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

ANONYMOUS AUTHOR(S)

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 provide such a 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.

Additional Key Words and Phrases: String Constraint Solving, Regular Experssions, Transducers, Symbolic Execution

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 are to be 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 [de Moura and Bjørner 2008], CVC4 [Liang et al. 2014], Z3-str/2/3/4 [Berzish et al. 2017; Berzish, Murphy 2021; Zheng et al. 2015, 2013], ABC [Bultan and contributors 2015], Norn [Abdulla et al. 2014], Trau [Abdulla et al. 2017, 2018; Bui and contributors 2019], OSTRICH [Chen et al. 2019], S2S [Le and He 2018], Qzy [Cox and Leasure 2017], Stranger [Yu et al. 2010], Sloth [Abdulla et al. 2019; Holík et al. 2018], Slog [Wang et al. 2016], Slent [Wang et al.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

2018], Gecode+S [Scott et al. 2017], G-Strings [Amadini et al. 2017], HAMPI [Kiezun et al. 2012], 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 [Chen et al. 2018]; 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 [Loring et al. 2019] 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 [Hopcroft and Ullman 1979]. 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 `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"`.

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

```
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;
}
```

Fig. 1. Normalize a decimal by removing the leading and trailing zeros

solvers (e.g. see [Abdulla et al. 2018; Chen et al. 2019; Holík et al. 2018; Liang et al. 2014; Trinh et al. 2016; Yu et al. 2014]), 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 [Loring et al. 2019].

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 [Loring et al. 2019], where the path constraints are mapped to constraints in the theory of strings with concatenation and regular constraints in Z3 [de Moura and Bjørner 2008]. 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 [Loring et al. 2019], 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 [Loring et al. 2019]) 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 real-world regular expressions (involving more than 100,000 benchmarks) suggest that our solver substantially outperforms the existing method (i.e. [Loring et al. 2019]); by 30–50-fold in these benchmarks. 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

[1]JavaScript was chosen because it is relevant to string solving [Hooimeijer et al. 2011; Saxena et al. 2010], due to vulnerabilities in JavaScripts caused by string manipulations. Our method can be easily adapted to RegEx semantics in other languages.

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 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 extract). We also support replace (resp. 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 [Loring et al. 2019] across 415,487 NPM packages.

A crucial step in the development of our string solver is a formalization of the semantics of the extract, replace, and 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 Transducer (PSST)*, which is inspired by two automata/transducer models: prioritized finite-state automata [Berglund and van der Merwe 2017a] and streaming string transducers [Alur and Cerný 2010; Alur and Deshmukh 2011]. 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 extract, replace, and 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 a sound sequent calculus, our string solver (implemented in the standard DPLL(T) setting of SMT solvers [Nieuwenhuis et al. 2006]) will exploit crucial closure and algorithmic properties satisfied by PSST. 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 $\cdot$, replace, replaceAll, and 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 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 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 [Lin and Barceló 2016]), 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 in a new solver EMU on top of the existing open-source solver OSTRICH [Chen et al. 2019], 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 [Davis et al. 2019], 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 file. We run EMU on these SMT files. EMU is able to answer all four (resp. three) queries in 97.9% (resp. 97.6%) of the match (resp. replace) SMT files, with the average time 1.19 (resp. 1.48) seconds per file. Running ExpoSE [Loring et al. 2019] on the same benchmarks[2] with a fixed time budget (otherwise, it won't terminate), we show that EMU offers a 30x−50x speedup in comparison to ExpoSE, making EMU 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

```
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*|.))*)([A-Z](?:\w*|.))/g;
    return authorList.replace(nameReg, "$2, $1");
  }
  else return authorList;
}
```

Fig. 2. Change the author list from the DBLP format to the ACM format

In this section, we provide one worked out 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.

- 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

[2]More precisely, since ExpoSE is a symbolic execution engine that calls Z3, one can create a small wraparound function, which in effect turns ExpoSE into a string solver.

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

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

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 done by 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 as in Fig. 2.

Our solver is able to show that (1) is unsatisfiable. On the calculus level (introduced in more detail 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).

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

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.

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

where

- $\cdot$ is the string concatenation operation which concatenates two strings,
- e ∈RegEx and pat ∈RegEx,
- for the extract function, $i \in \mathbb{N}$,
- for the replace and replaceAll operation, rep ∈ 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 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-the 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.* 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* 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* extract *as well.*

The function $\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}(x)$ is parameterized by the *pattern* pat $\in$ *RegEx* and the *replacement string* rep $\in$ 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 replace 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 ith capturing group. There are three special references[3] $\$0$, $\$^{\leftarrow}$, 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 $\mathrm{STR}_{\mathrm{SL}}$ denote the set of straight-line STR formulas.

As a crucial step for solving the string constraints in STR, we are going to 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

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

language *recognized* by $\mathcal{A}$, is denoted by $\mathcal{L}(\mathcal{A})$. The *size* $|\mathcal{A}|$ of $\mathcal{A}$ is the cardinality of δ, the set of transitions.

For a finite set Q, let $\overline{Q} = \bigcup_{n\in\mathbb{N}}\{(q_1,\dots,q_n) \mid \forall i \in [n], q_i \in Q \land \forall i,j \in [n], i \neq j \rightarrow q_i \neq q_j\}$. 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 $P = (q_1,\dots,q_n) \in \overline{Q}$ and $q \in Q$, we write $q \in P$ if $q = q_i$ for some $i \in [n]$. Moreover, for $P_1 = (q_1,\dots,q_m) \in \overline{Q}$ and $P_2 = (q'_1,\dots,q'_n) \in \overline{Q}$, we say $P_1 \cap P_2 = \emptyset$ if $\{q_1,\dots,q_m\} \cap \{q'_1,\dots,q'_n\} = \emptyset$.

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

- Q is a finite set of states,
- Σ is the input and output alphabet,
- X is a finite set of string variables,
- $\delta \in Q \times \Sigma \rightarrow \overline{Q}$ defines the non-ε transitions as well as their priorities (from highest to lowest),
- $\tau \in Q \rightarrow \overline{Q} \times \overline{Q}$ 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 $X \rightarrow (X \cup \Sigma)^*$ 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 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 $k, l : i \le k < l < j$, $(q_k, q_{k+1}) \neq (q_l, q_{l+1})$.

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 $\delta(q_{j-1}, a_j) = (\dots, q_j, \dots, q'_j, \dots)$,
  - $q_0 a_1 q_1 \dots q_{j-1} a_j = q_0 a'_1 q'_1 \dots q'_{j-1} a'_j$, $q_j \neq q'_j$, $a_j = \varepsilon$, and one of the following conditions 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 $\mathcal{T}(w) = \perp$, 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 $\{(w, \mathcal{T}(w)) \mid w \in \Sigma^*, \mathcal{T}(w) \neq \perp\}$.

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.

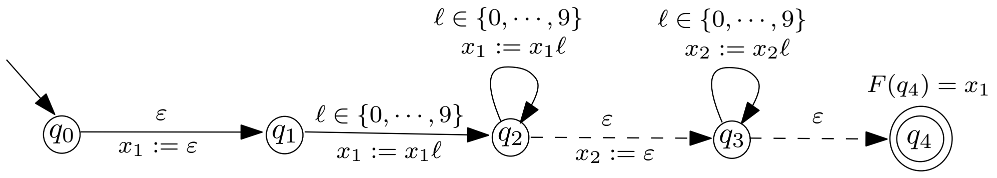


Fig. 4. The PSST $\mathcal{T}$: Extract the matching of the first capturing group in (\d+)(\d*)

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, RegEx is 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 [Berglund et al. 2014; Berglund and van der Merwe 2017a], 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 [Thompson 1968]. 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 [Berglund et al. 2014; Berglund and van der Merwe 2017a] can be easily adapted to construct PSSTs out of regular expressions. Nevertheless, the construction in [Berglund et al. 2014; Berglund and van der Merwe 2017a] 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 [Berglund et al. 2014; Berglund and van der Merwe 2017a], 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.

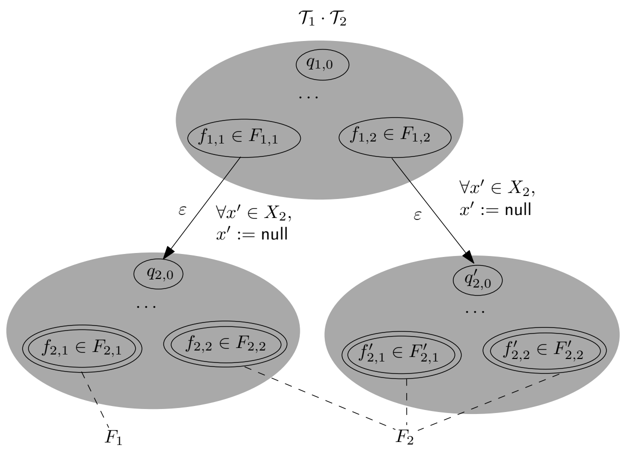


Fig. 5. $\mathcal{T}_1 \cdot \mathcal{T}_2$: Concatenation of $\mathcal{T}_1$ and $\mathcal{T}_2$

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. Note here for technical convenience, we assume that F in a PSST is a set of final states, instead of an output function. Therefore, the PSSTs constructed in the sequel are of the form $(Q, \Sigma, X, \delta, \tau, E, q_0, (F_1, F_2))$. 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 $\mathcal{T}_i = (Q_i, \Sigma, X_i, \delta_i, \tau_i, E_i, q_{i,0}, (F_{i,1}, F_{i,2}))$ be a PSST. 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 Figure 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.

Note that the construction is technical and below we only select to present some representative cases. The other cases are relegated to Appendix A.1.

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 Figure 6). 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

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

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}, and 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 $a \in \Sigma^\varepsilon$).

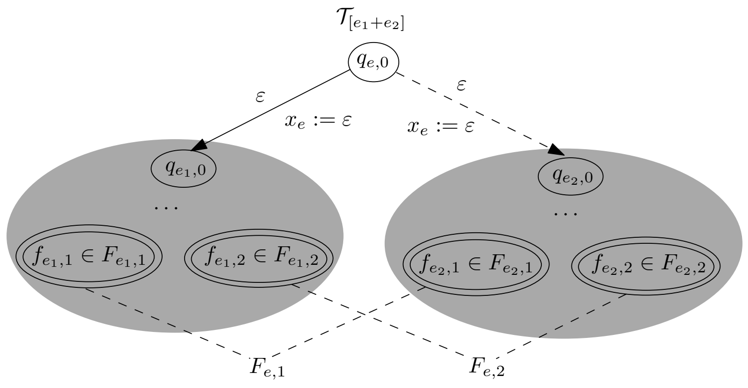


Fig. 6. The PSST $\mathcal{T}_{[e_1+e_2]}$

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 Figure 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 Figure 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 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 $a \in \Sigma^\varepsilon$).

Case $e = [e_1^{??}]$ (see Figure 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 Figure 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},

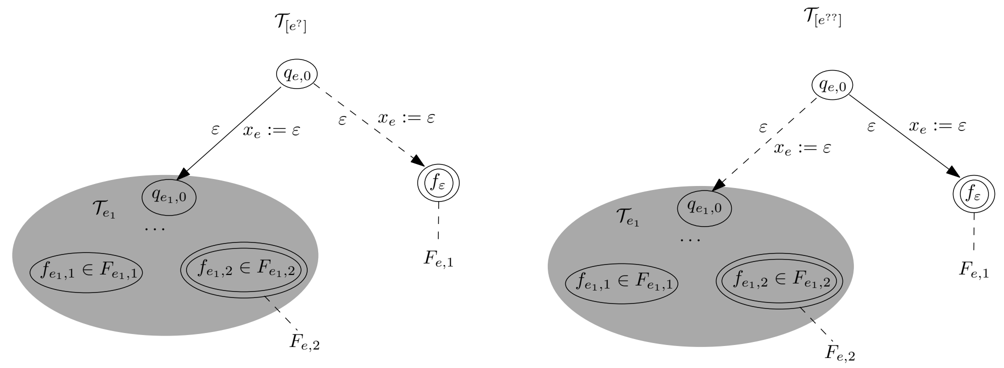

$\mathcal{T}_{[e^?]}$
$q_{e,0}$
ε $x_e := \varepsilon$
ε $x_e := \varepsilon$
$\mathcal{T}_{e_1}$
$q_{e_1,0}$
f_ε
$F_{e,1}$
$f_{e_1,1} \in F_{e_1,1}$
$f_{e_1,2} \in F_{e_1,2}$
$F_{e,2}$
$\mathcal{T}_{[e^{??}]}$
$q_{e,0}$
ε $x_e := \varepsilon$
ε $x_e := \varepsilon$
$\mathcal{T}_{e_1}$
$q_{e_1,0}$
f_ε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

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

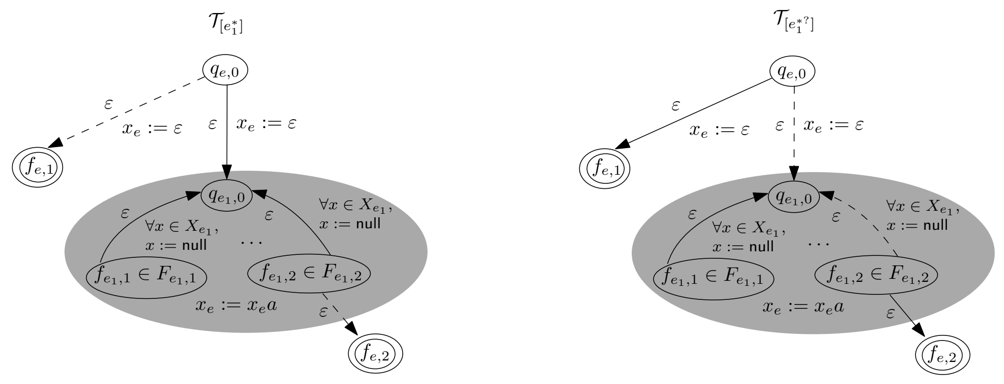

$\mathcal{T}_{[e_1^*]}$
$q_{e,0}$
ε $x_e := \varepsilon$
ε $x_e := \varepsilon$
$f_{e,1}$
$q_{e_1,0}$
ε $\forall x \in X_{e_1}$, $x :=$ null
ε $\forall x \in X_{e_1}$, $x :=$ null
$f_{e_1,1} \in F_{e_1,1}$
$f_{e_1,2} \in F_{e_1,2}$
$x_e := x_e a$
ε
$f_{e,2}$
$\mathcal{T}_{[e_1^{*?}]}$
$q_{e,0}$
ε $x_e := \varepsilon$
ε $x_e := \varepsilon$
$f_{e,1}$
$q_{e_1,0}$
ε $\forall x \in X_{e_1}$, $x :=$ null
ε $\forall x \in X_{e_1}$, $x :=$ null
$f_{e_1,1} \in F_{e_1,1}$
$f_{e_1,2} \in F_{e_1,2}$
$x_e := x_e a$
ε
$f_{e,2}$

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

- 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 $f_{e_1,2} \in F_{e_1,2}$,
- E_e inherits E_{e_1} and the assignments $E_e(q_{e,0}, \varepsilon, f_{e,1})(x_e) = E_e(q_{e,0}, \varepsilon, q_{e_1,0})(x_e) = \varepsilon$, as well as $E_e(f_{e_1,1}, \varepsilon, q_{e_1,0})(x) = E_e(f_{e_1,2}, \varepsilon, q_{e_1,0})(x) = \text{null}$ for every $f_{e_1,1} \in F_{e_1,1}$, $f_{e_1,2} \in F_{e_1,2}$, and $x \in X_{e_1}$. (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 Figure 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 $e = [e_1^{\{m_1,m_2\}}]$ for $1 \leq m_1 < m_2$ (see Figure 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 $q_{e_1,0}^{(1)}$ to a fresh state f_0' (of the lowest priority), and the ε-transitions from each $f_{e_1,2}^{(i)} \in F_{e_1,2}^{(i)}$ with $1 \leq i < m_2 - m_1$ to $q_{e_1,0}^{(i+1)}$ (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 $f_{e_1,1}^{(i)} \in F_{e_1,1}^{(i)}$ 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}.

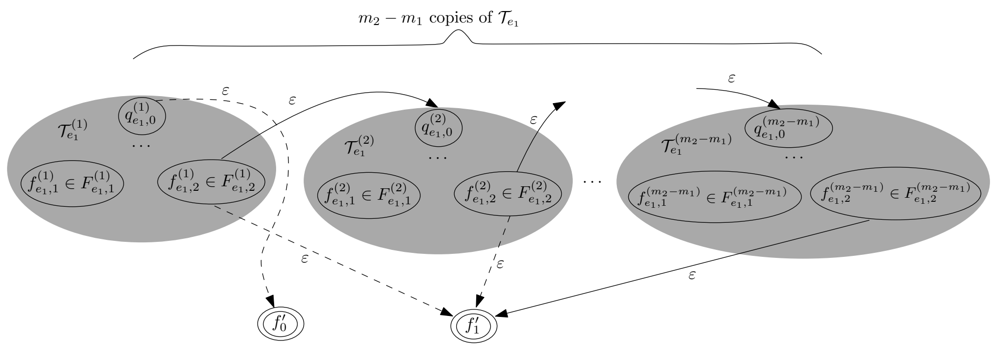

$m_2 - m_1$ copies of $\mathcal{T}_{e_1}$
$\mathcal{T}_{e_1}^{(1)}$
$q_{e_1,0}^{(1)}$
$f_{e_1,1}^{(1)} \in F_{e_1,1}^{(1)}$
$f_{e_1,2}^{(1)} \in F_{e_1,2}^{(1)}$
$\mathcal{T}_{e_1}^{(2)}$
$q_{e_1,0}^{(2)}$
$f_{e_1,1}^{(2)} \in F_{e_1,1}^{(2)}$
$f_{e_1,2}^{(2)} \in F_{e_1,2}^{(2)}$
$\mathcal{T}_{e_1}^{(m_2-m_1)}$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

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

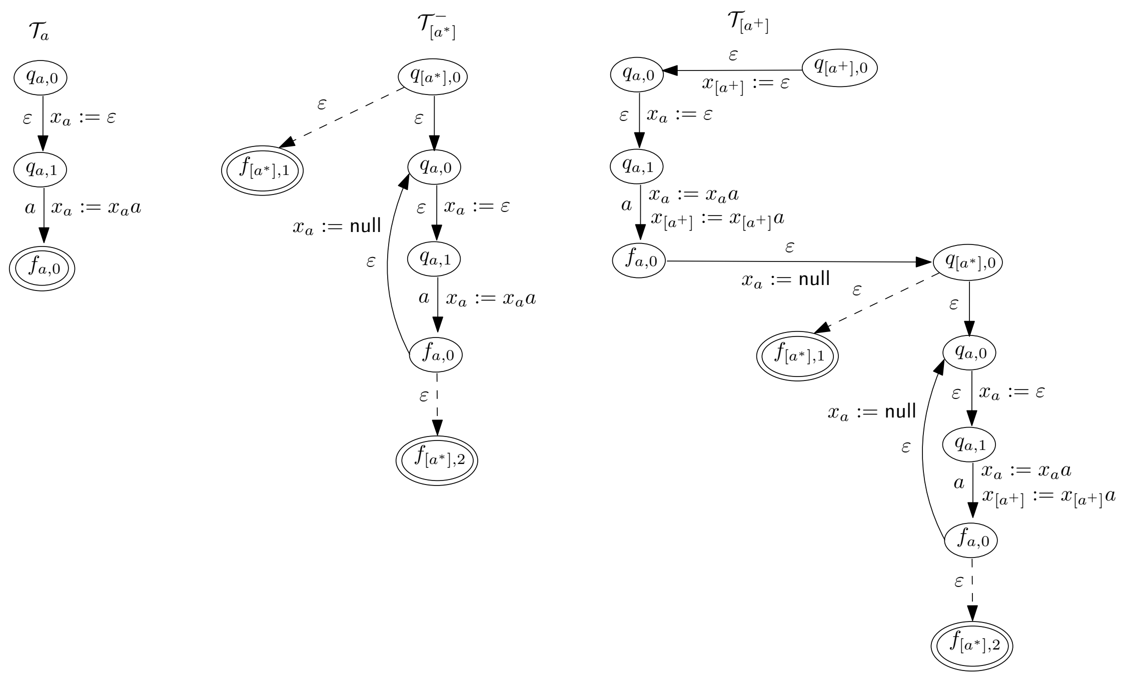


Fig. 10. The PSST $\mathcal{T}_e$ for $e = [a^+]$

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 $\tau_e(q_{e,0}) = ((q_{e_1,0}); ())$ (assuming that $q_{e_1,0}$ is the initial state of $\mathcal{T}_{e_1}^{\{m_1\}}$), and the assignments $E_e(q_{e_0}, \varepsilon, q_{e_1,0})(x_e) = \varepsilon$, as well as $E_e(p, a, q)(x_e) = x_e a$ for each transition (p, a, q) in $\mathcal{T}_{e_1}^{\{m_1\}} \cdot \mathcal{T}_{e_1}^{\{1,m_2-m_1\}}$.

Example 4.5. Let us consider the 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 Figure 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 Figure 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 $\mathcal{T}_a$.

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, namely, 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: $\mathbb{S} = \{a, \ldots, z\}$, $\mathbb{C} = \{A, \ldots, Z\}$, $\mathbb{D} = \{0, \ldots, 9\}$, 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 [^a-zA-Z0-9] (where ^ 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 $10 + 10 * 10 + 10 * 10 * 10 = 1110$ RegExs.

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 extract, replace and 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 4.6. *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 $\text{extract}_{i,e}$ 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 replaceAll here. Full formal details are given in Appendix A.2. The encoding of replace is almost identical to that of replaceAll.

A call $\text{replaceAll}_{\text{pat},\text{rep}}(x)$ 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 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 appendix.

Let $\mathsf{rep} = w_1 \$i_1 w_2 \cdots w_k \$i_k w_{k+1}$. Moreover, let $e'_{i_1}, \ldots, e'_{i_k}$ be the subexpressions of pat corresponding to the i_1th, …, i_kth capturing groups. Note, in particular, that we may have $i_j = i_{j'}$ with $j \neq j'$. By abuse of notation, we will write $\mathsf{rep}[x_{e'_{i_1}}/\$i_1, \cdots, x_{e'_{i_k}}/\$i_k]$ to denote the replacement of the references, in order of appearance, by the contents of the variables $x_{e'_{i_j}}$. E.g., if $\mathsf{rep} = a\$1a\$2a\$1a$ then $\mathsf{rep}[x_{e'_{i_1}}/\$i_1, x_{e'_{i_2}}/\$i_2, x_{e'_{i_3}}/\$i_3]$ is $aw_1aw_2aw_3a$ when w_j is the value stored in $x_{e'_{i_j}}$.

Although, in this notation, it is possible to substitute different occurrences of a reference with different word values, our encoding will always substitute the same value. That is, if $i_j = i_{j'}$ then x_j will contain the same value as $x_{j'}$. We use two variables in this case to satisfy the copyless property [Alur and Cerný 2010], which leads to improved complexity results in some cases (discussed in the sequel). If we tried to use a single variable – e.g. $\mathsf{rep}[x/\$i_1, x/\$i_2]$ – then the resulting transition in the encoding below would not be copyless.

Suppose $\mathcal{T}_{\mathsf{pat}} = (Q_{\mathsf{pat}}, \Sigma, X_{\mathsf{pat}}, \delta_{\mathsf{pat}}, \tau_{\mathsf{pat}}, q_{\mathsf{pat},0}, (F_{\mathsf{pat},1}, F_{\mathsf{pat},2}))$. Then $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ is obtained from $\mathcal{T}_{\mathsf{pat}}$ by adding a fresh state q'_0 such that (see Fig. 11)

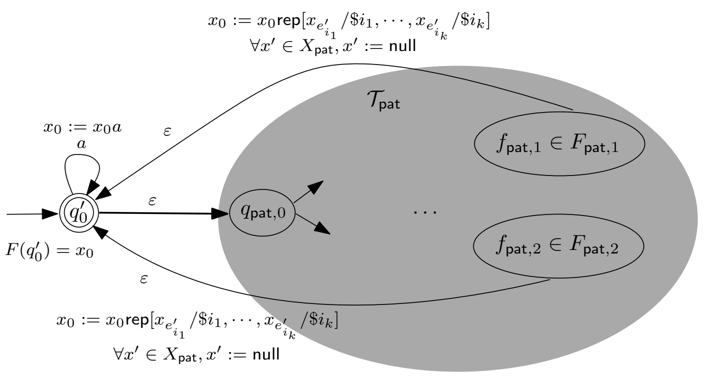
- $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ goes from q'_0 to $q_{\mathsf{pat},0}$ via an ε-transition of higher priority than the non-ε-transitions, in order to search the first match of pat starting from the current position,
- when $\mathcal{T}_{\mathsf{replaceAll}_{\mathsf{pat},\mathsf{rep}}}$ 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,
- 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, appends the replacement string, which is $\mathsf{rep}[x_{e'_{i_1}}/\$_{i_1}, \cdots, x_{e'_{i_k}}/\$_{i_k}]$, to the end of x_0, resets the values of al 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 $x_0 = x_0 w_1 x_{e'_{i_1}} w_2 \cdots w_k x_{e'_{i_k}} w_{k+1}$ and $x_{e'} = \mathsf{null}$ for all the variables $x_{e'} \in X_{\mathsf{pat}}$. 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, state its correctness, and observe that it gives rise to a decision procedure for the fragment $\mathsf{STR}_{\mathsf{SL}}$ of 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



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

Fig. 11. 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., $\mathcal{L}(e^c) = \Sigma^* \setminus \mathcal{L}(e)$.

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

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

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

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

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

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

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

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

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

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

a suitable strategy for applying the rules, as well as standard SMT optimizations such as non-chronological back-tracking and conflict-driven learning. An implementation also has to choose a suitable effective representation of RegEx membership constraints, for instance using finite-state

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

Fig. 12. Proof of unsatisfiability for (3) in Example 5.1

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

Fig. 13. Proof of satisfiability for (4) in Example 5.2

automata.[4] 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 [Gentzen 1935] 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[5]:

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

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 (Figure 12). 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.

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

[5]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.

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

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

Figure 13 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., [Harrison 2009]. Rule $\notin$ 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 use 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. 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, $\text{replace}_{\text{pat},\$0\$0}$ 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* [Carton et al. 2006]. 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 [Abdulla et al. 2014; Chen et al. 2019]. 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 Appendix A.3. Moreover, we have already shown in Lemma 4.6 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 STR_{SL}

One of the main results of this paper is the decidability of the STR_{SL} fragment of straightline formulas including concatenation, extract, replace, and replaceAll:

Theorem 5.7. *Satisfiability of STR_{SL} formulas is decidable.*

Proof. We define a terminating strategy to apply the rules in Table 1 to formulas in the STR_{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. In addition apply =-Prop to systematically propagate constraints across equations. This phase terminates because STR_{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 Appendix A.4.

When $\$^{\leftarrow}$ and $\$^{\rightarrow}$ are not used, the PSSTs in the reduction are copyless, which means the exponential blow-up in the size of the input FA $\mathcal{A}$ can be avoided. This means that the exponentials do not stack on top of each other during the backwards analysis and the non-elementary blow-up is not necessary.

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

For our experiments, we have implemented an SMT solver, EMU,[6] based on the calculus for STR. EMU extends the open-source solver OSTRICH [Chen et al. 2019], and is able to decide satisfiability of $\mathrm{STR}_{\mathrm{SL}}$ formulas. EMU also inherits support for most of the other operations of the SMT-LIB theory of Unicode strings[7] from OSTRICH.

6.1 Implementation

Our solver extends classical regular expressions in SMT-LIB with indexed re.capture and re.reference operators, which denote capturing groups and references to them. We also add re.*?, re.+?, and re.loop? as the lazy counterparts of Kleene star, plus 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. As an example, 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 $\mathrm{extract}_{i,e}$ in STR. For instance,

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

[6]Name anonymized for doubly-blind review, and will be provided in the final version.
[7]http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml

```
(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);
```

Fig. 14. Harnesses with replace-all: SMT-LIB for EMU (left), and JavaScript for ExpoSE (right).

extracts the left-most, longest sub-string consisting of only 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.

The implementation in EMU revolves around PSSTs. The three string operators mentioned above will be converted into an equivalent PSST (see Appendix A.2). EMU 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.

6.2 Experimental evaluation

Our experiments have the purpose of answering the following main questions:

R1: How does EMU compare to other solvers that can handle real-world regular expressions, including greedy/lazy quantifiers and capturing groups?
R2: How does EMU 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 [de Moura and Bjørner 2008] for solving string constraints, but augments it with a refinement loop to approximate the RegEx semantics.[8] For **R1**, we compared EMU with ExpoSE+Z3 on 98,117 RegExes taken from [Davis et al. 2019].

For each of the regular expressions, we created four harnesses: two in SMT-LIB, as inputs for EMU, and two in JavaScript, as inputs for ExpoSE+Z3. The two harnesses shown in Fig. 14 use one of the regular expressions from [Davis et al. 2019] (`<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

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

	EMU						ExpoSE+Z3				
	# queries solved within 60s						# paths covered within 60s				
	0	1	2	3	4	#Err	0	1	2	3	4
Match	172	92	61	534	96,126	1132	3,333	9,274	36,916	48,594	
(98,117	Average time: 1.19s						Average time: 28.0s				
benchm.)	Total #sat: 249,975, #unsat: 136,345						Total #paths covered: 228,888				
Replace	445	229	576	95,735	—	1,132	5,281	18,221	69,059	5,556	—
(98,117	Average time: 1.48s						Average time: 55.0s				
bench.)	Total #sat: 273,927, #unsat: 14,659						Total #paths covered: 173,007				

Table 2. The number of queries answered by EMU, and number of paths covered by ExpoSE+Z3, in the **R1** experiments. 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.

to the ones in Fig. 14, 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. EMU is able to answer all four queries in 96,126 of the match benchmarks (97.9%), and all three queries in 95,735 of the replace-all benchmarks (97.6%). The errors in 1,132 cases are due to back-references in `<re1>`, which are not handled by EMU. ExpoSE+Z3 can on the match problems cover 228,888 paths in total (91.5% of the number of sat results of EMU), although the runtime of ExpoSE+Z3 is on average 23x higher than that of EMU. For replace, ExpoSE+Z3 can cover 173,007 paths (63.2%), showing that this class of constraints is harder; the runtime of ExpoSE+Z3 is on average 37x higher than that of EMU. Overall, even taking into account that ExpoSE+Z3 has to analyze JavaScript code, as opposed to the SMT-LIB given to EMU, the experiments show that EMU is a highly competitive solver for RegExes.

For ***R2****:* For this experiment, we integrated EMU into the symbolic execution tool Aratha [Amadini et al. 2019]. We compare Aratha+EMU with ExpoSE+Z3 on the regression test suite of ExpoSE [Loring et al. 2017], 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+EMU can within 60s cover slightly more paths than ExpoSE+Z3. Aratha+EMU can discover feasible paths more quickly than ExpoSE+Z3, however: on all three families of benchmarks, Aratha+EMU terminates on average in less than 10s, and it discovers all paths within 35s. ExpoSE+Z3 needs full 60s for 29 of the programs ("T.O." in the table), and it finds new paths until the end of the 60s. 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 EMU is more efficient than the CEGAR-augmented symbolic execution for dealing with RegExes.

7 RELATED WORK

7.1 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 [Câmpeanu et al. 2003; Carle and Narendran 2009] and also more recently in [Berglund and van der Merwe 2017b; Freydenberger 2013; Freydenberger and Schmid 2019; Schmid 2016], 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

	Aratha+EMU						ExpoSE+Z3					
	# paths covered within 60s						# paths covered within 60s					
	0	1	2	3	≥4	#T.O.	0	1	2	3	≥4	#T.O.
ExpoSE *(49 programs)*	14	9	9	2	15	0	15	8	9	2	15	1
	Average time: **6.49**s						Average time: 13.46s					
	Total #paths covered:124						Total #paths covered:120					
Match *(28 programs)*	3	7	12	6	0	0	3	8	12	5	0	6
	Average time: **4.30**s						Average time: 23.66s					
	Total #paths covered: 49						Total #paths covered: 47					
Replace *(38 programs)*	12	20	6	0	0	0	12	23	3	0	0	22
	Average time: **2.71**s						Average time: 41.73s					
	Total #paths covered: 32						Total #paths covered: 29					

Table 3. Results of Expose+Z3 and Aratha+EMU on Javascript programs for **R2**. Experiments were done on an Intel-Xeon-E5-2690-@2.90GHz machine, running 64-bit Linux and Java 1.8. Runtime was limited to 60s wall-clock time. Average time is wall-clock time needed per benchmark, and counts timeouts as 60s. #T.O. is the number of timeouts. Note that some paths may have already been covered before T.O.

therein. In the software engineering community, some empirical studies were recently reported for these regular expressions, including portability across different programing languages [Davis et al. 2019] and DDos attacks [Staicu and Pradel 2018], as well as how programmers write them in practice [Michael et al. 2019].

Prioritized finite-state automata and transducers were proposed in [Berglund and van der Merwe 2017a]. 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 in general, e.g., swapping the first and last name as in Example 1.1. In contrast, PSSTs store the matches of capturing groups in string variables, which can then be referred to, thus allowing us to conveniently model the match and replace(all) function. Streaming string transducers were also used in [Zhu et al. 2019] to solve the straight-line string constraints with concatenation, finite-state transducers, and regular constraints.

7.2 String Constraint Solving

As we discussed Section 1, there has been a wealth of research activities focussing on the development of string constraint solving algorithms, especially in the past ten years or so. Solvers are typically using a combination of different 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 [de Moura and Bjørner 2008], CVC4 [Liang et al. 2014], Z3-str/2/3/4 [Berzish et al. 2017; Berzish, Murphy 2021; Zheng et al. 2015, 2013], ABC [Bultan and contributors 2015], Norn [Abdulla et al. 2014], Trau [Abdulla et al. 2017, 2018; Bui and contributors 2019], OSTRICH [Chen et al. 2019], S2S [Le and He 2018], Qzy [Cox and Leasure 2017], Stranger [Yu et al. 2010], Sloth [Abdulla et al. 2019; Holík et al. 2018], Slog [Wang et al. 2016], Slent [Wang et al. 2018], Gecode+S [Scott et al. 2017], G-Strings [Amadini et al. 2017], HAMPI [Kiezun et al. 2012], and S3 [Trinh et al. 2014]. Most modern string solvers provide support of concatenation and regular constraints. The push (e.g. see [Ganesh and Berzish 2016; Ganesh et al. 2012; Kiezun et al. 2012; Lin and Barceló 2016; Saxena et al. 2010; Trinh et al. 2014]) towards incorporating other functions—e.g. length, string-number conversions, replace, replaceAll—in a string theory has been an important theme in the area, owing to the desire to be able to reason about increasingly complex real-world string-manipulating programs. These functions, among others, are now part of the SMT-LIB Unicode Strings standard.[9]

To the best of our knowledge, at the moment there is no solver that supports RegEx features like greedy/lazing matching or capturing groups (apart from our own solver EMU). This was also remarked in [Loring et al. 2019], 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 EMU. Note, however, that the features that we do not cover are also rare in practice, according to [Loring et al. 2019] — 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 EMU 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 [Amadini 2020]. In this work, we consider a string constraint language which is undecidable in general, and propose 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. The decision procedure we use extends the backward-reasoning approach in [Chen et al. 2019], 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 [Loring et al. 2019] 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 EMU 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 generally undecidable, we have also shown that our solver terminates (and therefore is complete) for the so-called 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 EMU. This would essentially lift EMU to a symbolic execution engine (i.e. working at the level of programs). Secondly, it would be desirable to incorporate other features of RegEx that are not yet considered in our framework, e.g., lookahead and backreferences. 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 integer data types.

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

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. 2017. 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*. 602–617. https://doi.org/10.1145/3062341.3062384

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. 2018. Trau: SMT solver for string constraints. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–5. https://doi.org/10.23919/FMCAD.2018.8602997

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String Constraints for Verification. In *CAV*. 150–166.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukás Holík, and Petr Janku. 2019. Chain-Free String Constraints. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. 277–293. https://doi.org/10.1007/978-3-030-31784-3_16

Rajeev Alur and Pavol Cerný. 2010. 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*. 1–12.

Rajeev Alur and Jyotirmoy V. Deshmukh. 2011. Nondeterministic Streaming String Transducers. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II (Lecture Notes in Computer Science)*, Luca Aceto, Monika Henzinger, and Jirí Sgall (Eds.), Vol. 6756. Springer, 1–20.

Roberto Amadini. 2020. A Survey on String Constraint Solving. *CoRR* abs/2002.02376 (2020). arXiv:2002.02376 https://arxiv.org/abs/2002.02376

Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2019. Constraint Programming for Dynamic Symbolic Execution of JavaScript. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings (Lecture Notes in Computer Science)*, Louis-Martin Rousseau and Kostas Stergiou (Eds.), Vol. 11494. Springer, 1–19. https://doi.org/10.1007/978-3-030-19212-9_1

Roberto Amadini, Graeme Gange, Peter J. Stuckey, and Guido Tack. 2017. A Novel Approach to String Constraint Solving. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings (Lecture Notes in Computer Science)*, J. Christopher Beck (Ed.), Vol. 10416. Springer, 3–20. https://doi.org/10.1007/978-3-319-66158-2_1

Martin Berglund, Frank Drewes, and Brink van der Merwe. 2014. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. In *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014 (EPTCS)*, Zoltán Ésik and Zoltán Fülöp (Eds.), Vol. 151. 109–123. https://doi.org/10.4204/EPTCS.151.7

Martin Berglund and Brink van der Merwe. 2017a. On the semantics of regular expression parsing in the wild. *Theoretical Computer Science* 679 (2017), 69 – 82.

Martin Berglund and Brink van der Merwe. 2017b. Regular Expressions with Backreferences Re-examined. In *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*, Jan Holub and Jan Zdárek (Eds.). Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 30–41. http://www.stringology.org/event/2017/p04.html

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

Berzish, Murphy. 2021. *Z3str4: A Solver for Theories over Strings*. Ph.D. Dissertation. http://hdl.handle.net/10012/17102

Egon Börger, Erich Grädel, and Yuri Gurevich. 1997. *The Classical Decision Problem*. Springer.

Diep Bui and contributors. 2019. Z3-Trau. https://github.com/diepbp/z3-trau.

Tevfik Bultan and contributors. 2015. ABC string solver. https://github.com/vlab-cs-ucsb/ABC.

Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. 2003. A Formal Study Of Practical Regular Expressions. *Int. J. Found. Comput. Sci.* 14, 6 (2003), 1007–1018.

Benjamin Carle and Paliath Narendran. 2009. On Extended Regular Expressions. In *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings (Lecture Notes in Computer Science)*, Adrian-Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide (Eds.), Vol. 5457. Springer, 279–289.

Olivier Carton, Christian Choffrut, and Serge Grigorieff. 2006. Decision problems among the main subfamilies of rational relations. *ITA* 40, 2 (2006), 255–275. https://doi.org/10.1051/ita:2006005

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

Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2019. Decision Procedures for Path Feasibility of String-Manipulating Programs with Complex Operations. *PACMPL* 3, POPL, Article 49 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290362

Arlen Cox and Jason Leasure. 2017. Model Checking Regular Language Constraints. arXiv:1708.09073 [cs.LO]

James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. 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)*. Association for Computing Machinery, New York, NY, USA, 443–454.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. 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.* 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Emmanuel Filiot and Pierre-Alain Reynier. 2017. Copyful Streaming String Transducers. In *Reachability Problems - 11th International Workshop, RP 2017, London, UK, September 7-9, 2017, Proceedings (Lecture Notes in Computer Science)*, Matthew Hague and Igor Potapov (Eds.), Vol. 10506. Springer, 75–86.

Dominik D. Freydenberger. 2013. Extended Regular Expressions: Succinctness and Decidability. *Theory Comput. Syst.* 53, 2 (2013), 159–193.

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

Vijay Ganesh and Murphy Berzish. 2016. Undecidability of a Theory of Strings, Linear Arithmetic over Length, and String-Number Conversion. *CoRR* abs/1605.09442 (2016). http://arxiv.org/abs/1605.09442

Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. 2012. 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*. 209–226. https://doi.org/10.1007/978-3-642-39611-3_21

Gerhard Gentzen. 1935. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift* 39 (1935), 176–210, 405–431. English translation, "Investigations into Logical Deduction," in [Szabo 1969].

John Harrison. 2009. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press. I–XIX, 1–681 pages.

Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. 2018. String constraints with concatenation and transducers solved efficiently. *PACMPL* 2, POPL (2018), 4:1–4:32. https://doi.org/10.1145/3158092

Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium*.

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

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

Quang Loc Le and Mengda He. 2018. A Decision Procedure for String Logic with Quadratic Equations, Regular Expressions and Length Constraints. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science)*, Sukyoung Ryu (Ed.), Vol. 11275. Springer, 350–372. https://doi.org/10.1007/978-3-030-02768-1_19

Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *CAV*. 646–662.

Anthony W. Lin and Pablo Barceló. 2016. String Solving with Word Equations and Transducers: Towards a Logic for Analysing Mutation XSS *(POPL '16)*. ACM, 123–136.

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: practical symbolic execution of standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, Hakan Erdogmus and Klaus Havelund (Eds.). ACM, 196–199. https://doi.org/10.1145/3092282.3092295

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. 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.* ACM, 425–438.

Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. 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)*. IEEE Press, 415–426. https://doi.org/10.1109/ASE.2019.00047

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *J. ACM* 53, 6 (2006), 937–977.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. 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*. 513–528. https://doi.org/10.1109/SP.2010.38

Markus L. Schmid. 2016. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inf. Comput.* 249 (2016), 1–17.

Joseph D. Scott, Pierre Flener, Justin Pearson, and Christian Schulte. 2017. Design and Implementation of Bounded-Length Sequence Variables. In *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings (Lecture Notes in Computer Science)*, Domenico Salvagnin and Michele Lombardi (Eds.), Vol. 10335. Springer, 51–67. https://doi.org/10.1007/978-3-319-59776-8_5

Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in Javascript-Based Web Servers. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 361–376.

M. E. Szabo (Ed.). 1969. *The Collected Papers of Gerhard Gentzen*. North-Holland, Amsterdam.

Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *CCS*. 1232–1243.

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2016. 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*. Springer, 218–240.

Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. 2016. 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 (Lecture Notes in Computer Science)*, Vol. 9779. Springer, 241–260. https://doi.org/10.1007/978-3-319-41528-4

Hung-En Wang, Shih-Yu Chen, Fang Yu, and Jie-Hong R. Jiang. 2018. 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)*. ACM, 623–633. https://doi.org/10.1145/3238147.3238189

Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2010. Stranger: An Automata-Based String Analysis Tool for PHP. In *TACAS*. 154–157. Benchmark can be found at http://www.cs.ucsb.edu/~vlab/stranger/.

Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. 2014. Automata-based Symbolic String Analysis for Vulnerability Detection. *Form. Methods Syst. Des.* 44, 1 (2014), 44–70.

Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. 2015. 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*. Springer, 235–254. https://doi.org/10.1007/978-3-319-21690-4_14

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

Qizhen Zhu, Hitoshi Akama, and Yasuhiko Minamide. 2019. Solving String Constraints with Streaming String Transducers. *Journal of Information Processing* 27 (2019), 810–821. https://doi.org/10.2197/ipsjjip.27.810