X' = \{x_0\} \cup X_{\mathsf{pat}}$,
- $F'(q'_0) = x_0$, and $F'(q')$ is undefined for every $q' \in Q_{\mathsf{pat}}$,
- δ' comprises the transitions in δ_{pat}, and the transition $\delta'(q'_0, a) = (q'_0)$ for $a \in \Sigma$,
- τ' comprises the transitions in τ_{pat}, the transitions $\tau'(q'_0) = ((q_{\mathsf{pat},0}); ())$, $\tau'(f_{\mathsf{pat},1}) = ((q'_0); ())$ and $\tau'(f_{\mathsf{pat},2}) = ((q'_0); ())$ for $f_{\mathsf{pat},1} \in F_{\mathsf{pat},1}$ and $f_{\mathsf{pat},2} \in F_{\mathsf{pat},2}$,
- E' inherits E_{pat}, and also includes the assignments $E'(q'_0, a, q'_0)(x_0) = x_0 a$ for $a \in \Sigma$, $E'(f, \varepsilon, q'_0)(x_0) = x_0 \mathsf{rep}[x_{e'_{i_1}}/\$i_1, \ldots, x_{e'_{i_k}}/i_k]$ and $E'(f, \varepsilon, q'_0)(x) = \mathsf{null}$ for every $f \in F_{\mathsf{pat},1} \cup F_{\mathsf{pat},2}$ and $x \in X_{\mathsf{pat}}$.

□

The construction of the PSST for $\mathsf{replace}_{\mathsf{pat},\mathsf{rep}}$ is similar and illustrated in Fig. 18. The details are omitted.

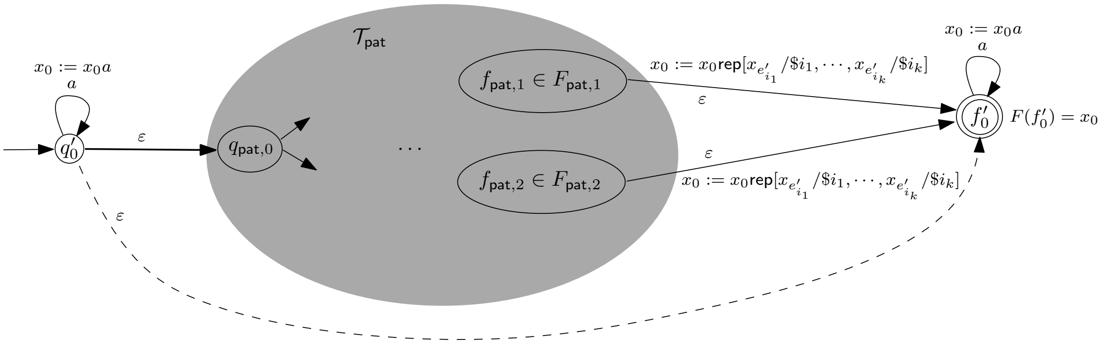

$\mathcal{T}_{\mathsf{pat}}$
$x_0 := x_0 a$
a
q'_0
ε
$q_{\mathsf{pat},0}$
$\cdots$
$f_{\mathsf{pat},1} \in F_{\mathsf{pat},1}$
$f_{\mathsf{pat},2} \in F_{\mathsf{pat},2}$
$x_0 := x_0\mathsf{rep}[x_{e'_{i_1}}/\$i_1, \cdots, x_{e'_{i_k}}/\$i_k]$
ε
ε
$x_0 := x_0\mathsf{rep}[x_{e'_{i_1}}/\$i_1, \cdots, x_{e'_{i_k}}/\$i_k]$
ε
$x_0 := x_0 a$
a
f'_0
$F(f'_0) = x_0$

Figure 18: The PSST $\mathcal{T}_{\mathsf{replace}_{\mathsf{pat},\mathsf{rep}}}$

A.3 Proof of Lemma 5.5

Lemma 5.5. *Given a PSST $\mathcal{T} = (Q_T, \Sigma, X, \delta_T, \tau_T, E_T, q_{0,T}, F_T)$ and an FA $\mathcal{A} = (Q_A, \Sigma, \delta_A, q_{0,A}, F_A)$, we can compute an FA $\mathcal{B} = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$ in exponential time such that $\mathscr{L}(\mathcal{B}) = \mathcal{R}_{\mathcal{T}}^{-1}(\mathscr{L}(\mathcal{A}))$.*

We prove Lemma 5.5 in the sequel.

Let $\mathcal{T} = (Q_T, \Sigma, X, \delta_T, \tau_T, E_T, q_{0,T}, F_T)$ be a PSST and $\mathcal{A} = (Q_A, \Sigma, \delta_A, q_{0,A}, F_A)$ be an FA. Without loss of generality, we assume that $\mathcal{A}$ contains no ε-transitions. For convenience, we use $\mathcal{E}(\tau_T)$ to denote $\{(q, q') \mid q' \in \tau_T(q)\}$. For convenience, for $a \in \Sigma$, we use $\delta_A^{(a)}$ to denote the relation $\{(q, q') \mid (q, a, q') \in \delta_A\}$.

To illustrate the intuition of the proof of Lemma 5.5, let us start with the following natural idea of firstly constructing a PFA $\mathcal{B}$ for the pre-image: $\mathcal{B}$ simulates a run of $\mathcal{T}$ on w, and, for each $x \in X$, records an $\mathcal{A}$-abstraction of the string stored in x, that is, the set of state pairs $(p, q) \in Q_A \times Q_A$ such that starting from p, $\mathcal{A}$ can reach q after reading the string stored in x. Specifically, the states of $\mathcal{B}$ are of the form (q, ρ) with $q \in Q$ and $\rho \in (\mathcal{P}(Q_A \times Q_A))^X$. Moreover, the priorities of $\mathcal{B}$ inherit those of $\mathcal{T}$. The PFA $\mathcal{B}$ is then transformed to an equivalent FA by simply dropping all priorities. We refer to this FA as $\mathcal{B}'$.

Nevertheless, it turns out that this construction is flawed: A string w is in $\mathcal{R}_{\mathcal{T}}^{-1}(\mathscr{L}(\mathcal{A}))$ iff the (unique) accepting run of $\mathcal{T}$ on w produces an output w' that is accepted by $\mathcal{A}$. However, a string w is accepted by $\mathcal{B}'$ iff *there is a run of $\mathcal{T}$ on w, not necessarily of the highest priority*, producing an output w' that is accepted by $\mathcal{A}$.

While the aforementioned natural idea does not work, we choose to construct an FA $\mathcal{B}$ that simulates the *accepting* run of $\mathcal{T}$ on w, and, for each $x \in X$, records an $\mathcal{A}$-abstraction of the string stored in x, that is, the set of state pairs $(p, q) \in Q_A \times Q_A$ such that starting from p, $\mathcal{A}$ can reach q after reading the string stored in x. To simulate the accepting run of $\mathcal{T}$, it is necessary to record all the states accessible through the runs of higher priorities to ensure the current run is indeed the accepting run of $\mathcal{T}$ (of highest priority). Moreover, $\mathcal{B}$ also remembers the set of ε-transitions of $\mathcal{T}$ after the latest non-ε-transition to ensure that no transition occurs twice in a sequence of ε-transitions of $\mathcal{T}$.

Specifically, each state of $\mathcal{B}$ is of the form (q, ρ, Λ, S), where $q \in Q_T$, $\rho \in (\mathcal{P}(Q_A \times Q_A))^X$, $\Lambda \subseteq \mathcal{E}(\tau_T)$, and $S \subseteq Q_T$. For a state (q, ρ, Λ, S), our intention for S is that the states in it are those that can be reached in the runs of higher priorities than the current run, by reading the same sequence of letters and applying the ε-transitions as many as possible. Note that when recording in S all the states accessible through the runs of higher priorities, we do not take the non-repetition of ε-transitions into consideration since if a state is reachable by a sequence of ε-transitions where some ε-transitions are repeated, then there exists also a sequence of non-repeated ε-transitions reaching the state. Moreover, when simulating an a-transition of $\mathcal{T}$ (where $a \in \Sigma$) at a state (q, ρ, Λ, S), suppose $\delta_T(q, a) = (q_1, \cdots, q_m)$ and $\tau_T(q) = (P_1, P_2)$, then $\mathcal{B}$ nondeterministically chooses q_i and goes to the state $(q_i, \rho', \emptyset, S')$, where

- ρ' is obtained from ρ and $E_T(q, \sigma, q_i)$,
- Λ is reset to $\emptyset$,
- all the states obtained from S by applying an a transition should be *saturated by ε-transitions* and put into S', more precisely, all the states reachable from S by first applying an a-transition, then a sequence of ε-transitions, should be put into S',
- moreover, all the states obtained from $q_1, \cdots, q_{i-1}$ (which are of higher priorities than q_i) by saturating with ε-transitions should be put into S',

- finally, all the states obtained from those in $P_1' = \{q' \in P_1 \mid (q, q') \not\in \Lambda\}$ (which are of higher priorities than q_i) by saturating with non-Λ ε-transitions first (i.e. the ε-transitions that do not belong to Λ), and applying an a-transition next, finally saturating with ε-transitions again, should be put into S', (note that according to the semantics of PSST, the ε-transitions in Λ should be avoided when defining P_1' and saturating the states in P_1' with ε-transitions).

The above construction does not utilize the so-called *copyless* property (i.e. for each transition t and each variable x, x appears at most once on the right-hand side of the assignment for t) [5, 6], thus it works for general, or *copyful*, PSSTs [29]. It can be noted that the powerset in $\rho \in (\mathcal{P}(Q_A \times Q_A))^X$ is required to handle copyful transductions as the contents of a variable may be used in many different situations, each requiring a different abstraction. If the PSST is copyless, we can instead use $\rho \in (Q_A \times Q_A)^X$. That is, each variable is used only once, and hence only one abstraction pair is needed. The powerset construction in the transitions can be replaced by a non-deterministic choice of the particular pair of states from Q_A that should be kept. This avoids the construction being exponential in the size of A, which in turn avoids the tower of exponential blow-up in the backwards reasoning.

For the formal construction of $\mathcal{B}$, we need some additional notations.

- For $S \subseteq Q_T$, $\delta_T^{(ip)}(S, a) = \{q_1' \mid \exists q_1 \in S, q_1' \in \delta_T(q_1, a)\}$.
- For $q \in Q_T$, if $\tau_T(q) = (P_1, P_2)$, then $\tau_T^{(ip)}(\{q\}) = S$ such that $S = P_1 \cup P_2$. Moreover, for $S \subseteq Q_T$, we define $\tau_T^{(ip)}(S) = \bigcup\limits_{q \in S} \tau_T^{(ip)}(\{q\})$. We also use $\left(\tau_T^{(ip)}\right)^*$ to denote the ε-closure of $\mathcal{T}$, namely, $\left(\tau_T^{(ip)}\right)^*(S) = \bigcup\limits_{n \in \mathbb{N}} \left(\tau_T^{(ip)}\right)^n(S)$, where $\left(\tau_T^{(ip)}\right)^0(S) = S$, and for $n \in \mathbb{N}$, $\left(\tau_T^{(ip)}\right)^{n+1}(S) = \tau_T^{(ip)}\left(\left(\tau_T^{(ip)}\right)^n(S)\right)$.
- For $S \subseteq Q_T$ and $\Lambda \subseteq \mathcal{E}(\tau_T)$, we use $\left(\tau_T^{(ip)} \backslash \Lambda\right)^*(S)$ to denote the set of states reachable from S by sequences of ε-transitions where *no* transitions (q, ε, q') such that $(q, q') \in \Lambda$ are used.
- For $\rho \in (\mathcal{P}(Q_A \times Q_A))^X$ and $s \in X \to (X \cup \Sigma)^*$, we use $s(\rho)$ to denote ρ' that is obtained from ρ as follows: For each $x \in X$, if $s(x) = \varepsilon$, then $\rho'(x) = \{(p, p) \mid p \in Q_A\}$, otherwise, let $s(x) = b_1 \cdots b_\ell$ with $b_i \in \Sigma \cup X$ for each $i \in [\ell]$, then $\rho'(x) = \theta_1 \circ \cdots \circ \theta_\ell$, where $\theta_i = \delta_A^{(b_i)}$ if $b_i \in \Sigma$, and $\theta_i = \rho(b_i)$ otherwise, and $\circ$ represents the composition of binary relations.

We are ready to present the formal construction of $\mathcal{B} = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$.

- $Q_B = Q_T \times (\mathcal{P}(Q_A \times Q_A))^X \times \mathcal{P}(\mathcal{E}(\tau_T)) \times \mathcal{P}(Q_T)$,
- $q_{0,B} = (q_{0,T}, \rho_\varepsilon, \emptyset, \emptyset)$ where $\rho_\varepsilon(x) = \{(q, q) \mid q \in Q\}$ for each $x \in X$,
- δ_B comprises
  - the tuples $((q, \rho, \Lambda, S), a, (q_i, \rho', \Lambda', S'))$ such that
    - $*$ $a \in \Sigma$,
    - $*$ $\delta_T(q, a) = (q_1, \ldots, q_i, \ldots, q_m)$,
    - $*$ $s = E((q, a, q_i))$,
    - $*$ $\rho' = s(\rho)$,

    - $\Lambda' = \emptyset$, (Intuitively, Λ is reset.)
    - let $\tau_T(q) = (P_1, P_2)$, then $S' = \big(\tau_T^{(ip)}\big)^* \big(\{q_1, \ \ldots, \ q_{i-1}\} \cup \delta_T^{(ip)}\big(S \cup \big(\tau_T^{(ip)} \setminus \Lambda\big)^*(P'_1), a\big)\big)$, where $P'_1 = \{q' \in P_1 \mid (q, q') \not\in \Lambda\}$;
  - the tuples $((q, \rho, \Lambda, S), \varepsilon, (q_i, \rho', \Lambda', S'))$ such that
    - $\tau_T(q) = ((q_1, \ldots, q_i, \ldots, q_m); \cdots)$,
    - $(q, q_i) \not\in \Lambda$,
    - $s = E(q, \varepsilon, q_i)$,
    - $\rho' = s(\rho)$,
    - $\Lambda' = \Lambda \cup \{(q, q_i)\}$,
    - $S' = S \cup \big(\tau_T^{(ip)} \setminus \Lambda\big)^*(\{q_j \mid j \in [i-1], (q, q_j) \not\in \Lambda\})$;
  - the tuples $((q, \rho, \Lambda, S), \varepsilon, (q_i, \rho', \Lambda', S'))$ such that
    - $\tau_T(q) = ((q'_1, \ldots, q'_n); (q_1, \ldots, q_i, \ldots, q_m))$,
    - $(q, q_i) \not\in \Lambda$,
    - $s = E(q, \varepsilon, q_i)$,
    - $\rho' = s(\rho)$,
    - $\Lambda' = \Lambda \cup \{(q, q_i)\}$,
    - $S' = S \cup \{q\} \cup \big(\tau_T^{(ip)} \setminus \Lambda\big)^* \big(\{q'_j \mid j \in [n], (q, q'_j) \not\in \Lambda\} \cup \{q_j \mid j \in [i-1], (q, q_j) \not\in \Lambda\}\big)$. (Note that here we include q into S', since the non-ε-transitions out of q have higher priorities than the transition (q, ε, q_i).)

- Moreover, F_B is the set of states $(q, \rho, \Lambda, S) \in Q_B$ such that

  1. $F_T(q)$ is defined,
  2. for every $q' \in S$, $F_T(q')$ is not defined,
  3. if $F_T(q) = \varepsilon$, then $q_{0,A} \in F_A$, otherwise, let $F_T(q) = b_1 \cdots b_\ell$ with $b_i \in \Sigma \cup X$ for each $i \in [\ell]$, then $(\theta_1 \circ \cdots \circ \theta_\ell) \cap (\{q_{0,A}\} \times F_A) \neq \emptyset$, where for each $i \in [\ell]$, if $b_i \in \Sigma$, then $\theta_i = \delta_A^{(b_i)}$, otherwise, $\theta_i = \rho(b_i)$.

A.4 Tower-Hardness of String Constraints with Streaming String Transductions

We show that the satisfiability problem for $\mathsf{STR}_{\mathsf{SL}}$ is Tower-hard.

Theorem A.2. *The satisfiability problem for* $\mathsf{STR}_{\mathsf{SL}}$ *is Tower-hard.*

Our proof will use tiling problems over extremely wide corridors. We first introduce these tiling problems, then how we will encode potential solutions as words. Finally, we will show how $\mathsf{STR}_{\mathsf{SL}}$ can verify solutions.

A.4.1 Tiling Problems

A *tiling problem* is a tuple $\big(\Theta, H, V, t^0, f\big)$ where Θ is a finite set of tiles, $H \subseteq \Theta \times \Theta$ is a horizontal matching relation, $V \subseteq \Theta \times \Theta$ is a vertical matching relation, and $t^0, f \in \Theta$ are initial and final tiles respectively.

A solution to a tiling problem over a n-width corridor is a sequence

$$\begin{array}{c} t_1^1 \dots t_n^1 \\ t_1^2 \dots t_n^2 \\ \dots \\ t_1^h \dots t_n^h \end{array}$$

where $t_1^1 = t^0$, $t_n^h = f$, and for all $1 \leq i < n$ and $1 \leq j \leq h$ we have $\left(t_i^j, t_{i+1}^j\right) \in H$ and for all $1 \leq i \leq n$ and $1 \leq j < h$ we have $\left(t_i^j, t_i^{j+1}\right) \in V$. Note, we will assume that t^0 and f can only appear at the beginning and end of the tiling respectively.

Tiling problems characterise many complexity classes [15]. In particular, we will use the following facts.

- For any n-space Turing machine, there exists a tiling problem of size polynomial in the size of the Turing machine, over a corridor of width n, that has a solution iff the n-space Turing machine has a terminating computation.
- There is a fixed $\left(\Theta, H, V, t^0, f\right)$ such that for any width n there is a unique solution

  $$\begin{array}{c} t_1^1 \dots t_n^1 \\ t_1^2 \dots t_n^2 \\ \dots \\ t_1^h \dots t_n^h \end{array}$$

  and moreover h is exponential in n. One such example is a Turing machine where the tape contents represent a binary number. The Turing machine starts from a tape containing only 0s and finishes with a tape containing only 1s by repeatedly incrementing the binary encoding on the tape. This Turing machine can be encoded as the required tiling problem.

A.4.2 Large Numbers

The crux of the proof is encoding large numbers that can take values between 1 and m-fold exponential.

A linear-length binary number could be encoded simply as a sequence of bits

$$b_0 \dots b_n \in \{0, 1\}^n \ .$$

To aid with later constructions we will take a more oblique approach. Let $\left(\Theta_1, H_1, V_1, t_1^0, f_1\right)$ be a copy of the fixed tiling problem from the previous section for which there is a unique solution, whose length must be exponential in the width. In the future, we will need several copies of this problem, hence the indexing here. Note, we assume each copy has disjoint tile sets. Fix a width n and let N_1 be the corresponding corridor length. A *level-1* number can encode values from 1 to N_1. In particular, for $1 \leq i \leq N_1$ we define

$$[i]_1 = t_1^i \dots t_n^i$$

where $t_1^i \dots t_n^i$ is the tiling of the ith row of the unique solution to the tiling problem.

A *level-2* number will be derived from tiling a corridor of width N_1, and thus the number of rows will be doubly-exponential. For this, we require another copy $\left(\Theta_2, H_2, V_2, t_2^0, f_2\right)$ of the above tiling problem. Moreover, let N_2 be the length of the solution for a corridor of width N_1. Then for any $1 \leq i \leq N_2$ we define

$$[i]_2 = [1]_1 t_1^i [2]_1 t_2^i \ldots [N_1]_1 t_{N_1}^i$$

where $t_1^i \ldots t_{N_1}^i$ is the tiling of the ith row of the unique solution to the tiling problem. That is, the encoding indexes each tile with it's column number, where the column number is represented as a level-1 number.

In general, a *level-m* number is of length $(m-1)$-fold exponential and can encode numbers m-fold exponential in size. We use a copy $\left(\Theta_m, H_m, V_m, t_m^0, f_m\right)$ of the above tiling problem and use a corridor of width N_{m-1}. We define N_m as the length of the unique solution to this problem. Then, for any $1 \leq i \leq N_m$ we have

$$[i]_m = [1]_{m-1} t_1^i [2]_{m-1} t_2^i \ldots [N_{m-1}]_{m-1} t_{N_{m-1}}^i$$

where $t_1^i \ldots t_{N_{m-1}}^i$ is the tiling of the ith row of the unique solution to the tiling problem.

Note that we can define regular languages to check that a string is a large number. In particular

$$R_m^n = \begin{cases} [\Theta_1]^n & m = 1 \\ \left[R_{m-1}^n \Theta_m\right]^* & m > 1 \ . \end{cases}$$

A.4.3 Hardness Proof

We show that the satisfiability problem for $\mathsf{STR}_{\mathsf{SL}}$ is Tower-hard. We first introduce the basic framework of solving a hard tiling problem. Then we discuss the two phases of transductions required by the reduction. These are constructing a large boolean formula, and then evaluating the formula. This two phases are described in separate sections.

The Framework The proof is by reduction from a tiling problem over an m-fold exponential width corridor. In general, solving such problems is hard for m-ExpSpace.

Let N_m be the width of the corridor. Fix a tiling problem

$$\left(\Theta, H, V, t^0, f\right) \ .$$

We will compose an $\mathsf{STR}_{\mathsf{SL}}$ formula S with a free variable x. If S is satisfiable, x will contain a string encoding a solution to the tiling problem. In particular, the value of x will be of the form

$$\begin{gathered} [1]_m t_1^1 \ldots [N_m]_m t_{N_m}^1 \# \\ [1]_m t_1^2 \ldots [N_m]_m t_{N_m}^2 \# \\ \ldots \\ [1]_m t_1^h \ldots [N_m]_m t_{N_m}^h \# \ . \end{gathered}$$

That is, each row of the solution is separated by the # symbol. Between each tile of a row is it's index, encoded using the large number encoding described in the previous section.

The formula S will use a series of replacements and assertions to verify that the tiling encoded by x is a valid solution to the tiling problem. We will give the formula in three steps.

We will define the alphabet to be

$$\Sigma = \Theta \cup \overline{\Theta}$$

where Θ is the set of tiles, and $\overline{\Theta}$ is the set of characters required to encode large numbers, plus #.

The first part is

$$
\begin{array}{l}
\mathsf{assert}\left(x \in \left[[R_m^n \Theta]^* \,\#\right]^*\right) ; \\
\mathsf{assert}\left(x \in R_m^n t^0\right) ; \\
\mathsf{assert}\left(x \in \Sigma^* f \#\right) ; \\
\mathsf{assert}\left(x \in \left[\left[\sum\limits_{(t_1,t_2)\in H} R_m^n t_1 R_m^n t_2\right]^* [R_m^n \Theta]^? \,\#\right]^*\right) ; \\
\mathsf{assert}\left(x \in \left[[R_m^n \Theta]\left[\sum\limits_{(t_1,t_2)\in H} R_m^n t_1 R_m^n t_2\right]^* [R_m^n \Theta]^? \,\#\right]^*\right) ;
\end{array}
$$

The first asserts simply verify the format of the value of x is as expected and moreover, the first appearing element of Θ in the string is t^0, and the last element is f.

The final two assertions check the horizontal tiling relation. In particular, the first checks that even pairs of tiles are in H, while the second checks odd pairs are in H.

The main challenge is checking the vertical tiling relation. This is done by a series of transductions operating in two main phases. The first phase rewrites the encoding into a kind of large Boolean formula, which is then evaluated in the second phase.

Constructing the Large Boolean Formula The next phase of the formula is shown below and explained afterwards. For convenience, we will describe the construction using transductions. After the explanation, we will describe how to achieve these transductions using $\mathsf{replaceAll}$.

$$
\begin{array}{l}
x_m^1 = \mathcal{T}_m^1(x); \\
x_m^2 = \mathcal{T}_m^2\left(x_m^1\right); \\
x_m^3 = \mathcal{T}_m^3\left(x_m^2\right); \\
x_{m-1}^1 = \mathcal{T}_{m-1}^1\left(x_m^3\right); \\
x_{m-1}^2 = \mathcal{T}_{m-1}^2\left(x_{m-1}^1\right); \\
x_{m-1}^3 = \mathcal{T}_{m-1}^3\left(x_{m-1}^2\right); \\
\ldots \\
x_1^1 = \mathcal{T}_1^1\left(x_2^3\right); \\
x_1^2 = \mathcal{T}_1^2\left(x_1^1\right); \\
x_1^3 = \mathcal{T}_1^3\left(x_1^2\right); \\
x_0 = \mathcal{T}_0\left(x_1^3\right) \ .
\end{array}
$$

The Boolean formula is constructed by rewriting the encoding stored in x. We need to check the vertical tiling relation by comparing t_j^i with t_j^{i+1}. However, these are separated by a huge number of other tiles, which also need to be checked against their counterpart in the next row.

The goal of the transductions is to "rotate" the encoding so that instead of each tile being directly next to its horizontal counterpart, it is directly next to its vertical counterpart. Our transductions do not quite achieve this goal, but instead place the tiles in each row next to potential vertical counterparts. The Boolean formula contains large disjunctions over these possibilities and use the indexing by large numbers to pick out the correct pairs.

The idea is best illustrated by showing the first three transductions, $\mathcal{T}_m^1$, $\mathcal{T}_m^2$, and $\mathcal{T}_m^3$. We start with

$$
\begin{array}{c}
[1]_m t_1^1 \ldots [N_m]_m t_{N_m}^1 \# \\
[1]_m t_1^2 \ldots [N_m]_m t_{N_m}^2 \# \\
\ldots \\
[1]_m t_1^h \ldots [N_m]_m t_{N_m}^h \# \ .
\end{array}
$$

The transducer $\mathcal{T}_m^1$ saves the row it is currently reading. Then, when reading the next row, it outputs each index and tile of the current row followed by a copy of the last row. The output is shown below. We use a disjunction symbol to indicate that, after the transduction, the tile should match one of the tiles copied after it. Between each pair of a tile and a copied row, we use the conjunction symbol to indicate that every disjunction should have one match. The result is shown below. To aid readability, we underline the copied rows. The parentheses $\langle\rangle$ are also inserted to aid future parsing.

$$\begin{gathered}\left\langle [1]_m t_1^2 \vee \underline{[1]_m t_1^1 \dots [N_m]_m t_{N_m}^1} \right\rangle \wedge \dots \wedge \left\langle [N_m]_m t_{N_m}^2 \vee \underline{[1]_m t_1^1 \dots [N_m]_m t_{N_m}^1} \right\rangle \\ \wedge \dots \wedge \\ \left\langle [1]_m t_1^h \vee \underline{[1]_m t_1^{h-1} \dots [N_m]_m t_{N_m}^{h-1}} \right\rangle \wedge \dots \wedge \left\langle [N_m]_m t_{N_m}^h \vee \underline{[1]_m t_1^{h-1} \dots [N_m]_m t_{N_m}^{h-1}} \right\rangle \ .\end{gathered}$$

After this transduction, we apply $\mathcal{T}_m^2$. This transduction forms pairs of a tile, with all tiles following it from the previous row (up to the next $\wedge$ symbol). This leaves us with a conjunction of disjunctions of pairs. Inside each disjunct, we need to verify that one pair has matching indices and tiles that satisfy the vertical tiling relation V. The result of the second transduction is shown below.

$$\begin{gathered}\langle [1]_m t_1^2 [1]_m t_1^1 \vee \dots \vee [1]_m t_1^2 [N_m]_m t_{N_m}^1 \rangle \wedge \dots \wedge \langle [N_m]_m t_{N_m}^2 [1]_m t_1^1 \vee \dots \vee [N_m]_m t_{N_m}^2 [N_m]_m t_{N_m}^1 \rangle \\ \wedge \dots \wedge \\ \langle [1]_m t_1^h [1]_m t_1^{h-1} \vee \dots \vee [1]_m t_1^h [N_m]_m t_{N_m}^{h-1} \rangle \wedge \dots \wedge \langle [N_m]_m t_{N_m}^h [1]_m t_1^{h-1} \vee \dots \vee [N_m]_m t_{N_m}^h [N_m]_m t_{N_m}^{h-1} \rangle \ .\end{gathered}$$

Notice that we now have each tile in a pair with its vertical neighbour, but also in a pair with every other tile in the row beneath. The indices can be used to pick out the right pairs, but we will need further transductions to analyse the encoding of large numbers.

To simplify matters, we apply $\mathcal{T}_m^3$. This transduction removes the tiles from the string, retaining each pair of indices where the tiles satisfy the vertical tiling relation. When the tiling relation is not satisfied, we insert $\perp_m$. We use $\langle$, #, and $\rangle$ to delimit the indices. We are left with a string of the form

$$\bigwedge \bigvee \langle [i]_m \# [j]_m \rangle \vee \perp_m \vee \dots \vee \perp_m \ .$$

We will often elide the $\perp_m$ disjuncts for clarity. They will remain untouched until the formula is evaluated in the next section.

We consider a pair $\langle [i]_m \# [j]_m \rangle$ to evaluate to true whenever $i = j$. The truth of the formula can be computed accordingly. However, it's not straightforward to check whether $i = j$ as they are large numbers. The key observation is that they are encoded as solutions to indexed tiling problems, which means we can go through a similar process to the transductions above.

First, recall that $[i]_m$ is of the form

$$[1]_{m-1} d_1^i [2]_{m-1} d_2^i \dots [N_{m-1}]_{m-1} d_{N_{m-1}}^i$$

where we use d to indicate tiles instead of t.

We apply three transductions $\mathcal{T}_{m-1}^1$, $\mathcal{T}_{m-1}^2$, and $\mathcal{T}_{m-1}^3$. The first copies the first index of each pair directly after the tiles of the second index. That is, each pair

$$\langle [i]_m \# [j]_m \rangle$$

is rewritten to

$$\left\langle [1]_{m-1} d_1^j \vee [i]_m \right\rangle \wedge \dots \wedge \left\langle [N_{m-1}]_{m-1} d_{N_{m-1}}^j \vee [i]_m \right\rangle \ .$$

Then we apply a similar second transduction: each disjunction is expanded into pairs of indices and tiles. The result is

$$\begin{gathered}\left\langle [1]_{m-1} d_1^j [1]_{m-1} d_1^i \vee \ldots \vee [1]_{m-1} d_1^j [N_{m-1}]_{m-1} d^i_{N_{m-1}} \right\rangle \\ \wedge \ldots \wedge \\ \left\langle [N_{m-1}]_{m-1} d^j_{N_{m-1}} [1]_{m-1} d_1^i \vee \ldots \vee [N_{m-1}]_{m-1} d^j_{N_{m-1}} [N_{m-1}]_{m-1} d^i_{N_{m-1}} \right\rangle \ .\end{gathered}$$

The third transduction replaces with $\bot_{m-1}$ all pairs where we don't have $d^j_k = d^i_{k'}$ (recall, we need to check that $i = j$ so the tiles at each position should be the same). As before, for a single pair, this leaves us with a string formula of the form

$$\bigwedge \bigvee \langle [i']_{m-1} \# [j']_{m-1} \rangle \vee \bot_{m-1} \vee \cdots \vee \bot_{m-1} \ .$$

Again, we will elide the $\bot_{m-1}$ disjuncts for clarity as they will be untouched until the formula is evaluated. Recalling that there are many pairs in the input string, the output of this series of transductions is a string formula of the form

$$\bigwedge \bigvee \bigwedge \bigvee \langle [i']_{m-1} \# [j']_{m-1} \rangle \ .$$

We repeat these steps using $\mathcal{T}^1_{m-2}$, $\mathcal{T}^2_{m-2}$, $\mathcal{T}^3_{m-2}$ all the way down to $\mathcal{T}^1_1$, $\mathcal{T}^2_1$, $\mathcal{T}^3_1$. We are left with a string formula of the form

$$\bigwedge \bigvee \cdots \bigwedge \bigvee \langle [i']_1 \# [j']_1 \rangle \ .$$

Recall each $[i']_1$ is of the form

$$d_1^{i'} \ldots d_n^{i'} \ .$$

The final step interleaves the tiles of the two numbers. The result is a string formula of the form

$$\bigwedge \bigvee \cdots \bigwedge \bigvee \bigwedge dd' \ .$$

This is the formula that is evaluated in the next phase.

To complete this section we need to implement the above transductions using replaceAll.

First, consider $\mathcal{T}^1_m$. We start with

$$\begin{gathered}[1]_m t_1^1 \ldots [N_m]_m t^1_{N_m} \# \\ [1]_m t_1^2 \ldots [N_m]_m t^2_{N_m} \# \\ \ldots \\ [1]_m t_1^h \ldots [N_m]_m t^h_{N_m} \# \ .\end{gathered}$$

We are aiming for

$$\begin{gathered}\left\langle [1]_m t_1^2 \vee [1]_m t_1^1 \ldots [N_m]_m t^1_{N_m} \right\rangle \wedge \ldots \wedge \left\langle [N_m]_m t^2_{N_m} \vee [1]_m t_1^1 \ldots [N_m]_m t^1_{N_m} \right\rangle \\ \wedge \ldots \wedge \\ \left\langle [1]_m t_1^h \vee [1]_m t_1^{h-1} \ldots [N_m]_m t^{h-1}_{N_m} \right\rangle \wedge \ldots \wedge \left\langle [N_m]_m t^h_{N_m} \vee [1]_m t_1^{h-1} \ldots [N_m]_m t^{h-1}_{N_m} \right\rangle \ .\end{gathered}$$

We use two replaceAlls. The first uses $\$^{\leftarrow}$ to do the main work of copying the previous row into the current row a huge number of times. In fact, $\$^{\leftarrow}$ will copy too much, as it will copy everything that came before, not just the last row. The second replaceAll will cut down the contents of $\$^{\leftarrow}$ to only the last row. That is, we first apply $\mathsf{replaceAll}_{\mathsf{pat}_1,\mathsf{rep}_1}$ and then $\mathsf{replaceAll}_{\mathsf{pat}_2,\mathsf{rep}_2}$ where

$$\begin{array}{rcl} \mathsf{pat}_1 & = & (t) \\ \mathsf{rep}_1 & = & \$1 \lhd \$^{\leftarrow} \rhd \end{array}$$

and $\lhd$ and $\rhd$ are two characters not in Σ, and, letting $\Sigma_{\#} = \Sigma \setminus \{\#\}$,

$$\begin{array}{rcl} \mathsf{pat}_2 & = & \lhd \Sigma_{\#}^{*} \# (\Sigma_{\#}^{*}) \# \Sigma_{\#}^{*} \rhd \\ \mathsf{rep}_2 & = & \vee \$1 \ . \end{array}$$

That is, the first replace adds after each tile the entire preceding string, delimited by $\lhd$ and $\rhd$. The second replace picks out the final row of each string between $\lhd$ and $\rhd$ and adds the $\vee$. Notice that the second replace does not match anything between $\lhd$ and $\rhd$ on the first row. In fact, we need another $\mathsf{replaceAll}$ to delete the first row. That is $\mathsf{replaceAll}_{\mathsf{pat}_3,\mathsf{rep}_3}$ where

$$\begin{array}{rcl} \mathsf{pat}_3 & = & [\Sigma \cup \{\lhd, \rhd\}]^{*} \rhd \Sigma_{\#}^{*} \# \\ \mathsf{rep}_3 & = & \varepsilon \ . \end{array}$$

Notice, the pattern above matches any row containing at least one $\rhd$. This means only the first row will be deleted as delimiters have already been removed from the other rows. To complete the step, we replace all $\#$ with $\wedge$ and insert the parenthesis $\langle\rangle$ using another $\mathsf{replaceAll}$ (and a concatenation at the beginning and the end of the string).

The transduction $\mathcal{T}_m^2$ uses similar techniques to the above and we leave the details to the reader. The same is true of the other similar transductions $\mathcal{T}_i^1$ and $\mathcal{T}_i^2$.

Transduction $\mathcal{T}_m^3$ (and similarly the other $\mathcal{T}_i^2$) replaces all pairs

$$[i]_m t_i^2 [i+1]_m t_{i+1}^1$$

that do not satisfy the vertical tiling relation with $\perp_m$, and rewrites them to

$$\langle [i']_1 \# [j']_1 \rangle$$

if the vertical tiling relation is matched. This can be done in two steps: first replace the non-matches, then replace the matches. To replace the non-matches we use $\mathsf{replaceAll}_{\mathsf{pat}_1,\mathsf{rep}_1}$ where

$$\begin{array}{rcl} \mathsf{pat}_1 & = & \sum\limits_{(t_1,t_2) \notin V} R_m^n t_1 R_m^n t_2 \\ \mathsf{rep}_1 & = & \perp_m \ . \end{array}$$

For the matches we use $\mathsf{replaceAll}_{\mathsf{pat}_2,\mathsf{rep}_2}$ where

$$\begin{array}{rcl} \mathsf{pat}_2 & = & \sum\limits_{(t_1,t_2) \in V} (R_m^n) t_1 (R_m^n) t_2 \\ \mathsf{rep}_1 & = & \langle \$1 \# \$2 \rangle \ . \end{array}$$

The final transduction takes a string of the form

$$\bigwedge \bigvee \cdots \bigwedge \bigvee \langle [i']_1 \# [j']_1 \rangle$$

where each $[i']_1$ is of the form

$$d_1^{i'} \ldots d_n^{i'} \ .$$

We need to interleave the tiles of the two numbers, giving a string of the form

$$\bigwedge \bigvee \cdots \bigwedge \bigvee \bigwedge dd' \ .$$

This can be done with a single $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}$ where

$$\begin{array}{rcl} \mathsf{pat} & = & \langle (\Theta_1) \ldots (\Theta_1) \# (\Theta_1) \ldots (\Theta_1) \rangle \\ \mathsf{rep} & = & \langle \$1\$(n+1) \wedge \cdots \wedge \$n\$(2n) \rangle \ . \end{array}$$

Evaluating the Large Boolean Formula The final phase of S evaluates the Boolean formula and is shown below. Again we write the formula using transductions and explain how they can be done with $\mathsf{replaceAll}$.

$$\begin{array}{c} x_0^\wedge = \mathcal{T}_0(x_0); \\ x_1^\vee = \mathcal{T}_0^\wedge(x_0^\wedge); \\ x_1^\wedge = \mathcal{T}_1^\vee(x_1^\vee); \\ x_2^\vee = \mathcal{T}_1^\wedge(x_1^\wedge); \\ x_2^\wedge = \mathcal{T}_2^\vee(x_2^\vee); \\ x_3^\wedge = \mathcal{T}_2^\wedge(x_2^\wedge); \\ \cdots \\ x_m^\vee = \mathcal{T}_{m-1}^\wedge(x_{m-1}^\wedge); \\ x_m^\wedge = \mathcal{T}_m^\vee(x_m^\vee); \\ x_f = \mathcal{T}_m^\wedge(x_m^\vee); \\ \mathsf{assert}\left(x_f \in \mathsf{pat}_f\right) \end{array}$$

The first transducer $\mathcal{T}_1$ reads the string formula

$$\bigwedge \bigvee \cdots \bigwedge \bigvee \bigwedge dd' \ .$$

copies it to its output, except replacing each pair dd' with $\top_1$ if $d = d'$ and with $\bot_1$ otherwise. This is requires two simple $\mathsf{replaceAll}$ calls.

The remaining transductions evaluate the innermost disjunction or conjunction as appropriate (the parenthesis $\langle\rangle$ are helpful here). For example $\mathcal{T}_1^\vee$ replaces the innermost $\bigvee v$ with $\top_1$ if $\top_1$ appears somewhere in the disjunction and $\bot_1$ otherwise. This can be done by greedily matching any sequence of characters from $\{\top_1, \bot_1, \vee\}$ that contains at least one $\top_1$ and replacing the sequence with $\top_1$, then greedily matching any remaining sequence of $\{\bot_1, \vee\}$ and replacing it with $\bot_1$. The evaluation of conjunctions works similarly, but inserts $\top_2$ and $\bot_2$ in the move to the next level of evaluation.

The final assert checks that x_f contains only the character $\top_{m+1}$ and fails otherwise.

This completes the reduction.

A.5 Exponential Space Copyless Algorithm

We argue that, when the PSSTs are copyless, satisfiability for $\mathsf{STR}_{\mathsf{SL}}$ can be decided in exponential space.

The algorithm in the proof of Theorem 5.7 for $\mathsf{STR}_{\mathsf{SL}}$ applies a subset of the proof rules in Table 1. These proof rules branch on disjunctions and backward propagation through transductions and concatenations. We can explore all branches through the proof tree, storing the sequence of branches chosen in polynomial space. Thus, we can consider each branch independently.

A branch consists of a backwards propagation through PSSTs and concatenations. We will argue that each state of the automata constructed can be stored in exponential space. Since these states are combinations of states of PSSTs and finite automata constructed by earlier stages of the algorithm, it is possible to calculate the next states from the current state on the fly. Thus, if the states can be stored in exponential space, the full algorithm will only require exponential space.

We first consider the case where we have PSSTs and FAs rather than string functions using RegEx. Let n be the size of the largest PSST or FA. Let x be the maximum number of variables in the PSST. Finally, let ℓ be the length of the longest branch of transductions and concatenations in the proof tree (which is linear in the size of the constraint).

The FA in the pre-image of a concatenation all have the same size as the output automaton $\mathcal{A}$. The FA $\mathcal{B}$ in the pre-image of a PSST $\mathcal{T}$ with output automaton $\mathcal{A}$ is an automaton such that $\mathscr{L}(\mathcal{B}) = \mathcal{R}_{\mathcal{T}}^{-1}(\mathscr{L}(\mathcal{A}))$. It has states of the form (q, ρ, Λ, S), where q is a state of $\mathcal{T}$, ρ is a function from variables of $\mathcal{T}$ to pairs of states of $\mathcal{A}$, $\Lambda \subseteq \mathcal{E}(\tau_T)$, and $S \subseteq Q_T$. Note, because we assume $\mathcal{T}$ is copyless, ρ is a function to pairs of states, not to sets of pairs of states. The space needed to store a state of of $\mathcal{B}$ is hence $\mathcal{O}(s + 2xs + 2n)$ where s is the space required to store a state of $\mathcal{A}$. Consequently, after ℓ backwards propagations, we can store the states of the automaton in space $\mathcal{O}(2^{\ell} x^{\ell} n)$. That is, exponential space. This remains true when the PSST and FA may be exponential in the size of the *RegEx*.