DOI 10.1007/s00165-011-0214-3
BCS © 2012
Formal Aspects of Computing

Formal Aspects of Computing

Simple linear string constraints

Xiang Fu[1], Michael C. Powell[1], Michael Bantegui[1] and Chung-Chih Li[2]

[1] Department of Computer Science, Hofstra University, Hempstead, NY 11549, USA
E-mail: Xiang.Fu@hofstra.edu, michaelpowellcs@gmail.com, mbante2@gmail.com
[2] School of Information and Technology, Illinois State University, Normal, IL 61790, USA
E-mail: cli2@ilstu.edu

Abstract. Modern web applications often suffer from command injection attacks. Even when equipped with sanitization code, many systems can be penetrated due to software bugs. It is desirable to automatically discover such vulnerabilities, given the bytecode of a web application. One approach would be symbolically executing the target system and constructing constraints for matching path conditions and attack patterns. Solving these constraints yields an attack signature, based on which, the attack process can be replayed. Constraint solving is the key to symbolic execution. For web applications, string constraints receive most of the attention because web applications are essentially text processing programs. We present *simple linear string equation* (*SISE*), a decidable fragment of the general string constraint system. SISE models a collection of regular replacement operations (such as the greedy, reluctant, declarative, and finite replacement), which are frequently used by text processing programs. Various automata techniques are proposed for simulating procedural semantics such as left-most matching. By composing atomic transducers of a SISE, we show that a recursive algorithm can be used to compute the solution pool, which contains the value range of each variable in concrete solutions. Then a concrete variable solution can be synthesized from a solution pool. To accelerate solver performance, a symbolic representation of finite state transducer is developed. This allows the constraint solver to support a 16-bit Unicode alphabet in practice. The algorithm is implemented in a Java constraint solver called SUSHI. We compare the applicability and performance of SUSHI with Kaluza, a bounded string solver.

Keywords: String analysis; Symbolic execution; Constraint solving; Vulnerability detection

1. Introduction

Automatic vulnerability detection is a key problem in computer security research. In the context of web application analysis, we are interested in a variation of the problem: *Assuming that the binary code of a web application is available, can a "malicious" input be synthesized automatically?* In particular, it is challenging to find exploits that bypass user input sanitization procedures.

Correspondence and offprint requests to: X. Fu, E-mail: Xiang.Fu@hofstra.edu
This paper is based on the preliminary findings reported in [FLP+07, FQ08, FL10a, FL10b].

Published online: 17 January 2012

A recent emerging paradigm for detecting software vulnerabilities is the application of symbolic execution [Kin76]. A target system is symbolically executed where the program inputs are treated as symbolic literals. Path condition, a symbolic constraint, is used to trace the conjunction of all branch conditions encountered during an execution. At critical points, e.g., where a SQL query is submitted, path conditions are paired with attack patterns. Solving these constraints then leads to attack signatures. The variations of this paradigm are widely applied, e.g., the EXE project [CGP+06] uses symbolic execution to find memory reference defects that crash operating systems. Moser, Kruegel, and Kirda adopt dynamic symbolic execution for malware analysis [MKK07]. In the area of web application security, there are several related efforts that rely on constraint solving, e.g., the JDBC Checker [GSD04], the Stranger analyzer [YBI09, YAB10, YAB09], the HAMPI constraint solver [KGJE09, KGG+09], the DRPLE decision procedure [HW09], and the Kudzu analyzer [SAH+10]. SAFELI/JavaSye [FLP+07, FQ08], our earlier work in this area, instruments Java web applications for detecting potential SQL injection vulnerabilities.

The success of symbolic execution relies on the power of constraint solvers. In the arena of web application security, "strings" receive special attention because web applications are essentially text processing programs—they accept HTTP parameters in the form of strings and produce outputs as strings (i.e., HTML documents). Many application level attacks are related to strings, e.g., SQL injection [Anl02] usually takes advantage of string concatenation operations, and Cross-Site Scripting (XSS) attacks [Raf01] are sensitive to the matching of script tags. In the following, we refer to the problem of solving constraints related to strings as "string constraint solving".

The major contribution of this paper is the *simple linear string equation* (*SISE*) [FL10a, FL10b], a decidable string constraint theory. SISE can be used for representing path conditions and attack patterns. A SISE equation resembles a word equation [Lot02]: it is built upon word literals and string variables. The major advantage of SISE is its support of typical procedural semantics of regular replacement operations (such as greedy and reluctant semantics), adopted by various programming languages such as PHP, Java, and Perl.

Similar to many on-going efforts in the area (e.g., HAMPI [KGG+09], Stranger [YAB10], Kaluza [SAH+10], "Pure Library Language" of .Net [BTV09], and DRPLE [HW09, HW10]), the design of SISE has to consider the trade-off between expressiveness and decidability. For example, *unbounded string length* and *replacement operations*, two important features of SISE, can lead to the undecidability of a general string constraint system. The proof can be adapted from the one shown by Bjørner, Tillmann, and Voronkov for the full .Net string library [BTV09] (using an earlier result by Büchi and Senger in [BS88]). To make the SISE theory decidable, we have to impose certain syntactic restrictions, such as limiting the occurrence of variables. That is the reason we call SISE a *linear* string equation system.

These syntactic restrictions permit an automata based solution, which breaks down a SISE into a number of atomic string operations. Each atomic operation is then solved by a *backward image computation*. This is quite different from the solution of word equations using Makanin's algorithm [Mak77]. Given a set of strings R, and given a string operation f (e.g., `substring` and `charAt`), the backward image of R w.r.t. f is a maximal set of strings X where for each string $s \in X : f(s) \in R$. For most string operations, backward image computation can be defined using regular expressions. Solving string substitution can be realized using finite state transducers. String concatenation operations are handled by standard automata algorithms for processing regular expression quotient.

Different from many recent works on string constraints (e.g., [BTV09, YAB09, KGG+09, HW09]), the SISE theory supports precise modeling of several popular regular replacement operations. This is motivated by the wide application of regular replacement in user input sanitization. This paper models the declarative, finite, greedy, and reluctant replacement, using finite state transducers (FST). By projecting an FST to its input/output tapes, one can compute the backward and forward images of atomic regular replacement operations. This paper also reports several interesting discoveries, e.g., finite regular replacement and most operations of a reluctant replacement can be modeled using deterministic FST, which is strictly weaker than a nondeterministic FST.

A string constraint solver named SUSHI is constructed for solving SISE constraints. The source code of the tool is available at [Fu09]. SUSHI supports a 16-bit Unicode alphabet. As an explicit encoding of FST transitions cannot meet the performance needs of security analysis, we have developed a compact symbolic representation of FST and a set of customized FST operations. SUSHI is applied to finding delicate SQL injection and XSS attacks. It can be extended for discovering other command injection attacks such as request forgery attacks [Shi04], format string attacks [New00], and PHP injection attacks [Chr06].

```
1  protected void processRequest(
2   HttpServletRequest request ...)
3   throws ServletException{
4   PrintWriter out = response.getWriter();
5   try {
6     String sUname = request.getParameter("sUname");
7     String sPwd = request.getParameter("sPwd");
8     Connection conn = DriverManager.getConnection("...");
9     Statement stmt = conn.createStatement();
10    String strCmd= "SELECT * FROM users\nWHERE uname='"
11      + massage(sUname) + "' AND pwd='"
12      + massage(sPwd) + "'";
13    ResultSet srs = stmt.executeQuery(strCmd);
14    if(srs.next()){
15     out.println("Welcome " + sUname);
16    }else{
17     out.println("Login fail!");
18    }
19   }catch(Exception exc){...}
20  }
21
22  protected String massage(String str){
23    String strOut = str.replaceAll("'", "''");
24    if(strOut.length()>16) return strOut.substring(0,16);
25    return strOut;
26  }
```

Listing 1. Vulnerable authentication

The rest of the paper is organized as follows. Section 2 motivates the research with two application examples. Section 3 formalizes the notion of simple linear string equation. Section 4 introduces formal models of declarative and finite replacement semantics. Section 5 presents the reluctant and greedy regular replacement operations. Section 6 provides a recursive algorithm for solving a SISE string constraint. Section 7 briefly discusses the implementation details of the SUSHI constraint solver. Section 8 reports the experimental evaluation of SUSHI and compares it with the Kaluza constraint solver [SAH+10]. Section 9 discusses related work, and Sect. 10 concludes.

2. Motivating examples

This section presents two application examples. The first example (originally introduced in [FLP+07]) shows that it is possible to discover a password bypassing attack using the string constraint solving technique. The second demonstrates the need for precisely modeling various regular replacement semantics.

This paper assumes the availability of symbolic execution, and the discussion is not restricted to one programming language. This is based on the observation that many symbolic execution frameworks are emerging recently. The following are a few examples: JPF-SE [APV07] for Java, Kudzu [SAH+10] for JavaScript, BitFuzz [CPM+10] for X86 binaries, and RubyX [CF10] for Ruby-on-Rails.

2.1. Example 1: SQL injection for password bypassing

2.1.1. Vulnerable sanitization

Listing 1 displays a Java servlet called `BadLogin`, which provides the user authentication service for a web-email system. Function `processRequest` reads a user name and a password from the HTTP request (lines 6 to 7), and verifies if they do exist in the back-end database (lines 10 to 18).

The servlet is equipped with a sanitization function named `massage` (lines 22 to 26). In SQL injection attacks, hackers very often take advantage of single quote characters to change the logical structure of a SQL query being constructed. To prevent this, `massage` applies a number of counter approaches. First, at line 23, it replaces every single quote character with its escaping form (a sequence of two single quotes). In addition, it limits the size of each user input string to 16 characters (at line 24). Here the size restriction (16) can actually be any positive integer.

The length restriction protection intends to limit the room of attackers for playing tricks. *Good intention, however, may not eventually lead to desired effects!* Combined with string substitution, it actually causes a delicate vulnerability and the following is the *shortest* attack signature discovered by SUSHI. The first string (value of `sUname`) is 9 characters long (starting with an 'a' and continued with eight single quotes). The second string (value of `sPwd`) is 14 characters long (starting with a single quote character).

```
sUname:  a''''''''
sPwd:    '' OR uname<>'
```

Readers can verify that the above attack signature indeed works. By applying the `massage` function on `sUname`, each of the eight single quotes in `sUname` is converted to a sequence of two single quotes. However, the last quote is chopped off by the `substring()` function (at line 24 of Listing 1). Similar handling is applied to `sPwd`. This results in a malicious SQL query (displayed in Listing 2), which bypasses password checking.

```
SELECT * FROM users
WHERE uname='a''''''''''''''''_AND_pwd=''''' OR uname<>''
```

Listing 2. Malicious SQL Query

Notice that the `WHERE` clause in Listing 2 has a very interesting structure. Each pair of single quotes between `a` and `AND` is regarded as the escaping form by SQL (i.e., it is treated as one single quote character instead of a control symbol). This causes the logical structure of the `WHERE` clause to be a *disjunction* of two conditions. Since the `uname` field (the primary key of `users`) cannot be empty, this makes the `WHERE` clause a tautology in practice, which eventually leads the execution of `processRequest` to line 15 in Listing 1, i.e., the hacker gets into the system without knowing a correct password.

2.1.2. *Vulnerability detection using SUSHI*

We now briefly describe how the SUSHI constraint solver can be used for discovering the aforementioned attack. First, the `processRequest` function is symbolically executed. Input variables `sUname` and `sPwd` are initialized with symbolic literals. Let them be x and y, respectively. Then, by executing the statements one by one, at line 13, where the SQL query is submitted, the symbolic value of `strCmd` is represented as a string expression. It is a concatenation of five string terms (T_1 to T_5), simulating the structure of the concatenation statement at lines 10 to 12. Note that the contents of constant words are denoted using the `courier` font.

1. T_1: constant word `SELECT * FROM users \nWHERE uname='`
2. T_2: term $x^{+}_{\rightarrow''}[0, 16]$
3. T_3: constant word `' AND pwd='`
4. T_4: term $y^{+}_{\rightarrow''}[0, 16]$
5. T_5: constant word `'`

SISE constraints have a collection of operators to model string operations provided by programming languages (e.g., those of the `java.lang.String` class). For example, in term T_2, $x^{+}_{\rightarrow''}[0, 16]$ represents the sanitization performed by `massage` on `sUname`. It replaces every single quote with its escaping form (two single quote characters) and then performs a substring extraction operation on the user input. Here a greedy replacement semantics is used, which is denoted using "+" in the formula. Similar is $y^{+}_{\rightarrow''}[0, 16]$, which applies the same sanitization on `sPwd`. Now by associating the symbolic string expression with predefined attack patterns, we can construct SISE equations. For example, the following is a sample SISE equation, based on one of the pre-collected SQL injection attack patterns, where $\circ$ represents concatenation:

$$T_1 \circ T_2 \circ T_3 \circ T_4 \circ T_5 \quad \equiv \quad \texttt{SELECT * FROM users \nWHERE uname='([^'] | '')*' OR 'uname<>''} \tag{1}$$

Intuitively, the SISE equation asks the following question: after all the sanitization procedures are applied, is it feasible to make the `WHERE` clause of the SQL query essentially a tautology (the attack pattern is expressed using a regular expression "`OR *uname<>''`")? Using SUSHI, one can obtain a regular language (called "solution pool") capturing all solutions for x (and y as well). Starting from the solution pool, the concrete attack strings are generated. The entire constraint solving process takes 1.6 seconds on a workstation with a 2.0 GHz Intel Xeon CPU and 3GB RAM.

2.1.3. Discussion

The first example shows the power of the constraint solving techniques. Black-box vulnerability scanning tools such as Nikto [SL] and WebInspect [HP] will have difficulty in discovering the aforementioned vulnerability, because it is a "corner case" bug in the user input validation code. To simply embed special characters like single quotes in user input (or to mutate existing attack strings) will not likely uncover the bugs either. Only when the control/data flow information is analyzed by the vulnerability detection tool can such a deeply hidden vulnerability be revealed.

2.2. Example 2: cross-site scripting attack

Regular replacement operators are the key component of SISE constraints. The second example discusses the importance of precisely modeling the various semantics of regular replacement.

```
1 <?php
2   $msg = $_POST["msg"];
3   $sanitized =  preg_replace(
4                   "/\<script.*?\>.*?\<\/script.*?\>/i",
5                   "", $a);
6   save_to_db($sanitized)
7 ?>
```

Listing 3. Vulnerable XSS Sanitation

2.2.1. Precise modeling of regular replacement matters

Listing 3 shows a vulnerable PHP snippet called "`postMessage`". The code takes a message from an anonymous user and posts it on a bulletin. With a primitive protection against XSS attacks, the PHP program is still vulnerable.

At line 3, the program calls `preg_replace()` to remove any pair of `<script>` and `</script>` tags and the contents between them. Notice that the wild card operator `*?` is a *reluctant* operator, i.e., it matches the shortest string possible. For example, given word `<script>a</script></script>`, the call on line 3 returns `</script>`. If `*` (the greedy operator) is used, the call on line 3 returns an empty string.

A SISE equation can be constructed when symbolic execution reaches line 3. Let α be the regular expression `<script.*?>.*?</script.*?>` and ϵ the empty word. The following SISE equation asks if it is possible to have a JavaScript code snippet saved to the database, even after the regular replacement is applied.

$$\texttt{msg}^{-}_{\alpha\rightarrow\epsilon} \equiv \texttt{<script.*?>alert('a')</script.*?>}$$

SUSHI solves the SISE constraint and produces the solution to `msg` as below.

```
<<script></script>script>alert('a')</script>
```

Readers can verify that the replacement on the above string yields `<script>alert('a')</script>`. The trick is that after the reluctant replacement kills the *shortest* match (i.e., `<script></script>`), the first character "`<`" is then continued with the word "`script`" and forms a new working JavaScript code snippet.

Now, a natural question following the above analysis is: If one approximates the reluctant semantics using the greedy semantics, could the static analysis be still effective? The answer is negative. When the `*?` operators in Listing 3 are treated as `*`, SUSHI reports no solution for the aforementioned SISE equation, i.e., a false negative report on the vulnerable sanitization.

In summary, a precise modeling of the various regular replacement semantics is helpful in improving the precision of security analysis.

3. Simple linear string equation

This section presents the notion of simple linear string equation (SISE). We start with the introduction of string operators and string expressions.

3.1. Preliminaries

Let N denote the set of natural numbers and Σ a finite alphabet. # (begin marker) and \$ (end marker) are two reserved symbols not contained in Σ. Let $\Sigma_2 = \Sigma \cup \{\#, \$\}$. If $\omega \in \Sigma^*$, we say that ω is a word. $|\omega|$ denotes the size of ω. Given $0 \leq i < |\omega|$ and $i \leq j \leq |\omega|$, $\omega[i]$ is the i'th element of ω and $\omega[i, j]$ denotes the substring of ω starting at index i and ending at $j - 1$.[1] ϵ represents an empty word. The reverse of a word ω is denoted as $\texttt{reverse}(\omega)$. A word ω_1 is said to be a suffix of another word ω_2, written as $\omega_1 \dot{\prec} \omega_2$, if there exists $\omega_3 \in \Sigma^*$ s.t. $\omega_2 = \omega_3\omega_1$ (i.e., concatenation of ω_3 and ω_1). ω_1 is a prefix of ω_2, denoted using $\omega_1 \prec \omega_2$, if there exists $\omega_3 \in \Sigma^*$ s.t. $\omega_2 = \omega_1\omega_3$. Given a word ω, $PREFIX(\omega) = \{\omega' \mid \omega' \prec \omega\}$ and $SUFFIX(\omega) = \{\omega' \mid \omega' \dot{\prec} \omega\}$. Let R be the set of regular expressions over Σ. If $r \in R$, let $L(r)$ be the language represented by r. We abuse the notation by writing $\omega \in r$ if $\omega \in L(r)$, when the context is clear that r is a regular expression. There are infinitely many distinguishable string variables and let this set of variables be denoted by V. Given $\omega \in \Sigma_2^*$, the projection of ω to Σ, written as $\pi_\Sigma(\omega)$ is the result of removing all characters that do not belong to Σ. When context is clear, it is written as $\pi(w)$. Given $r \in R$, $PREFIX(r)$ is $\bigcup_{\omega \in r} PREFIX(\omega)$, and $SUFFIX(r)$ is defined similarly. For any regular expression $r \in R$, $\texttt{reverse}(r)$ denotes its reverse s.t. $L(\texttt{reverse}(r)) = \{\omega \mid \texttt{reverse}(\omega) \in r\}$.

3.2. String operator semantics

SISE supports a representative collection of string operators available in popular programming languages. This set is denoted using $O = \{\circ, [i, j], x_{r\to\omega}, x^-_{r\to\omega}, x^+_{r\to\omega}\}$. They are concatenation ($\circ$), substring extraction ($[i, j]$), declarative replacement ($x_{r\to\omega}$), reluctant regular replacement ($x^-_{r\to\omega}$), and greedy regular replacement ($x^+_{r\to\omega}$). Intuitively, the greedy replacement tries to find the longest match of r, and the reluctant matches the shortest. Both the reluctant and greedy operators enforce *left-most* matching, while the declarative replacement does not and it may produce multiple output words given one input word. The following example demonstrates their difference.

Example 3.1 Let $a, b \in \Sigma$, $s \in \Sigma^*$, and $r \in R$. Consider the following two cases. (i) If $s = aaab$, $r = (aa \mid ab)$, and $\omega = c$, then $s_{r\to\omega} = \{cc, acb\}$, and $s^-_{r\to\omega} = s^+_{r\to\omega} = cc$. (ii) If $s = aaa$, $r = a^+$, and $\omega = b$, then $s_{r\to\omega} = \{b, bb, bbb\}$, $s^-_{r\to\omega} = bbb$, and $s^+_{r\to\omega} = b$.

$s^-_{r\to\omega}$ and $s^+_{r\to\omega}$ are uniquely defined for any $s \in \Sigma^*$ and $r \in R$, because they are left-matching (characters are processed one by one from the left). They are also called *procedural replacement*. The formal definition of the three regular replacement operators is given below:

Definition 3.2 Let $s, \omega \in \Sigma^*$ and $r \in R$ with $\epsilon \notin r$.[2]

$$s_{r\to\omega} = \begin{cases} \{s\} & \text{if } s \notin \Sigma^* r \Sigma^*; \\ \{v_{r\to\omega}\omega\,\mu_{r\to\omega} \mid s = v\beta\mu, \beta \in r\} & \text{otherwise.} \end{cases}$$

If $s_{r\to\omega} = \{s\}$, *then let* $s^-_{r\to\omega} = s^+_{r\to\omega} = s$; *otherwise, define*

- $s^-_{r\to\omega} = v\omega\mu^-_{r\to\omega}$ where $s = v\beta\mu$ such that, $v \notin \Sigma^* r \Sigma^*$, and $\beta \in r$, and for every x, y, u, t, m, n with $v = xy$, $\beta = ut$, and $\mu = mn$: if $y \neq \epsilon$ then $yu \notin r$ and $y\beta m \notin r$; and if $t \neq \epsilon$ then $u \notin r$.
- $s^+_{r\to\omega} = v\omega\mu^+_{r\to\omega}$ where $s = v\beta\mu$ such that, $v \notin \Sigma^* r \Sigma^*$, and $\beta \in r$, and for every x, y, u, t, m, n with $v = xy$, $\beta = ut$, and $\mu = mn$: if $y \neq \epsilon$ then $yu \notin r$ and if $m \neq \epsilon$ then $y\beta m \notin r$. □

[1] String index starts from 0, i.e., $s[0]$ represents the first character of s, and $s[|\,s\,|-1]$ is its last element.

[2] The formal definition of the $\epsilon \in r$ case is given in Appendix A. It complies with the `Java.util.regex` semantics. For example, if $s = a$, $r = a^*$, and $\omega = b$, then $s^-_{r\to\omega} = bab$, $s^+_{r\to\omega} = bb$.

The above definition uses recursion to define the semantics of regular replacement. For both the greedy and reluctant semantics, the left-most matching procedure is enforced. For example, in the definition of $s^{-}_{r\to\omega}$, the requirement "if $y \neq \epsilon$ then $yu \notin r$ and $y\beta m \notin r$" enforces that β is the left-most matching of the regular search pattern r, i.e., there is no earlier matching of r than β.

3.3. String expression

Intuitively, a *string expression* is an expression over Σ with occurrences of variables in V and operators in O. String expressions are the building blocks of SISE.

Definition 3.3 Let E denote the set of string expressions, which is defined recursively as below:

1. If $x \in (V \cup \Sigma^*)$, then $x \in E$.
2. If $\mu, \nu \in E$, then $\mu\nu \in E$ (also written as $\mu \circ \nu$).
3. If $\mu \in E$, then $\mu[i, j] \in E$.
4. If $\mu \in E$, then $\mu_{r\to\omega}$, $\mu^{-}_{r\to\omega}$, $\mu^{+}_{r\to\omega} \in E$ for all $r \in R$ and $\omega \in \Sigma^*$.
5. Nothing else is in E except those described above. □

3.4. Simple linear string equation

The purpose of SISE is to capture a solvable fragment of those string constraints that arise from a symbolic execution and exploits of security attacks. Its syntax is defined as below.

Definition 3.4 A simple linear string equation (SISE) $\mu \equiv r$ is a string equation such that $\mu \in E$, $r \in R$ provided that every string variable occurs at most once in μ. □

Note that variables do not appear in the right hand side (RHS) of a SISE, because RHS is usually an attack pattern. The restriction of "one occurrence" allows a fast and simple recursive algorithm for solving a SISE equation. However, this restriction has trade-off – it limits the applicability of SISE in practice. In Sect. 8, We discuss the design decision (assumptions and limitations) of SISE, compare it with other string constraint systems, and present several extensions to improve its applicability.

The solution to a SISE is defined in the following. Intuitively, a solution is a mapping that sets the value of each variable.

Definition 3.5 Let φ be a SISE in the form of $\mu \equiv r$ and let $\vec{V}$ be the set of variables in μ. A solution to φ is a function $\rho : \vec{V} \to \Sigma^*$ s.t. $\rho(\mu) \cap L(r) \neq \emptyset$. Here $\rho(\mu)$ represents the set of words that are generated by replacing each variable v with $\rho(v)$ in μ.[3] □

4. Declarative and finite regular replacement

Regular replacement operators are the key component of SISE constraints. This section discusses how to model the declarative and finite regular substitution semantics. Finite state transducer (FST) [RE97, JM08] was widely applied, e.g., in processing phonological rules [KK94]. We find it also useful for modeling regular replacements. Notice that some advanced features such as back-references can not be modeled using FST. At this moment, it remains an open problem whether a regular search pattern of mixed greedy and reluctant operators can be modeled using FST.

[3] Note that some string operators such as $S_{r\to\omega}$ might produce multiple words. Thus, $\rho(\mu)$ is a set of words.

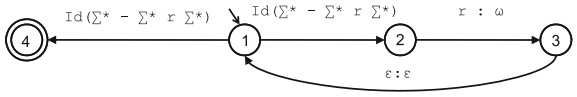


Fig. 1. FST for $s_{r\to\omega}$

4.1. Augmented finite state transducer

The standard FST model is given in Definition 4.1. An equivalent augmented FST model is presented in Definition 4.3.

Definition 4.1 Let Σ^ϵ denote $\Sigma \cup \{\epsilon\}$. A finite state transducer (FST) is an enhanced two-taped nondeterministic finite state machine described by a quintuple $(\Sigma, Q, q_0, F, \delta)$, where Σ is the alphabet, Q the set of states, $q_0 \in Q$ the initial state, $F \subseteq Q$ the set of final states, and δ is the transition function, which is a total function of type $Q \times \Sigma^\epsilon \times \Sigma^\epsilon \to 2^Q$. □

Given $\omega_1, \omega_2 \in \Sigma^*$ and an FST M, we say $(\omega_1, \omega_2) \in L(M)$ if the word pair is accepted by M. It is well known that an FST accepts a regular relation. Non-deterministic FST (NFST) is strictly more expressive than deterministic FST (DFST). FST is closed under concatenation, union, Kleene star, and composition; but it is not closed under complement and intersection. Composition of FST is very useful in modeling filters in our later discussion. It is formally defined as below:

Definition 4.2 Let M_3 be the composition of two FSTs M_1 and M_2, denoted as $M_3 = M_1 \,||\, M_2$, then $L(M_3) = \{(\mu, \nu) \mid (\mu, \eta) \in L(M_1)$ and $(\eta, \nu) \in L(M_2)$ for some $\eta \in \Sigma^*\}$. □

Let L_1 and L_2 be two languages. If an FST, M, accepts (ω_1, ω_2) iff $(\omega_1, \omega_2) \in L_1 \times L_2$, we say that M recognizes the language pair (L_1, L_2), and is denoted by $M_{L_1\times L_2}$. It is straightforward to argue that L_1 and L_2 are regular iff $M_{L_1\times L_2}$ exists. For convenience, we can extend the transition labels to regular relations, obtaining an *augmented FST*, denoted by AFST.

Definition 4.3 An augmented finite state transducer (AFST) is a quintuple $(\Sigma, Q, q_0, F, \delta)$ with the transition function δ augmented to $Q \times \mathcal{R} \to 2^Q$, where $\mathcal{R}$ is the set of regular relations over Σ. □

Lemma 4.4 *For each AFST M, there exists an FST M' s.t. $L(M) = L(M')$.*

The above lemma results from the fact that each transition in an AFST can be translated into an equivalent FST. Thus, AFST can be regarded as a succinct and hierarchical representation of FST. We alternatively use FST and AFST for the time being without loss of generality.

While we have tried to keep our setup as general as possible, we would often restrict the transition function of an AFST to the following two types: (1) $Q \times R \times \Sigma^* \to 2^Q$; and (2) $Q \times \{Id(r) \mid r \in R\} \to 2^Q$ where $Id(r) = \{(\omega, \omega) \mid \omega \in L(r)\}$. A type (1) arc is labeled as $r : \omega$ in a transition diagram, where $r \in R$ and $\omega \in \Sigma^*$. An arc of type (2) is denoted as $Id(r)$. By convention of [KK94], $Id(r)$ is called an identity relation.

4.2. Declarative regular replacement

The modeling of declarative regular replacement is straightforward using AFST. Given $s_{r\to\omega}$ for any $\omega \in \Sigma^*$ and $r \in R$ (with $\epsilon \notin r$), Fig. 1 shows its AFST model (letting it be $\mathcal{M}_{r\to\omega}$). Given any two $s, \eta \in \Sigma^*$, we can use the AFST to check if η is a string obtained from s by replacing every occurrence of patterns in r with ω. $\mathcal{M}_{r\to\omega}$ uses nondeterminism to handle the declarative nature of $s_{r\to\omega}$. It represents the following algorithm:

1. Transition $1 \to 2$ handles a substring that does not contain a match of the search pattern r. Such a substring is modeled using $\Sigma^* - \Sigma^* r \Sigma^*$. The transition simply returns the same string on the output tape, using the identity relation.

2. Transition $2 \rightarrow 3$ performs the replacement.
3. Transition $3 \rightarrow 1$ accomplishes the loop of the above two steps, until no match of r can be found.
4. Transition $1 \rightarrow 4$ handles the last substring that does not contain r. Note that because $\epsilon \notin r$, empty word ϵ is an element of $\Sigma^* - \Sigma^* r \Sigma^*$, this includes the case that the input word is ended with a match of r.

Lemma 4.5 *For any $r \in R$ s.t. $\epsilon \notin r$ and $\omega \in \Sigma^*$, $L(\mathcal{M}_{r\rightarrow\omega}) = \{(s, \eta) \mid s \in \Sigma^*$ and $\eta \in s_{r\rightarrow\omega}\}$.*

Proof. We first prove that for any $(s, \eta) \in L(\mathcal{M}_{r\rightarrow\omega})$: $\eta \in s_{r\rightarrow\omega}$. The claim holds for the vacuous case where $s = \eta$ (i.e., s does not contain any match of r). Then, assume that η is generated by visiting transition $1 \rightarrow 2$ for k times. Based on the observation of $\mathcal{M}_{r\rightarrow\omega}$, η can be written as $w_1\omega w_2\omega \ldots w_k\omega w_{k+1}$ where $s = w_1\beta_1 w_2\beta_2 \ldots w_k\beta_k w_{k+1}$; and for all $i \in [1, k]$: $\beta_i \in L(r)$ and $w_i \in \Sigma^* - \Sigma^* r \Sigma^*$; and $w_{k+1} \in \Sigma^* - \Sigma^* r \Sigma^*$. Now let $\nu = w_1\beta_1 \ldots w_k$, $\beta = \beta_k$, and $\mu = w_{k+1}$. Using induction, we could show that $w_1\omega \ldots w_k \in \nu_{r\rightarrow\omega}$. Then we have $\eta = \nu_{r\rightarrow\omega}\omega\, \mu_{r\rightarrow\omega}$ and $s = \nu\beta\, \mu$. This completes the proof of $\eta \in s_{r\rightarrow\omega}$ by Definition 3.2.

We now prove that for each $\eta \in s_{r\rightarrow\omega}$: $(s, \eta) \in L(\mathcal{M}_{r\rightarrow\omega})$. This could also be proved by induction on the length of η. By Definition 3.2, $\eta = \nu_{r\rightarrow\omega}\omega\, \mu_{r\rightarrow\omega}$ while $s = \nu\beta\, \mu$ with $\beta \in L(r)$. Based on the induction assumption, $(\nu, \nu_{r\rightarrow\omega})$ is accepted by $\mathcal{M}_{r\rightarrow\omega}$. Similar is $(\mu, \mu_{r\rightarrow\omega})$. Let R_1 and R_3 be the acceptance run of $(\nu, \nu_{r\rightarrow\omega})$ and $(\mu, \mu_{r\rightarrow\omega})$ on $\mathcal{M}_{r\rightarrow\omega}$, respectively. From the structure of $\mathcal{M}_{r\rightarrow\omega}$ we could see that R_1 consists of a loop of transitions $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$, and ends with a transition $1 \rightarrow 4$. Let R_1' be the run derived from R_1 by replacing the last transition (i.e., $1 \rightarrow 4$) with $1 \rightarrow 2$. Readers can verify that $R_1' \circ 2 \rightarrow 3 \rightarrow 1 \circ R_3$ is an acceptance run for (s, η). The transition $2 \rightarrow 3$ replaces β with ω. This immediately leads to $(s, \eta) \in L(\mathcal{M}_{r\rightarrow\omega})$. □

4.3. Finite regular replacement

This section shows that regular replacement with finite language pattern can be modeled (under a minor syntactic restriction) using DFST, which is strictly weaker than NFST. This subsumes constant word replacement.

We fix the notation of DFST first. Intuitively, at any state q of a DFST, the input symbol uniquely determines the destination state and the symbol on the output tape. If there is a transition labeled with ϵ on the input, then this is the only transition from q.

Definition 4.6 An FST $\mathcal{A} = (\Sigma, Q, s_0, F, \delta)$ is *deterministic* if for any $q \in Q$ and any $a \in \Sigma$ the following is true. Let $t_1, t_2 \in \{a, \epsilon\}$, $b_1, b_2 \in \Sigma \cup \{\epsilon\}$, and $q_1, q_2 \in Q$. $q_1 = q_2$, $t_1 = t_2$, and $b_1 = b_2$ if $q_1 \in \delta(q, t_1 : b_1)$ and $q_2 \in \delta(q, t_2 : b_2)$. □

Definition 4.7 A regular expression $r \in R$ is said to be *finite* if $L(r)$ is a finite set.

If r is finite, there exists a bound $n \in N$ s.t. for any $\omega \in L(r)$: $|\omega| \leq n$. This leads to the lemma below, which states that for any finite regular search pattern r, there exists a DFST for modeling the procedural replacement of r.

Lemma 4.8 *Let $\$ \notin \Sigma$ be an end marker. Given a finite regular expression $r \in R$ (with $\epsilon \notin r$) and a word $\omega_2 \in \Sigma^*$, there exist DFST $\mathcal{A}^-$ and $\mathcal{A}^+$ s.t. for any $\omega, \omega_1 \in \Sigma^*$: $\omega_1 = \omega^-_{r\rightarrow\omega_2}$ iff $(\omega\$, \omega_1\$) \in L(\mathcal{A}^-)$; and, $\omega_1 = \omega^+_{r\rightarrow\omega_2}$ iff $(\omega\$, \omega_1\$) \in L(\mathcal{A}^+)$.*

Proof. We briefly describe how $\mathcal{A}^+$ is constructed for $\omega^+_{r\rightarrow\omega_2}$. $\mathcal{A}^-$ can be created similarly. Given a finite regular expression r, and assume its length bound is n. Let $\Sigma^{\leq n} = \bigcup_{0\leq i\leq n} \Sigma^i$. Then $\mathcal{A}^+$ is defined as a quintuple $(\Sigma \cup \{\$\}, Q, q_0, F, \delta)$. The set of states Q has $|\Sigma^{\leq n}|$ elements, and let $\mathcal{B} : \Sigma^{\leq n} \rightarrow Q$ be a bijection. Let $q_0 = \mathcal{B}(\epsilon)$ be the initial state and the only final state. A transition $(q, q', a : b)$ is defined as follows for any $q \in Q$ and $a \in \Sigma \cup \{\$\}$, letting $\beta = \mathcal{B}^{-1}(q)$: (*case 1*) if $a \neq \$$ and $|\beta| < n$, then $b = \epsilon$ and $q' = \mathcal{B}(\beta a)$; or (*case 2*) if $a \neq \$$ and $|\beta| = n$: if $\beta \notin r\Sigma^*$, then $b = \beta[0]$ and $q' = \mathcal{B}(\beta[1, |\beta|]a)$; otherwise, let $\beta = \mu\nu$ where μ is the longest match of r, then $b = \omega_2$ and $q' = \mathcal{B}(\nu a)$; or (*case 3*) if $a = \$$, then $b = \beta^+_{r\rightarrow\omega_2}\$$ and $q' = q_0$. □

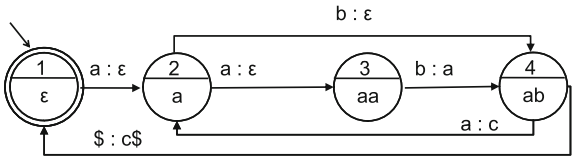


Fig. 2. Partial DFST for $s^{+}_{(ab|bb)\to c}$

Intuitively, the above algorithm simulates the left-most matching. It buffers the current string processed so far, and the buffer size is the length bound of r. Once the buffer is full (case 2), it examines the buffer and checks if there is a match. If not, it emits the first character and produces it as output; otherwise, it produces ω_2 on the output tape. The bijection $\mathcal{B}$ is feasible because of the bounded length of r. Note that the \$ (appended to ω and ω_1) is necessary in the lemma. It is used to terminate the DFST.

Example 4.9 Figure 2 presents a part of the DFST for $s^{+}_{(ab|bb)\to c}$. Note that the length bound of regular pattern is 2. For input word $aabab\$$, the only matching output word is $acc\$$, as recognized by the DFST (visiting states 1,2,3,4,2,4,1).

Each state in the DFST "stores" the current substring processed so far (up to the length bound). For example, in Fig. 2, state 3 stores aa (which has just reached the length bound). When it sees b, since there is no match of the search pattern from the beginning, following case 2 in the proof, it emits the first character a on the output tape and the buffered string becomes ab. Now at state 4, if the next input character is a, the buffered string becomes aba where ab matches the search pattern. Then the rule to follow is the second branch of case 2: generate the replacement c on the output tape and buffers the rest of string (i.e., a). This is represented by the transition from state 4 to 2. □

It is then interesting to know if the general problem of regular substitution can be represented using DFST. The conclusion is false, as shown by the following lemma.

Lemma 4.10 *There exists a regular expression $r \in R$ and a word $\mu \in \Sigma^*$ s.t. for any DFST $\mathcal{A}$: $L(\mathcal{A}) \neq \{(\omega, \omega_1) \mid \omega_1 = \omega^{-}_{r\to\mu}\}$.*

Proof. We prove the lemma by contradiction. Let $r = a^+b$ and $\mu = c$, where $a, b, c \in \Sigma$.

Assume there exists a DFST $\mathcal{A}$ s.t. $L(\mathcal{A}) = \{(\omega, \omega_1) \mid \omega_1 = \omega^{-}_{a^+b\to c}\}$. It is known that $(a^n, a^n) \in L(\mathcal{A})$ and $(a^nb, c) \in L(\mathcal{A})$. Since $\mathcal{A}$ is deterministic, and a^n is a prefix of a^nb, it leads to the contradiction that a^n (as the result of $a^{n\,-}_{a^+b\to c}$) is a prefix of c (as the result of $a^nb^{-}_{a^+b\to c}$). Thus the assumption cannot be true. □

It is worthy of note that the same relation $\{(a^nb, c), (a^n, a^n)\}$ can be recognized by an NFST. Using the same technique, one can show that $\{(\omega, \omega\$) \mid \omega \in \Sigma^*\}$ is not accepted by any DFST. This answers the question on why \$ is needed as the string end marker in Lemma 4.8.

5. Procedural regular replacement

Modeling procedural replacement is more complex. The general idea is to compose a number of finite state transducers for generating the begin and end markers that denote all matches of a regular search pattern. Then another collection of transducers can be used to remove the markers that do not conform to the reluctant (greedy) semantics.

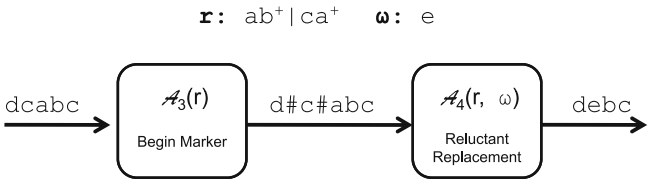


Fig. 3. Overview of reluctant replacement

We fix some notations first. A finite state automaton (FSA) is defined as a quintuple $(\Sigma, Q, q_0, F, \delta)$. Here Σ, Q, q_0, F are the alphabet, set of states, initial state, and set of final states, respectively. $\delta : Q \times \Sigma \to 2^Q$ is the transition function. Deterministic and nondeterministic FSA are denoted as DFSA and NFSA, respectively. $(q, a, q') \in \delta$ represents a transition[4] from state q to q' that is labeled with $a \in \Sigma$. Let $w = a_1 a_2 \dots a_n$, we say w has a *run from state* q to q', written as $q \rightsquigarrow_w^* q'$, if there exists $q_0, q_1, \dots, q_n \in Q$ s.t. $q = q_0$, and $q' = q_n$, and $\forall 1 \le i \le n : (q_{i-1}, a_i, q_i) \in \delta$. For any state q of a DFSA and any word $w \in \Sigma^*$, there is at most one $q' \in Q$ s.t. $q \rightsquigarrow_w^* q'$. The concept of run can be naturally extended to FST.

5.1. Modeling reluctant replacement

5.1.1. Overview

The FST model of reluctant replacement $S^-_{r\to\omega}$, as shown in Fig. 3, is a composition of two transducers: $\mathcal{A}_3(r)$ (begin marker transducer) and $\mathcal{A}_4(r, \omega)$ (reluctant replacement transducer). The subscript of each FST (e.g., 3 in $\mathcal{A}_3(r)$) indicates the step in the algorithm that produces the transducer.

Intuitively, an FST can be regarded as a computing device that accepts an input word, and produces one or more output words. Take the input word $dcabc$ in Fig. 3 as an example, it has two matches of pattern $ab^+ \mid ca^+$. They are ca (starting from the second character) and ab (starting from the third character). $\mathcal{A}_3(r)$ marks the beginning of each match and produces one and only one output word $d\#c\#abc$. Then the reluctant replacement transducer $\mathcal{A}_4(r, \omega)$ searches # from left to right. Whenever it discovers a #, $\mathcal{A}_4(r, \omega)$ enters the mode of substitution, which replaces the shortest match of r with ω (i.e., e in Fig. 3). This results in the final output $debc$. Notice that because the second # sits inside the match ca, it is filtered by the reluctant replacement transducer.

The design of the begin marker $\mathcal{A}_3(r)$ is interesting: the FST has some "look-ahead" capability due to the nondeterminism. At each position of the input word, it has to decide whether to insert a # sign (depending on if there is a subsequent substring matching r). $\mathcal{A}_3(r)$ makes sure that any run with an incorrect "look-ahead" decision will be rejected eventually. The construction of $\mathcal{A}_3(r)$ needs three steps, which are the first three contained in the road-map shown below. The entire algorithm consists of four steps:

- **Step 1:** Given $S^-_{r\to\omega}$, let $r' = \texttt{reverse}(r)$. Construct an end marking DFST $\mathcal{A}_1(r')$ which marks the end of any match of pattern r'.
- **Step 2:** $\mathcal{A}_1(r')$ is restricted in that its projection to the input tape accepts r' only. It is extended to $\mathcal{A}_2(r')$ so that any input word will be accepted (on its input tape). In the case there is no match of r', the output word is the same as input.
- **Step 3:** Reverse all transitions of $\mathcal{A}_2(r')$. This leads to the begin marker $\mathcal{A}_3(r)$ which marks the beginning of pattern r.
- **Step 4:** Replacement transducer $\mathcal{A}_4(r, \omega)$ is constructed by integrating several techniques such as enforcing the reluctant semantics. Then $\mathcal{M}^-_{r\to\omega} = \mathcal{A}_3(r) \,||\, \mathcal{A}_4(r, \omega)$ is the FST model of $S^-_{r\to\omega}$.

To distinguish the components of the various transducers in the algorithm, subscripts and superscripts are used in the notations for states and transitions. For example, Q_1 represents the set of states of $\mathcal{A}_1$, and q_5^2 is the fifth state in $\mathcal{A}_2$.

[4] Note that we write a transition of an FST as $(q, q', a : b)$ with the transition label $a : b$ intentionally placed as the third element, in order to distinguish it from a transition (q, a, q') of an FSA.

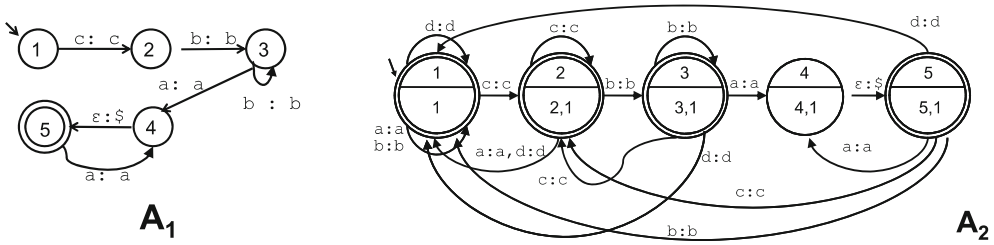


Fig. 4. Two examples of DFST end marker

5.1.2. Step 1 (End Marking DFST)

The objective of this step is to construct a DFST (letting it be $\mathcal{A}_1(r)$) that marks the end of of *every* match of a regular search pattern r.

Example 5.1 A_1 in Fig. 4 is the end marking DFST generated for a regular expression cb^+a^+. For example, ($cbaa$, $cba\$a\$$) is accepted by A_1. Notice that the \$ sign is appended to every possible match of the search pattern.

Construction Algorithm of $\mathcal{A}_1(r)$: Given a regular search pattern r, we first construct a deterministic FSA (letting it be DFSA(r)) that accepts r. Then we modify each final state f of DFSA(r) as below: (1) make f a non-final state, (2) create a new final state f' and establish a transition ϵ from f to f', (3) for any outgoing transition (f, a, s) from f, except (f, ϵ, f'), create a new transition (f', a, s) and remove (f, a, s) from f. Thus the ϵ transition is the only outgoing transition of f. Then convert the FSA into a DFST as below: for an ϵ transition, its output is \$ (end marker); for every other transition, its output is identical to the input symbol.

Take A_1 in Fig. 4 as an example. The f in the above algorithm is state 4 in A_1, and f' is state 5. In the DFSA for cb^+a^+, there is a self loop on state 4 (for modeling a^+), and now it is converted to the transition from state 5 to state 4 in A_1.

Lemma 5.2 and Corollary 5.3 follow directly from the construction algorithm.

Lemma 5.2 *$\mathcal{A}_1(r)$ is a DFST.*

Corollary 5.3 *For any state q_i in $\mathcal{A}_1(r)$, $\mu \in \Sigma^*$ and $\eta \in (\Sigma \cup \{\$\})^*$, there is at most one state q_j s.t. $q_i \rightsquigarrow^*_{(\mu,\eta)} q_j$ in $\mathcal{A}_1(r)$.*

Lemma 5.4 describes the major property of $\mathcal{A}_1(r)$. Notice that condition (3) specifies that $\mathcal{A}_1(r)$ never produces consecutive \$ signs on its output tape. The proof of Lemma 5.4 is available in Appendix B.1.

Lemma 5.4 *For any $r \in R$, $\mu \in \Sigma^*$ and $\eta \in (\Sigma \cup \{\$\})^*$, $(\mu, \eta) \in L(\mathcal{A}_1(r))$ iff all the following are satisfied:*

1. *$\mu \in L(r)$, $|\eta| > 0$, and $\eta[|\eta| - 1] = \$$; and,*
2. *$\mu = \pi_\Sigma(\eta)$; and,*
3. *for each $0 \leq i < |\eta|$: $\eta[i] = \$$ iff (i) $\pi_\Sigma(\eta[0, i]) \in L(r)$, and (ii) when $i > 0$, $\eta[i-1] \neq \$$.*

5.1.3. Step 2 (Generic End Marker)

$\mathcal{A}_1(r)$ has one limitation: for any (μ, η) accepted by $\mathcal{A}_1(r)$, μ has to be contained in $L(r)$. This does not work for the general case, as the entire input word will unlikely be an instance of the search pattern. We would like to generalize the transducer so that the new FST (called $\mathcal{A}_2(r)$) will accept any word on its input tape (and it marks the end of any match of r using \$ on its output tape). We call it the generic end marker for r.

Example 5.5 Let $\Sigma = \{a, b, c, d\}$ and the regular search pattern r be cb^+a^+. A_2 in Fig. 4 is the generic end marker for r (and Σ). For example, ($ccbbaa$, $ccbba\$a\$$) is accepted by A_2. Note that the input word, $ccbbaa$, is not an instance of cb^+a^+, and the match starts from the second c character. After each match of the search pattern, a \$ is appended. For another example, ($dcbaa$, $dcba\$a\$$) $\in L(A_2)$.

Definition 5.6 Given $r \in R$ and $\zeta \in (\Sigma \cup \{\$\})^*$, ζ is said to be *r-end-marked* if both of the following two conditions are satisfied:

- (D1) For any $0 \leq x < |\zeta|$, $\zeta[x] = \$$ *iff* (1) $\pi_\Sigma(\zeta[0, x]) \in \Sigma^* r$;[5] and (2) $\zeta[x-1] \neq \$$ or $x = 0$.
- (D2) If $\pi_\Sigma(\zeta) \in \Sigma^* r$ then $|\zeta| > 0$ and $\zeta[|\zeta| - 1] = \$$. □

For any word $\mu \in \Sigma^*$, ζ is called the r-end-marked output of μ, if $\pi_\Sigma(\zeta) = \mu$ and ζ is r-end-marked.

Lemma 5.7 *For any $r \in R$ and $\mu \in \Sigma^*$ there exists one and only one r-end-marked output of μ.*

The proof of Lemma 5.7 is presented in Appendix B.2. It relies on the fact that no duplicate \$ signs exist in a r-end-marked output.

Construction Algorithm of $\mathcal{A}_2(r)$: The generic end marker $\mathcal{A}_2(r)$ works by tracing the running of a word pair (μ, η) on $\mathcal{A}_1(r)$. It monitors states in $\mathcal{A}_1$ that could be reached by any suffix of (μ, η). This is accomplished using a bijection function $\mathcal{B}$ from the set of states of $\mathcal{A}_2(r)$ to the power-set of the set of states of $\mathcal{A}_1(r)$.

The construction process is formally presented as follows. Given $\mathcal{A}_1(r) = (\Sigma \cup \{\$\}, Q_1, q_0^1, F_1, \delta_1)$ as described in Step 1, $\mathcal{A}_2(r)$ is constructed as a quintuple $(\Sigma \cup \{\$\}, Q_2, q_0^2, F_2, \delta_2)$. We make Q_2 a set of $|2^{Q_1}|$ states. A *labeling function* $\mathcal{B} : Q_2 \to 2^{Q_1}$ is defined as a bijection s.t. $\mathcal{B}(q_0^2) = \{q_0^1\}$. Let F_0 be the set of final states of DFSA(r), from which $\mathcal{A}_1(r)$ is constructed from. The set of final states of $\mathcal{A}_2(r)$ is defined as: $F_2 = \{q \,|\, \mathcal{B}(q) \cap F_0 = \emptyset\}$.

The transition relation of $\mathcal{A}_2(r)$ is constructed using two case rules that define the out-going transitions for each state. Let $t \in Q_2$:

1. *Case 1 Rule*: If $t \in F_2$, i.e., $\mathcal{B}(t) \cap F_0 = \emptyset$, then for any $a \in \Sigma$: $(t, t', a : a) \in \delta_2$ iff $\mathcal{B}(t') = \{s' \mid \exists s \in \mathcal{B}(t) s.t. (s, s', a : a) \in \delta_1\} \cup \{q_0^1\}$. Notice that for each $a \in \Sigma$, there exists such a $t' \in Q_2$ because $\mathcal{B}(t')$ includes q_0^1 (the initial state of $\mathcal{A}_1(r)$). This makes $\mathcal{A}_2(r)$ to accept any word on its input tape.
2. *Case 2 Rule*: If $t \notin F_2$, i.e., $\mathcal{B}(t) \cap F_0 \neq \emptyset$, t has exactly one transition of the form $(t, t', \epsilon : \$) \in \delta_2$ where $t' = \mathcal{B}^{-1}((\mathcal{B}(t) - F_0) \cup \{q' \mid \exists\, q \in \mathcal{B}(t) : (q, q', \epsilon : \$) \in \delta_1\})$. Note that $\mathcal{B}(t') \cap F_0 = \emptyset$.

Example 5.8 A_2 in Fig. 4 is the result of applying the above algorithm on A_1. Each state in A_2 is mapped to a subset of states in A_1. For example, $\mathcal{B}(1) = \{1\}$, $\mathcal{B}(2) = \{2, 1\}$, and $\mathcal{B}(3) = \{3, 1\}$.

Running $(ccbb, ccbb)$ on A_2 results in a partial run to state 3. For $(ccbb, ccbb)$, $\mathcal{B}(3)$ tracks all possible states in A_1 that could be reached by the suffixes of its input word, and they are $ccbb$, cbb, bb, b, and ϵ. Among them, only cbb and ϵ can be extended to match r (i.e., cb^+a^+). In another word, if we run these input words (and their paired output words) on A_1, they would result in partial runs that end at states 3 (by (cbb, cbb)) and 1 (by (ϵ, ϵ)).[6] This is the intuition of having $\mathcal{B}(3) = \{3, 1\}$ in A_2.

Next we give one example on developing the transition relation for A_2. By applying Case Rule 1 on state 3 of A_2, we have $(3, 4, a : a) \in \delta_2$ and $\mathcal{B}(4) = \{4, 1\}$. This is because after appending (a, a) to (cbb, cbb), the new word pair $(cbba, cbba)$ reaches state 4 in A_1. The new suffix (ϵ, ϵ) always vacuously leads to state 1 in A_1. For all other suffixes being tracked, appending (a, a) leads to nowhere in A_1. Thus, state 4 in A_2 is mapped by $\mathcal{B}$ to $\{4, 1\}$ in A_1.

Finally, state 4 in A_2 is not a final state because $\mathcal{B}(4) \cap F_0$ is not empty. Readers can also verify that the transition from state 4 to 5 in A_2 is the result of applying the Case 2 Rule. □

Lemma 5.9 *For any $r \in R$, $\mathcal{A}_2(r)$ is a DFST.*

We formalize the properties of $\mathcal{A}_2(r)$ using a collection of lemmas. In particular, Lemma 5.10 states that $\mathcal{A}_2(r)$ simulates $\mathcal{A}_1$ on running suffixes of input words. Lemma 5.11 describes $\mathcal{A}_2(r)$'s function as a generic end marker. The proofs are available in Appendix B.3 and Appendix B.4.

[5] Note that $\zeta[0, x]$'s last element is $\zeta[x-1]$ (index starting from 0).

[6] Other input words, i.e., $ccbb$ and bb, make A_1 stuck and do not reach any states in A_1.

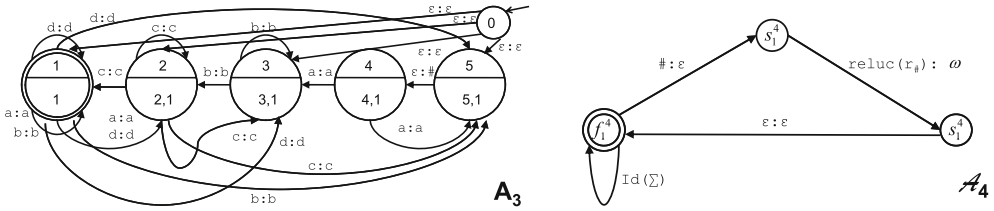


Fig. 5. Begin marker and reluctant replacement transducers

Lemma 5.10 *For any $r \in R$, let $\mathcal{A}_1(r)$ be $(\Sigma \cup \{\$\}, Q_1, q_0^1, F_1, \delta_1)$, and $\mathcal{A}_2(r)$ be $(\Sigma \cup \{\$\}, Q_2, q_0^2, F_2, \delta_2)$. For any $t \in Q_2$, any $\mu \in \Sigma^*$, and any $\eta \in (\sigma \cup \{\$\})^*$ s.t. $q_0^2 \rightsquigarrow^*_{(\mu,\eta)} t$ in $\mathcal{A}_2(r)$, both of the following are true:*[7]

1. *For each $q_i^1 \in \mathcal{B}(t)$, there exists $\nu \dot{\prec} \mu$ and $\zeta \in (\Sigma \cup \{\$\})^*$ s.t. $q_0^1 \rightsquigarrow^*_{(\nu,\zeta)} q_i^1$ in $\mathcal{A}_1(r)$.*
2. *For each $\nu \dot{\prec} \mu$ s.t. $\nu \in PREFIX(r)$, there exists $\zeta \in (\Sigma \cup \{\$\})^*$ and $q_i^1 \in \mathcal{B}(t)$ s.t. $q_0^1 \rightsquigarrow^*_{(\nu,\zeta)} q_i^1$ in $\mathcal{A}_1(r)$.*

Lemma 5.11 *For any $r \in R$, any $\mu \in \Sigma^*$, and any $\eta \in (\Sigma \cup \{\$\})^*$: $(\mu, \eta) \in L(\mathcal{A}_2(r))$ iff η is the r-end-marked output of μ.*

Corollary 5.12 For any $r \in R$, and any $\mu \in \Sigma^*$, there exists one and only one $\eta \in (\Sigma \cup \{\$\})^*$ s.t. $(\mu, \eta) \in L(\mathcal{A}_2(r))$.

5.1.4. Step 3 (Begin Marker of Regular Pattern)

From $\mathcal{A}_2(r)$ it is straightforward to construct a "reversed" transducer which marks the beginning of any match of the search pattern $\texttt{reverse}(r)$ in an input word.

Example 5.13 A_3 shown in Fig. 5 is a reverse of A_2 in Fig. 4. A_3 marks the beginning for pattern $r = a^+b^+c$ in any input word on alphabet $\{a, b, c, d\}$. For example, $(aabbcc, \#a\#abbcc) \in L(A_3)$.

Construction Algorithm of $\mathcal{A}_3(r)$: Let $\mathcal{A}_2(\texttt{reverse}(r))$ be a quintuple $(\Sigma \cup \{\$\}, Q_2, q_0^2, F_2, \delta_2)$. $\mathcal{A}_3(r)$ is defined as $(\Sigma \cup \{\#\}, Q_3, q_0^3, F_3, \delta_3)$, where $Q_3 = Q_2 \cup \{q_0^3\}$ and $F_3 = \{q_0^2\}$. δ_3 is the *minimal set of transitions* derived from δ_2, satisfying the following three rules:

1. For any $(t, t', a : a) \in \delta_2$: $(t', t, a : a)$ is contained in δ_3.
2. For any $(t, t', \epsilon : \$) \in \delta_2$: $(t', t, \epsilon : \#)$ is contained in δ_3.
3. For any $t \in F_2$: there exists $(q_0^3, t, \epsilon : \epsilon)$ in δ_3.

Definition 5.14 Given $r \in R$ and $\zeta \in (\Sigma \cup \{\#\})^*$, ζ is said to be r-begin-marked if both of the following two conditions are satisfied:

- (E1) For any $0 \leq x <|\zeta|$: $\zeta[x] = \#$ iff (1) $\pi_\Sigma(\zeta[x+1, |\zeta|]) \in r\Sigma^*$;[8] and (2) $\zeta[x+1] \neq \#$ or $x = |\zeta| - 1$.
- (E2) If $\pi_\Sigma(\zeta) \in r\Sigma^*$ then $\zeta[0] = \#$.[9]

Given $\omega \in \Sigma^*$, a word $\zeta \in (\Sigma \cup \{\#\})^*$ is said to be the r-begin-marked output of ω if $\pi_\Sigma(\zeta) = \omega$ and ζ is r-begin-marked.

Lemma 5.15 *For any $r \in R$ there exists an FST $\mathcal{A}_3(r)$ s.t. for any $\mu \in \Sigma^*$ and $\eta \in (\Sigma \cup \{\#\})^*$: $(\mu, \eta) \in L(\mathcal{A}_3(r))$ iff η is the r-begin-marked output of μ.*

Notice that $\mathcal{A}_3(r)$ is nondeterministic. However, due to the fact that $\mathcal{A}_2(r)$ is a DFST, $\mathcal{A}_3(r)$ can always make the "smart" decision to enforce that there is one and only one run which "correctly" inserts the label #. Any incorrect insertion will never reach a final state. The nondeterminism gives $\mathcal{A}_3(r)$ the "look ahead" ability. Based on Lemma 5.7, we have the following.

---

[7] Notice that the combination of the two conditions is not equivalent to the statement that $\mathcal{B}(t)$ is always equal to $\{q_i^1 \mid \exists \nu \dot{\prec} \mu, \zeta \in (\Sigma \cup \{\$\})^* s.t.\ q_0^1 \rightsquigarrow^*_{(\nu,\zeta)} q_i^1\}$.

[8] Note that $\zeta[x, |\zeta|]$ is the suffix of ζ starting from index x, with right bound $|\zeta|$ (not included). For $\zeta[|\zeta|, |\zeta|]$, it generates ϵ.

[9] E2 is not subsumed by E1. For example let $r = a^+$, word $a\#a$ satisfies E1 but not E2.

Lemma 5.16 *For any $r \in R$ and $\mu, \eta, \eta' \in \Sigma^*$: if $(\mu, \eta) \in L(\mathcal{A}_3(r))$ and $(\mu, \eta') \in L(\mathcal{A}_3(r))$, then $\eta = \eta'$.*

5.1.5. Step 4 (Reluctant Replacement)

The purpose of the last transducer is to perform the "reluctant replacement". The transducer (letting it be $\mathcal{A}_4(r, \omega)$) has to accomplish three tasks: (1) filtering extra # symbols, (2) enforcing reluctant semantics, and (3) performing procedural replacement. We address them one by one.

Task 1: Filtering Extra # Symbols.

Example 5.17 To see the needs of the filtering operation, consider the following example where $r \in R$ and $\mu, \omega \in \Sigma^*$. Let $r = ca \mid aa$, $\mu = caa$, and $\omega = d$. The r-begin-marked output of μ is $\#c\#aa$. Reluctant replacement $\mu^{-}_{r \to \omega}$ results in da where only the first match of r, i.e., ca is replaced, because the second match of r starts inside ca. In this example, only the first # is useful for identifying a match of r, and the second # is extra and should be filtered.

Definition 5.18 Let L be a regular language on Σ, and $x \notin \Sigma$ be a special symbol. L_x, the interspersed language of L with x, is defined as $L_x = \{\omega \mid \omega \in (\Sigma \cup \{x\})^* \text{and} \, \pi_\Sigma(\omega) \in L\}$. □

Task 2: Reluctant Semantics. Given a regular expression r, we write $L(r)_\#$ as $r_\#$ for simplicity. $L(r)_\# \times \{\omega\}$ (written as $r_\# \times \omega$ for short) defines a regular relation $\{(\mu, \eta) \mid \mu \in L(r)_\# \text{and} \, \eta = \omega\}$. It accomplishes the replacement with filtering, but without enforcing the reluctant semantics. We need an additional tool shown below.

Definition 5.19 Let $r \in R$, its reluctant version, $\texttt{reluc}(r)$ is defined as $\{\omega \mid \omega \in L(r)$, and there does not exist $\eta \prec \omega$ s.t. $\eta \neq \omega$ and $\eta \in L(r)\}$.

Lemma 5.20 *For any $r \in R$, $\texttt{reluc}(r)$ is a regular language.*

Proof. Let $\mathcal{A}$ be the DFSA that accepts r. A DFSA accepts $\texttt{reluc}(r)$ (letting it be $\mathcal{A}'$) can be constructed by removing all out-going transitions from each final state of $\mathcal{A}$. Any $\omega \in L(\mathcal{A}')$ is contained in $\texttt{reluc}(r)$ because its run does not travel through more than one final state. Any $\eta \in \texttt{reluc}(r)$ is accepted by $\mathcal{A}'$ because its run on $\mathcal{A}$ is also a run on $\mathcal{A}'$. □

Notice that the use of $\texttt{reluc}(r)$ does not help find the "shortest" instance of r, but the "reluctant" matches. To see the difference, consider $r = a^+ \mid cda^+$. The shortest instance of r is a. But in the process of reluctant replacement, given input word $ecdabc$, cda is the "reluctant match", because starting at position 1, it is the "shortest" match.

Task 3: Procedural Replacement and Construction Algorithm of $\mathcal{A}_4(r, w)$. Given $r \in R$ and $\omega \in \Sigma^*$, we now construct a transducer $\mathcal{A}_4(r, \omega)$, which given a r-begin-marked word on its input tape, generates the result of reluctant replacement on its output tape.

The design of $\mathcal{A}_4(r, \omega)$ is shown on the right half of Fig. 5. Intuitively, the transducer repeatedly consumes a symbol on both the input tape and output tape unless encountering a begin marker #. Once a # is consumed on the input tape, $\mathcal{A}_4(r, \omega)$ enters the replacement mode, which replaces the reluctant match of r with ω (and also removes extra # in the match). Piping with $\mathcal{A}_3(r)$ leads to the precise modeling of reluctant replacement.

Definition 5.21 Given $r \in R$ and $\omega \in \Sigma^*$, AFST $\mathcal{A}_4(r, \omega)$ is defined as a quintuple $(\Sigma \cup \{\#\}, Q_4, f_1^4, F_4, \delta_4)$ where $Q_4 = \{f_1^4, s_1^4, s_2^4\}$, $F_4 = \{f_1^4\}$, and δ_4 consists of the following four transitions: $\tau_1 : (f_1^4, f_1^4, Id(\Sigma))$, and $\tau_2 : (f_1^4, s_1^4, \# : \epsilon)$, and $\tau_3 : (s_1^4, s_2^4, \texttt{reluc}(r_\#) : \omega)$, and $\tau_4 : (s_2^4, f_1^4, \epsilon : \epsilon)$. □

We now prove that $\mathcal{A}_4(r, \omega)$ performs the reluctant replacement on a r-begin-marked word (when $\epsilon \notin r$). Lemma A.5 in Appendix A describes the $\epsilon \in r$ case.

Lemma 5.22 *Given $r \in R$ with $\epsilon \notin r$, and $\omega \in \Sigma^*$, for any r-begin-marked word $\kappa \in (\Sigma \cup \{\#\})^*$ and word $\zeta \in \Sigma^*$: $(\kappa, \zeta) \in L(\mathcal{A}_4(r, \omega))$ iff $\kappa = \zeta \in \Sigma^* - \Sigma^* r \Sigma^*$ or both of the following are true:*

- (F1) *κ can be written as $\nu \# \beta \mu$ such that, $\nu \in \Sigma^* - \Sigma^* r \Sigma^*$, and $\beta \in r_\#$, and for every x, y, u, t, m, n with $\nu = xy$, $\beta = ut$, and $\mu = mn$: if $y \neq \epsilon$ then $yu \notin r_\#$ and $y\beta m \notin r_\#$; and if $t \neq \epsilon$ then $u \notin r_\#$.*[10]
- (F2) *ζ can be written as $\nu\omega\eta$, and (μ, η) is accepted by $\mathcal{A}_4(r, \omega)$.*

Proof. The base case where $\kappa = \zeta$ and $\kappa \in \Sigma^* - \Sigma^* r \Sigma^*$ holds because the run of (κ, ζ) on $\mathcal{A}_4(r, \omega))$ is a self loop of the transition $(f_1^4, f_1^4, Id(\Sigma))$. We now concentrate on the case where state s_1^4 is visited at least once. Notice that the fact κ is r-begin-marked is essential in the proof.

(*Direction* $\Rightarrow$:) We first prove that if $(\kappa, \zeta) \in L(\mathcal{A}_4(r, \omega))$ then κ and ζ satisfy F1 and F2. Since we assume s_1^4 is visited at least once, let a run γ of (κ, ζ) on $\mathcal{A}_4(r, \omega)$ be written as follows:

$$f_1^4 \rightsquigarrow^*_{(\nu,\nu)} f_1^4 \rightsquigarrow_{(\#,\epsilon)} s_1^4 \rightsquigarrow_{(\beta,\omega)} s_2^4 \rightsquigarrow_{(\epsilon,\epsilon)} f_1^4 \rightsquigarrow^*_{(\mu,\eta)} f_1^4$$

In the above "presentation" of γ, we require that the "#" before s_1^4 and the s_2^4 before (ϵ, ϵ) are both their *first* occurrence in γ. Then we show that the β, μ, η resulted from this writing of γ satisfy F1 and F2.

From the structure of $\mathcal{A}_4(r, \omega)$, it follows that $\nu \in \Sigma^*$ and $\beta \in \texttt{reluc}(r_\#)$. We need to further prove that $\nu \in \Sigma^* - \Sigma^* r \Sigma^*$, this is available by inferring from the fact that κ is r-begin-marked (otherwise, there would be a # inside ν). For F2, $(\mu, \eta) \in L(\mathcal{A}_4(r, \omega))$ holds because the last segment of γ, i.e., $f_1^4 \rightsquigarrow^*_{(\mu,\eta)} f_1^4$, is an acceptance run.

The only proposition left to prove is: for any x, y, u, t, m, n s.t. $\nu = xy$, $\beta = ut$, and $\mu = mn$: (G1) if $y \neq \epsilon$ then $yu \notin r_\#$ and $y\beta m \notin r_\#$; and, (G2) if $t \neq \epsilon$ then $u \notin r_\#$. From γ, we can infer that $\beta \in \texttt{reluc}(r_\#)$, which leads to G2. G1 is proved using the fact that κ is r-begin-marked, and hence there does not exist an earlier match.

(*Direction* $\Leftarrow$:) It is straightforward to prove that if κ and ζ satisfy F1 and F2, then $(\kappa, \zeta) \in L(\mathcal{A}_4(r, \omega))$. The run of (κ, ζ) on $\mathcal{A}_4(r)$ is constructed using induction. □

Lemmas 5.22 and A.5 (the $\epsilon \in r$ case of Lemma 5.22, given in Appendix A) lead to Theorem 5.23. The complete proof can be found in Appendix B.5.

Theorem 5.23 *Given any $r \in R$ and $\omega \in \Sigma^*$, and let $\mathcal{M}^-_{r\to\omega}$ be $\mathcal{A}_3(r) \,||\, \mathcal{A}_4(r, \omega)$, then for any $\kappa, \zeta \in \Sigma^*$: $(\kappa, \zeta) \in L(\mathcal{M}^-_{r\to\omega})$ iff $\zeta = \kappa^-_{r\to\omega}$.*

5.2. Modeling greedy replacement

5.2.1. Overview

Modeling greedy replacement $S^+_{r\to\omega}$ involves composition of seven transducers as shown in the following. The purpose is to properly mark each greedy ("longest") match of r using a pair of # (begin marker) and $ (end marker). Then a replacement transducer can take the input and perform the replacement.

1. Begin marker transducer $\mathcal{A}_3(r)$. It inserts a # before each match of r.
2. Nondeterministic end marker transducer $\mathcal{A}_5(r)$. It *nondeterministically* inserts a $ after each match of r.
3. Pairing filter $\mathcal{A}_6$. It makes sure that each # is paired with a corresponding $.
4. Match filter $\mathcal{A}_7(r)$. It checks if the substring between each pair of # and $ markers is a match of r.
5. Begin marker protector $\mathcal{A}_8(r)$. It avoids incorrect removing of begin markers by $\mathcal{A}_6$.
6. Longest match filter $\mathcal{A}_9(r)$. It ensures that each marked match of r is the longest one starting from a given position in the input word.
7. Replacement transducer $\mathcal{A}_{10}(r, \omega)$, which performs the replacement. It enters the replacement mode once encountering a #, and completes the replacement when seeing a $.

It is important that these transducers are "chained" in the order specified as above.

[10] Note that by the last condition, $\beta \in \texttt{reluc}(r_\#)$.

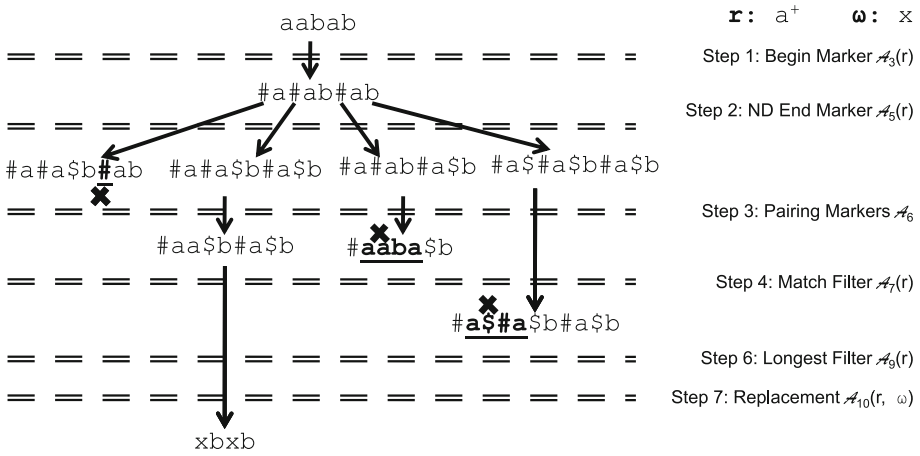


Fig. 6. An example of greedy replacement

Example 5.24 Figure 6 displays the process of applying the seven transducers on $S^+_{r\to\omega}$ where $S = aabab$, $r = a^+$, and $\omega = x$. In this example, each transducer is regarded as an input/output device.

- *Step 1:* begin marker transducer $\mathcal{A}_3(r)$ inserts a # before each match of r, and it yields output word $\#a\#ab\#ab$.
- *Step 2:* Nondeterministic end marker $\mathcal{A}_5(r)$ produces multiple output words. Each \$ sign is guaranteed to be preceded by a match of r; however, the inverse is not true for each match of r. For example, in the output word $\#a\#ab\#a\$b$ only the third match is ended with \$.
- *Step 3:* Pairing filter $\mathcal{A}_6$ tries to filter extra # signs, and pair # and \$ markers. If the effort fails, the input word will be rejected. For example, given input word $\#a\#ab\#a\$b$, $\mathcal{A}_6$ yields output word $\#aaba\$b$ – all the # signs between the first # and \$ are filtered. For another example, input word $\#a\#a\$b\#ab$ is rejected because there is no way to find a matching \$ for the last #. In summary, all the output words of $\mathcal{A}_6$ have each # paired with a \$, i.e., projecting the output word to the alphabet of {#, \$} always results in a word in $(\#\$)^*$.
- *Step 4:* Match filter $\mathcal{A}_7(r)$ rejects those input words where there exists a substring not in r between a pair of begin and markers. For example, $\#aaba\$b$ is rejected at this step.
- *Step 5:* Begin marker protector $\mathcal{A}_8(r)$ has no application here. It is used when r contain $*$ operators. We will show one example later.
- *Step 6:* Longest match filter $\mathcal{A}_9(r)$ rejects those input words that do not conform to the greedy semantics. For example, $\#a\$\#a\$b\#a\$b$ is rejected because there could be a longer match aa starting from the first #.
- *Step 7:* Replacement transducer $\mathcal{A}_{10}(r, \omega)$ replaces each substring inside a pair of # and \$ with ω. For example, given input word $\#aa\$b\#a\$b$, it generates $xbxb$. Note that $xbxb$ is the one and only one output word of $aabab$, generated by the composition of the seven transducers.

In the following, we present the details of the seven transducers.

5.2.2. *Step 1 (Begin Marker)*

The begin marker $\mathcal{A}_3(r)$ is defined in Sect. 5.1.4.

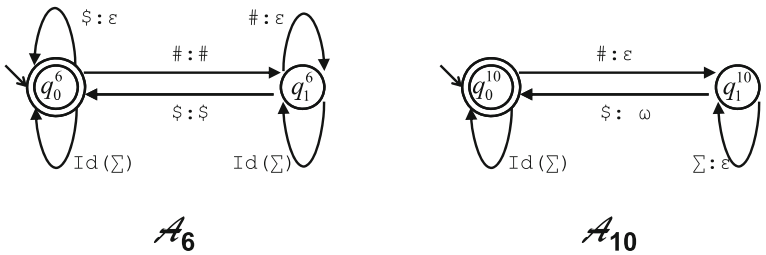


Fig. 7. Design of $\mathcal{A}_6$ and $\mathcal{A}_{10}(r, \omega)$

5.2.3. *Step 2 (Nondeterministic End Marker)*

The construction algorithm of the nondeterministic end marker (letting it be $\mathcal{A}_5(r)$) is very similar to that of $\mathcal{A}_2(r)$ (the generic end marker in Sect. 5.1.3). There are two differences: (1) the \$ after a match of r is *optional*; and (2) the input word now contains # (introduced by $\mathcal{A}_3(r)$ in Step 1). (1) is handled by adding an $(\epsilon : \epsilon)$ transition in the Case 2 rule (adapted from $\mathcal{A}_2(r)$). (2) is solved by adding (# : #) transitions to each state. The details are given below.

Construction Algorithm of $\mathcal{A}_5(r)$: Given $\mathcal{A}_1(r) = (\Sigma \cup \{\$\}, Q_1, q_0^1, F_1, \delta_1)$ as described in Sect. 5.1.2, $\mathcal{A}_5(r)$ is a quintuple $(\Sigma_2, Q_5, q_0^5, F_5, \delta_5)$. Q_5 is a set of $|2^{Q_1}|$ states, and similarly there is a bijection $\mathcal{B} : Q_5 \to 2^{Q_1}$ s.t. $\mathcal{B}(q_0^5) = \{q_0^1\}$. $F_5 = \{q \mid \mathcal{B}(q) \cap F_0 = \emptyset\}$. δ_5 is the minimum set of transition rules constructed using the following three derivation rules for each $t \in Q_5$:

1. *Case 1 Rule:* If $t \in F_5$, i.e., $\mathcal{B}(t) \cap F_0 = \emptyset$, then for any $a \in \Sigma$: $(t, t', a : a) \in \delta_5$ iff $\mathcal{B}(t') = \{s' \mid \exists s \in \mathcal{B}(t)\ s.t.\ (s, s', a : a) \in \delta_1\} \cup \{q_0^1\}$.
2. *Case 2 Rule:* If $t \notin F_5$, i.e., $\mathcal{B}(t) \cap F_0 \neq \emptyset$, t has exactly *two* transitions: $(t, t', \epsilon : \$)$ and $(t, t', \epsilon : \epsilon)$ where $t' = \mathcal{B}^{-1}((\mathcal{B}(t) - F_0) \cup \{q' \mid \exists q \in \mathcal{B}(t) : (q, q', \epsilon : \$) \in \delta_1\})$.
3. *Rule 3:* $(t, t, \# : \#) \in \delta_5$ for each state t in Q_5.

Lemma 5.25 *For any $r \in R$, any r-begin-marked word $\kappa \in (\Sigma \cup \{\#\})^*$, and $\zeta \in \Sigma_2^*$, $(\kappa, \zeta) \in L(\mathcal{A}_5(r))$ iff both of the following conditions are satisfied:*

- (I1) *for any $0 \leq x < |\zeta|$, $\zeta[x] = \$$ only if (1) $\pi_\Sigma(\zeta[0, x]) \in \Sigma^* r$; and (2) $\zeta[x-1] \neq \$$ or $x = 0$.*
- (I2) $\kappa = \pi_{\Sigma \cup \{\#\}}(\zeta)$.

Notice that $\Sigma_2 = \Sigma \cup \{\#, \$\}$. I1 in Lemma 5.25 is very similar to D1 in Definition 5.6 (for r-end-marked output), except that here **only if** (instead of **iff**) is used. The proof of Lemma 5.25 can be adapted from the proof of of Lemma 5.11.

5.2.4. *Step 3 (Pair Filter)*

Then we need a filter to remove extra markers so that every \$ is paired with a #. This is accomplished via a transducer $\mathcal{A}_6$ (note that $\mathcal{A}_6$ is not parameterized by r).

Construction Algorithm of $\mathcal{A}_6$: As shown in Fig. 7, $\mathcal{A}_6$ always swaps between two stages: (1) waiting for #, and (2) waiting for \$. While waiting for #, it removes any \$ encountered and generates an identical output character for each input symbol in Σ. While waiting for \$, it removes any # encountered. $\mathcal{A}_6$ ensures that there are equal numbers of begin and end markers in the output word, and they are paired. Note that in Step 3, we do not yet make sure that between a pair of # and \$, the substring inside is a match of r. This is accomplished later using additional filters.

Formally, $\mathcal{A}_6$ is defined using a quintuple $(\Sigma_2, Q_6, q_0^6, F_6, \delta_6)$ where $Q_6 = \{q_0^6, q_1^6\}$, $F = \{q_0^6\}$, and δ_6 consists of the following transitions: $(q_0^6, q_0^6, Id(\Sigma))$, and $(q_0^6, q_0^6, \$: \epsilon)$, and $(q_0^6, q_1^6, \# : \#)$, and $(q_1^6, q_1^6, \# : \epsilon)$, and $(q_1^6, q_1^6, Id(\Sigma))$, and $(q_1^6, q_0^6, \$: \$)$.

Example 5.26 We list four pairs of input and output words that are accepted by $\mathcal{A}_6$ in the following: ($\#ab\#c\$c\#ab\$$, $\#abc\$c\#ab\$$), and ($\#abc\$c\#a\#b\$$, $\#abc\$c\#ab\$$), and ($\#abcc\#a\#b\$$, $\#abccab\$$), and ($\#ab\#\$c\$c\#ab\$$, $\#ab\$cc\#ab\$$). The following are two examples of input words, pairing with any output words, will not be accepted by $\mathcal{A}_6$: $\#abc\$c\#ab\$\#$, and $abc\#c\#ab\$\#$.

Lemma 5.27 states that for each # in the output word of $\mathcal{A}_6$, there is a pairing \$, and there are no other markers between the pair.

Lemma 5.27 *$\mathcal{A}_6$ is a DFST. For any $\mu, \eta \in \Sigma_2^*$, if $(\mu, \eta) \in L(\mathcal{A}_6)$ then the following are true for η:*

1. $\pi_\Sigma(\mu) = \pi_\Sigma(\eta)$.
2. $\forall_i\, 0 \leq i <|\eta|: \eta[i] = \# \;\Rightarrow\; \exists_j\, i < j <|\eta|: \eta[j] = \$ \;\wedge\; \forall_k\, i < k < j : \eta[k] \in \Sigma$.
3. $\forall_i\, 0 \leq i <|\eta|: \eta[i] = \$ \;\Rightarrow\; \exists_j\, 0 \leq j < i : \eta[j] = \# \;\wedge\; \forall_k\, j < k < i : \eta[k] \in \Sigma$.

5.2.5. *Step 4 (Match Filter)*

Match filter makes sure that the substring between each pair of # and \$ is a match of r. It is formally defined in Definition 5.28. For regular languages, let $\cap$ and $\overline{r}$ represent the intersection and complement operators. In our following discussion (in Steps 4, 5, 6), whenever the complement operator is used, the alphabet of the regular language is $\Sigma_2 = \Sigma \cup \{\#, \$\}$ if not otherwise specified. We use regular expression and regular language interchangeably. □

Definition 5.28 Let $L_1 = \overline{\Sigma_2^*\#(\overline{r} \cap \Sigma^*)\$\Sigma_2^*}$. The match filter $\mathcal{A}_7(r)$ is an FST that accepts $Id(L_1)$.

Note that in the formula of L_1, $\overline{r}$ is defined as $\Sigma_2^* - r$. Thus, $\overline{r} \cap \Sigma^*$ is essentially $\Sigma^* - r$. Let DFSA(L_1) be the DFSA that accepts L_1. $\mathcal{A}_7(r)$ can be constructed from DFSA(L_1) by extending each transition (t, t', a) in DFSA(L_1) to the form of $(t, t', a : a)$.

Example 5.29 Let regular expression $r = ab^+ \mid b^+c$ and word $\alpha = ab\#bc\$\#bc\$$. α is accepted by L_1 because the substrings contained in the two #, \$ pairs are instances of r. Notice the use of $\overline{r} \cap \Sigma^*$ in the formula of L_1, it avoids treating the substring $bc\$\#bc$ of α (i.e., $\alpha[3, 9]$) as a substring between a pair of any (but not the closest neighboring) begin/end markers.[11] For another example, $aa\#aab\$dd\#bc\$ \notin L_1$ because aab (i.e., the substring inside the first pair of # and \$) is not a match of r.

5.2.6. *Step 5 (Begin Marker Protector)*

Occasionally $\mathcal{A}_6$ might treat "good" begin markers as the extra to filter. The begin marker protector $\mathcal{A}_8(r)$ is defined to avoid such cases. it is defined in Definition 5.30. We explain the design decision in Examples 5.31 and 5.32.

Definition 5.30 Let $L_2 = \overline{\Sigma_2^*[\hat{}\#](\$\Sigma\Sigma_2^* \cap r_{\#,\$}\Sigma_2^*)} \cap \overline{\Sigma_2^*[\hat{}\#](\$ \cap r_{\#,\$})}$. The begin marker protector $\mathcal{A}_8(r)$ is an FST that accepts $Id(L_2)$. □

L_2, as a conjunction of two regular languages (letting them be $L_2(1)$ and $L_2(2)$), handles two special cases: (1) to avoid removing begin markers for the next instance of r, in the middle of an input word; and, (2) to avoid removing the begin markers at the end of an input word if the pattern r includes ϵ.

Example 5.31 Let word $\alpha = bab$. The output of $\alpha^+_{a^* \to c}$ should be $cbccbc$, and the correct intermediate marking (i.e., the input word to the replacement transducer $\mathcal{A}_{10}(r, \omega)$) should be $\#\$b\#a\$\#\$b\#\$$.[12] Notice that there are four matches of a^*, and they are the ϵ before the first b in α, the first a, the ϵ before the second b, and the ϵ after the second b. We show that, without the proper protection provided by $L_2(1) = \overline{\Sigma_2^*[\hat{}\#](\$\Sigma\Sigma_2^* \cap r_{\#,\$}\Sigma_2^*)}$, an additional word $cbcbc$ (missing a c before the second b in the output word) might be generated.

[11] The claim assumes that $\mathcal{A}_7(r)$ is applied after $\mathcal{A}_6$, where all markers have been paired.

[12] Readers can verify the correctness of this argument using prevalent regex language packages, such as `java.util.regex`.

Consider a word $\beta = \#\$b\#a\$b\#\$$. It is an incorrect marking because the ϵ before the second b is not marked. However, it can be generated by the composition of the transducers of Steps 1, 2, 3, and 4. Details are shown below:

1. $(bab, \#b\#a\#b\#) \in L(\mathcal{A}_3(r))$ because $\mathcal{A}_3(r)$ has successfully identified the beginning position of all the four matches of a^*.
2. $(\#b\#a\#b\#, \#\$b\#a\#\$b\#\$) \in L(\mathcal{A}_5(r))$. $\#\$b\#a\#\$b\#\$$ is one of many available output words for the given input. This is because $\mathcal{A}_5(r)$ *nondeterministically* inserts \$ markers after the match of r. In this case, the \$ for the *second* # is not inserted.
3. $(\#\$b\#a\#\$b\#\$, \#\$b\#a\$b\#\$) \in L(\mathcal{A}_6) \cap L(\mathcal{A}_7(r))$. Now the interesting part is: the missing \$ for the second #, actually causes the third # being deleted by the pair filter $\mathcal{A}_6$! This incorrect marking can pass $\mathcal{A}_9(r)$ (the longest match filter), and when fed to $\mathcal{A}_{10}(r, \omega)$ later, produces $cbcbc$ as output.

Now we explain the design idea of $L_2(1)$. It is used to refute such a special "bad" pattern, i.e., a match of r is *not* preceded by a #, but by a \$ (which is used to mark the end of the previous match). To illustrate this point, we highlight this \$ sign in both the incorrect marking β (i.e., $\#\$b\#a\mathbf{\$}b\#\$$) and $L_2(1)$ (i.e., $\overline{\Sigma_2^*[\hat{}\#](\mathbf{\$}\Sigma\Sigma_2^* \cap r_{\#,\$}\Sigma_2^*)}$). Here r is a^* and its match is the ϵ before the second b in α. This ϵ is not preceded by the needed #. Note that in $L_2(1)$, the preceding and succeeding letters of the \$, i.e., $[\hat{}\#]$ and Σ are used to eliminate all possibilities of # preceding r. They match the highlighted a and b in $\#\$b\#\boldsymbol{a}\$\boldsymbol{b}\#\$$.

The major challenge of $L_2(1)$ is whether it can be an overkill? Would it be possible that the r in $L_2(1)$ does not have to be preceded by # at all? For example, this match of r could be inside another match which starts at an earlier position. All such cases can be eliminated. $L_2(1)$ is used in the Case 5.iii of the proof of Lemma 5.41.

Example 5.32 The greedy replacement $a^+_{a^*\to b}$ generates output bb because there are two matches of a^* in the input word, whereas $\#a\$\#\$$ is the proper marking. Without the protection of $L_2(2)$, it is possible to generate an incorrect marking $\#a\$$ by $\mathcal{A}_3(r)$ $||\mathcal{A}_5(r)||$ $\mathcal{A}_6$ $||$ $\mathcal{A}_7(r)$, which missed the second match of a^*.

$L_2(2) = \overline{\Sigma_2^*[\hat{}\#](\$ \cap r_{\#,\$})}$ refutes such a case by stating that: if $\epsilon \in r$ (this is implied by the $(\$ \cap r_{\#,\$})$ in the formula), then the intermediate marking (i.e., the input to $\mathcal{A}_{10}(r, \omega)$) should not be ended with a \$, which is preceded by a symbol that is not #.

The formal proof that uses $L_2(2)$ can be found in the proof of Lemma 5.41 (Case 2).

5.2.7. *Step 6 (Longest Match Filter)*

The longest match filter $\mathcal{A}_9(r)$ ensures that each substring between a marker pair is the longest match of r, starting from the begin marker. $\mathcal{A}_9(r)$ is effective based on the assumption that $\mathcal{A}_7(r)$ and $\mathcal{A}_8(r)$ have already been applied.

Definition 5.33 Let $L_3 = \overline{\Sigma_2^*\#(r_{\#,\$} \cap (\Sigma^*\$(\Sigma^+)_{\#,\$}))\Sigma_2^*}$. The longest match filter $\mathcal{A}_9(r)$ is an FST that accepts $Id(L_3)$. □

In L_3, the $r_{\#,\$}$ models a longer match of r which contains a shorter match. This is described by $(\Sigma^*\$(\Sigma^+)_{\#,\$})$.[13] Here the Σ^* before the \$ in $\Sigma^*\$(\Sigma^+)_{\#,\$}$ is the shorter match. The $(\Sigma^+)_{\#,\$}$ denotes the difference between the longer match and the shorter match, and it has at least one character in Σ. This is because $(\Sigma^+)_{\#,\$}$ refers to a word in Σ^+ interspersed with begin/end markers, i.e., for any $\omega \in (\Sigma^+)_{\#,\$}$, $|\ \pi(\omega)\ | > 0$.

In summary, L_3 rejects an improper marking \$ (highlighted in $\overline{\Sigma_2^*\#(r_{\#,\$} \cap (\Sigma^*\mathbf{\$}(\Sigma^+)_{\#,\$}))\Sigma_2^*}$). The \$ improperly marks a shorter match of r while another longer match exists and starts at the same #.

Example 5.34 Let regular expression $r = a^+$ and word $\beta = c\#a\$\#a\$$. β will be rejected by filter L_3. Here c in $\boldsymbol{c}\#a\$\#a\$$ matches the Σ_2^* in $\overline{\mathbf{\Sigma}_2^*\#(r_{\#,\$} \cap (\Sigma^*\$(\Sigma^+)_{\#,\$}))\Sigma_2^*}$. $a\$\#a\$$ is the longer match, and it corresponds to both the $r_{\#,\$}$ and $(\Sigma^*\$(\Sigma^+)_{\#,\$})$ in the formula. Here the improper \$ (marking a shorter match) is highlighted in both $a\mathbf{\$}\#a\$$ and $(\Sigma^*\mathbf{\$}(\Sigma^+)_{\#,\$})$. The difference between the longer and shorter match, i.e., $\#a\$$, corresponds to $(\Sigma^+)_{\#,\$}$ in the pattern.

[13] Here '(' and ')' are the grouping control characters in regular expression. They should not be treated as elements of Σ.

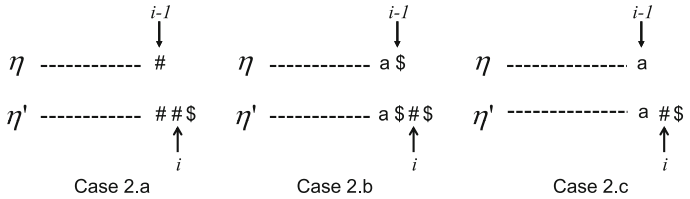


Fig. 8. Proof idea of Lemma 5.41 (Case 2)

5.2.8. Step 7 (Replacement Transducer)

The replacement transducer $\mathcal{A}_{10}(r, \omega)$, shown in Fig. 7, performs the substitution. It enters (and leaves) the replacement mode, once it sees a begin (end) marker, but it does not enforce the shortest match semantics. It assumes that the input word is already properly marked.

Definition 5.35 $\mathcal{A}_{10}(r, \omega)$ is a quintuple $(\Sigma_2, Q_{10}, q_0^{10}, F_{10}, \delta_{10})$. $Q_{10} = \{q_0^{10}, q_1^{10}\}$. $F_{10} = \{q_0^{10}\}$. δ_{10} has four transitions: $(q_0^{10}, q_0^{10}, Id(\Sigma))$, and $(q_0^{10}, q_1^{10}, \# : \epsilon)$, and $(q_1^{10}, q_1^{10}, \Sigma : \epsilon)$, and $(q_1^{10}, q_0^{10}, \$: \omega)$.

5.2.9. Correctness proof

We show that the composition of $\mathcal{A}_3(r) \,||\, \mathcal{A}_5(r) \,||\, \mathcal{A}_6 \,||\, \mathcal{A}_7(r) \,||\, \mathcal{A}_8(r) \,||\, \mathcal{A}_9(r)$ can correctly generate the proper greedy marking for any input word. Then feeding the marked word to $\mathcal{A}_{10}(r, \omega)$ yields the desired result.

Definition 5.36 The *greedy marking transducer* $\mathcal{M}_r^+$ is defined as $\mathcal{A}_3(r) \,||\, \mathcal{A}_5(r) \,||\, \mathcal{A}_6 \,||\, \mathcal{A}_7(r) \,||\, \mathcal{A}_8(r) \,||\, \mathcal{A}_9(r)$. The *procedural greedy replacement transducer* $\mathcal{M}_{r\to\omega}^+$ is $\mathcal{M}_r^+ \,||\, \mathcal{A}_{10}(r, \omega)$. □

Definition 5.37 Let $r \in R$ with $\epsilon \notin r$, for any $\kappa \in \Sigma^*$ and $\zeta \in \Sigma_2^*$, ζ is said to be an *r-greedy-marking* of κ iff one of the following two conditions is satisfied: □

1. $\kappa \in \Sigma^* - \Sigma^* r \Sigma^*$ and $\zeta = \kappa$; or
2. Let $\kappa = \nu\beta\mu$ such that, $\nu \notin \Sigma^* r \Sigma^*$, and $\beta \in r$, and for every x, y, u, t, m, n with $\nu = xy$, $\beta = ut$, and $\mu = mn$: if $y \neq \epsilon$ then $yu \notin r$ and if $m \neq \epsilon$ then $y\beta m \notin r$. Let η be an r-greedy-marking of μ. Then $\zeta = \nu\#\beta\$\eta$. □

Definition 5.37 is adapted from the formalization of $S_{r\to\omega}^+$ in Definition 3.2. Note that the $\epsilon \in r$ case is given in Definition A.8 in Appendix A. We list some properties about greedy marking in Lemma 5.38. The proof can be obtained by applying induction on the length of the input word. It is also not hard to infer Lemma 5.39 from the transition relation of $\mathcal{A}_{10}(r, \omega)$ and Definition 5.37.

Lemma 5.38 *For any $r \in R$ and any $\kappa \in \Sigma^*$, there exists one and only one r-greedy-marking of κ (letting it be ζ), and $\pi_\Sigma(\zeta) = \kappa$.*

Lemma 5.39 *For any $r \in R$ and any $\kappa, \zeta, \omega \in \Sigma^*$ and let η be the r-greedy-marking of κ, $\zeta = \kappa_{r\to\omega}^+$ iff $(\eta, \zeta) \in L(\mathcal{A}_{10}(r, \omega))$.*

Lemma 5.40 *For any $r \in R$, $\kappa \in \Sigma^*$ and $\zeta \in \Sigma_2^*$, if $(\kappa, \zeta) \in L(\mathcal{M}_r^+)$ then $\pi_\Sigma(\zeta) = \kappa$.*

Lemma 5.41 *For any $r \in R$, any $\kappa \in \Sigma^*$ and $\eta \in \Sigma_2^*$. η is the r-greedy-marking of κ iff $(\kappa, \eta) \in L(\mathcal{M}_r^+)$.*

Proof. We show one part of the proof and how $L_2(2)$ in $\mathcal{A}_8(r)$ is used. The complete proof is available in Appendix B.6.

(*Direction* $\Leftarrow$:) The proof goal is *if* $(\kappa, \eta) \in L(\mathcal{M}_r^+)$, *then* η *is the* r*-greedy-marking of* κ. It is assumed that the direction $\Rightarrow$ has been proved (see Appendix B.6). By Lemma 5.38 and direction $\Rightarrow$ we have: for any word $\alpha \in \Sigma^*$, there exists one and only one r-greedy marking (letting it be α'), and $(\alpha, \alpha') \in L(\mathcal{M}_r^+)$.

Based on the above, we only need to show the following proposition (J1) is true for accomplishing the proof goal:

(J1) If both (κ, η) *and* (κ, η') *are accepted by* $\mathcal{M}_r^+$, *then* $\eta = \eta'$.

Since $\mathcal{M}_r^+$ is a composition of a sequence of transducers, and each transducer can be regarded as an input/output device. One can write the process of reaching η and η' as the following, using subscripts to indicate the output of a particular transducer.

$$\xrightarrow{\kappa} \mathcal{A}_3(r) \xrightarrow{\eta_3} \mathcal{A}_5(r) \xrightarrow{\eta_5} \mathcal{A}_6 \xrightarrow{\eta_6} \mathcal{A}_7(r) \xrightarrow{\eta_7} \mathcal{A}_8(r) \xrightarrow{\eta_8} \mathcal{A}_9(r) \xrightarrow{\eta}$$
$$\xrightarrow{\kappa} \mathcal{A}_3(r) \xrightarrow{\eta'_3} \mathcal{A}_5(r) \xrightarrow{\eta'_5} \mathcal{A}_6 \xrightarrow{\eta'_6} \mathcal{A}_7(r) \xrightarrow{\eta'_7} \mathcal{A}_8(r) \xrightarrow{\eta'_8} \mathcal{A}_9(r) \xrightarrow{\eta'}$$

Based on Lemma 5.16, one can infer that $\eta_3 = \eta'_3$. Since $\mathcal{A}_7(r)$, $\mathcal{A}_8(r)$, and $\mathcal{A}_9(r)$ are "filters" (each of them accepts a language which is a subset of the identity relation). We have: $\eta_6 = \eta_7 = \eta_8 = \eta$, and $\eta'_6 = \eta'_7 = \eta'_8 = \eta'$. Thus, if $\eta \neq \eta'$, they have to depart right after $\mathcal{A}_5(r)$ (which nondeterministically introduces end markers).

We prove J1 by contradiction. Assume that $\eta \neq \eta'$. Let index i be the first index that η differs from η', i.e., $\eta[i] \neq \eta'[i]$ and $\forall_x\ 0 \leq x < i : \eta[x] = \eta'[x]$. Then there are seven cases to discuss (note that the length of η may be different from that of η'):

- **Case 1** (ϵ, a)**:** $|\eta| = i$ and $\eta'[i] \in \Sigma$ (letting it be a); or symmetrically $|\eta'| = i$ and $\eta[i] \in \Sigma$.
- **Case 2** $(\epsilon, \#)$**:** $|\eta| = i$ and $\eta'[i] = \#$; or symmetrically $|\eta'| = i$ and $\eta[i] = \#$.
- **Case 3** $(\epsilon, \$)$**:** $|\eta| = i$ and $\eta'[i] = \$$; or symmetrically $|\eta'| = i$ and $\eta[i] = \$$.
- **Case 4** (a, b)**:** $\eta[i] \in \Sigma$ and $\eta'[i] \in \Sigma$, however $\eta[i] \neq \eta'[i]$;
- **Case 5** $(b, \#)$**:** $\eta[i] \in \Sigma$ and $\eta'[i] = \#$; or symmetrically $\eta'[i] \in \Sigma$ and $\eta[i] = \#$.
- **Case 6** $(a, \$)$**:** $\eta[i] \in \Sigma$ and $\eta'[i] = \$$; or symmetrically $\eta'[i] \in \Sigma$ and $\eta[i] = \$$.
- **Case 7** $(\#, \$)$**:** $\eta[i] = \#$ and $\eta'[i] = \$$; or symmetrically $\eta'[i] = \#$ and $\eta[i] = \$$.

We now show that none of the above cases holds. Take Case 2 as an example.

Case 2: The goal is to prove by contradiction that the following claim (K) is false:

$$\textbf{(K)}: |\eta| = i \text{ and } \eta'[i] = \# \quad \text{and} \quad \eta[0, i] = \eta'[0, i].$$

Assume that (K) is true, one can reach the following:

(K1): $\epsilon \in r$, $|\eta'| = i + 2$, and $\eta'[i + 1] = \$$.

$\epsilon \in r$ follows from the following facts: (1) $\pi_\Sigma(\eta) = \pi_\Sigma(\eta')$ and $\pi_\Sigma(\eta) = \pi_\Sigma(\eta'[0, i])$, which implies that $\pi_\Sigma(\eta'[i, |\eta'|]) = \epsilon$; and (2) by Lemma 5.15, every # precedes a match of r. Thus, $\epsilon \in r$.

Since η' has gone through the pair filter $\mathcal{A}_6$ and the match filter $\mathcal{A}_7(r)$, it guarantees that there is a matching \$ at index $i + 1$ in η'. Because the non-deterministic end marker $\mathcal{A}_5(r)$ does not insert consecutive markers, the \$ must be the last element of η'. Therefore, we have $|\eta'| = i + 2$. This is basically to state that η' has two more characters #\$ than η.

Continuing from the above analysis, there are three cases to consider: (2.a) $\eta[i - 1] = \#$, (2.b) $\eta[i - 1] = \$$, and (2.c) $\eta[i - 1] \in \Sigma$. These three cases are presented in Fig. 8. We refute all of them in the following.

1. (Case 2.a) By assumption (K), we have $\eta'[i - 1] = \eta[i - 1] = \#$ (as shown in Fig. 8). This immediately contradicts with the fact that after passing $\mathcal{A}_6$, all the # and \$ markers should be paired.
2. (Case 2.b) This case assumes that $\eta[i - 1] = \eta'[i - 1] = \$$. Now consider $\eta[i - 2]$, it cannot be \$ or #, using an argument similar to 2.a. Now the only choice is $\eta'[i - 2] \in \Sigma$ (letting it be a). Thus η could be written as $\eta = \beta a\$$ where $\beta \in \Sigma_2^*$. Notice that η is an instance of $\Sigma_2^*[\hat{}\#](\$ \cap r_{\#,\$})$. This conflicts with filter $L_2(2)$ in $\mathcal{A}_8(r)$ (in another word, the ϵ at the end of η does not have a preceding #). In conclusion, Case 2.b cannot be true.

3. (Case 2.c) In this case, $\eta[i-1] = \eta'[i-1] \in \Sigma$ (letting it be a). In the following, we trace ("taint") the data processing that generates the last # in η', and let it be $\#_i$. We know that $\eta_3 = \eta'_3$, and thus $\#_i$ also occurs in η_3 but it disappears in η_6. The only way that it could disappear is to take the transition $(q_1^6, q_1^6, \# : \epsilon)$ in the pairing filter $\mathcal{A}_6$. Notice that q_1^6 is not a final state. To come back to the final state q_0^6, a transition $(q_1^6, q_0^6, \$: \$)$ has to be taken. Thus, after $\#_i$ is consumed on the input tape (and produces nothing on the output tape), a \$ sign is generated on the output tape for η_6. This implies that there should be a \$ after $\eta[i-1]$ in η. It conflicts with the fact that $|\eta| = i$.

This concludes the proof on Case 2. □

Lemmas 5.39 and 5.41 immediately lead to Theorem 5.42, given that $\mathcal{M}^+_{r\to\omega} = \mathcal{M}^+_r \,||\, \mathcal{A}_{10}(r, \omega)$.

Theorem 5.42 *Given any* $r \in R$ *and* $\kappa, \zeta, \omega \in \Sigma^*$, $\zeta = \kappa^+_{r\to\omega}$ *iff* $(\kappa, \zeta) \in L(\mathcal{M}^+_{r\to\omega})$.

6. Solving simple linear string equation

A SISE equation is first decomposed into a number of atomic steps. Each step is a string operation such as concatenation, substring extraction, and regular replacement. Most atomic operations can be precisely modeled using finite state transducers. Given a finite state transducer, it is straightforward to compute the input word given an output word, and vice versa, using FST composition and projection operations. Then the results of atomic steps are "chained" (composed) using the standard FST composition operation, which results in a "*solution pool*". Intuitively, for each variable, its solution pool is the set of all possible values it could take in some concrete solution. To solve a SISE, the solution pool is computed first, then based on which, concrete solutions are derived.

Definition 6.1 Let $\mu \equiv r$ be a SISE and v be a string variable in μ. The *solution pool* for v, denoted by $sp(v)$, is defined as $sp(v) = \{\omega \mid \omega = \rho(v)$ where ρ is a solution to $\mu \equiv r\}$. □

Example 6.2 By Definition 3.5, a SISE solution ρ is defined as a function $\rho : \vec{V} \to \Sigma^*$ where $\vec{V}$ is always finite, hence ρ is often written as a set of tuples. Let $x, y \in V$ and $a \in \Sigma$. Given a SISE $x \circ y \equiv aa$, the following are all concrete solutions to the equation: $\rho_1 = \{(x, \epsilon), (y, aa)\}$, and $\rho_2 = \{(x, a), (y, a)\}$, and $\rho_3 = \{(x, aa), (y, \epsilon)\}$. This immediately leads to $sp(x) = \{\epsilon, a, aa\}$ and $sp(y) = \{\epsilon, a, aa\}$.

It is shown later that $sp(v)$ is always a regular language for any string variable v in any SISE. In the following discussion, we describe an algorithm that takes a SISE as input and constructs as output regular expressions that represent the solution pools for all string variables in the equation.

6.1. Basic cases of SISE

According to Definition 3.3, the set of string expressions E is constructed recursively based on the atomic case (rule 1) and three operations: concatenation (rule 2), substring extraction (rule 3), and string replacement (rule 4). Solving a SISE can be reduced to solving the four basic cases. The atomic case is trivial. That is, for a SISE equation $E \equiv r$: when $E = x$ and $x \in V$, then the solution pool of x is simply $L(r)$. We next consider the other three cases.

6.1.1. Substring extraction case

A substring extraction case is formally defined as: $\mu[i, j] \equiv r$, where $\mu \in E$ and $i, j \in N$ with $i \le j$. The following equivalence is useful for removing a substring extraction operator during equation transformation.

Equivalence 6.3 For any SISE of the form $\mu[i, j] \equiv r$ where $\mu \in E$ and $i, j \in N$ with $i \le j$, ρ is a solution to $\mu[i, j] \equiv r$ iff it is a solution to $\mu \equiv \Sigma^i(r \cap \Sigma^{j-i})\Sigma^*$.

Example 6.4 Consider SISE $x[2, 4] \equiv ab^*$ where $x \in V$ and $a, b \in \Sigma$. Using Equivalence 6.3 we obtain $x \equiv \Sigma^2(ab^* \cap \Sigma^2)\Sigma^*$ and hence $sp(x) = \Sigma^2 ab\Sigma^*$. Consider an arbitrary word in $sp(x)$, e.g., $x = ccabccc$. Let $\rho = \{(x, ccabccc)\}$. According to Definition 3.5, it is a solution to $x[2, 4] \equiv ab^*$, because $\rho(x[2, 4]) \cap ab^* = \{ab\}$ is not empty. □

6.1.2. Concatenation case

A basic equation of concatenation has the form $\mu\nu \equiv r$, where $\mu, \nu \in E$. First consider a special case when $xr_1 \equiv r_2$, where $x \in V$ and $r_1, r_2 \in R$.[14] This can be solved using the *right quotient operator* of regular expressions [HU79]. By convention, the right quotient $r_2/r_1 = \{x \mid xw \in r_2 \text{ and } w \in r_1\}$. Similarly, the *left quotient* is defined as $r_2\backslash r_1 = \{x \mid wx \in r_2 \text{ and } w \in r_1\}$. We know that if r_1 and r_2 are regular, so are r_2/r_1 and $r_2\backslash r_1$.

Now consider the general case $\mu\nu \equiv r$ where both μ and ν are non-trivial string expressions. Let $approx(\nu)$ be the result of replacing every variable in ν with Σ^*. It follows that $approx(\nu)$ is a regular expression. Given a solution ρ and a string expression μ, let ρ_μ represent the restriction of ρ to the variable set of μ (i.e., if regarded as a tuple set, ρ_μ is a subset of ρ and it includes the tuples related to variables in μ). We have Equivalence 6.6. It relies on Lemma 6.5. In the following, we use $sp_{\mathcal{S}}(v)$ to denote the solution pool of variable v in a SISE $\mathcal{S}$.

Lemma 6.5 *Let ν be a string expression on Σ. For any $w \in approx(\nu)$ there is a solution to $\nu \equiv w$.*

Equivalence 6.6 Given $r_2 \in R$, and $\mu, \nu \in E$, and let $\mathcal{S}$, $\mathcal{S}_\mu$, and $\mathcal{S}_\nu$ be the following SISE equations, respectively: $\mu\nu \equiv r_2$, $\mu \equiv r_2/approx(\nu)$, and $\nu \equiv r_2\backslash approx(\mu)$. For each variable v in $\mathcal{S}$, the following are true:

1. If v occurs in μ, then $sp_{\mathcal{S}}(v) = sp_{\mathcal{S}_\mu}(v)$.
2. If v occurs in ν, then $sp_{\mathcal{S}}(v) = sp_{\mathcal{S}_\nu}(v)$.

Proof. Note that due to the single occurrence restriction of SISE, if v appears in μ, then it cannot appear in ν, and vice versa. We prove statement (1) and the proof for (2) is similar.

(Direction $sp_{\mathcal{S}}(v) \subseteq sp_{\mathcal{S}_\mu}(v)$): We can show that if ρ is a solution to $\mu\nu \equiv r_2$, then ρ_μ is a solution to $\mu \equiv r_2/approx(\nu)$. This can be inferred from the following facts: (L1) Since ρ is a solution to $\mathcal{S}$, there exist $w \in r_2$, $w_1 \in \rho(\mu)$, and $w_2 \in \rho(\nu)$ s.t. $w_1 w_2 = w$,[15] (L2) The w_2 in (L1) is contained in $approx(\nu)$. L1 and L2 imply that $\rho(\mu) \cap r_2/approx(\nu) \neq \emptyset$. Thus ρ_μ is a solution to $\mathcal{S}_\mu$.

(Direction $sp_{\mathcal{S}}(v) \supseteq sp_{\mathcal{S}_\mu}(v)$): We only need to prove that if ρ_μ is a solution to $\mu \equiv r_2/approx(\nu)$, then ρ_μ can be extended into a solution to $\mu\nu \equiv r_2$. Since ρ_μ is a solution to $\mathcal{S}_\mu$, we know that there exist $w_1' \in \rho_\mu(\mu)$, $w_2' \in approx(\nu)$, and $w' \in r_2$ s.t. $w_1' w_2' = w'$. By Lemma 6.5, there always exists a solution to $\nu \equiv w_2'$, and let it be ρ_ν. Thus $\rho' = \rho_\mu \cup \rho_\nu$ is a solution to $\mu\nu \equiv r_2$ (this is possible because μ and ν have disjoint variable sets by Definition 3.4). □

6.1.3. Replacement case

The replacement case has a general form of $\mu_{r_1 \to \omega} \equiv r_2$ (similarly $\mu^-_{r_1 \to \omega} \equiv r_2$, $\mu^+_{r_1 \to \omega} \equiv r_2$), where $\mu \in E$, $r_1, r_2 \in R$, and $\omega \in \Sigma^*$. In the following we discuss the solution to $\mu_{r_1 \to \omega} \equiv r_2$. The handling of procedural replacements will be similar, because all of them use finite state transducer algorithms.

Our goal is to construct an FST, denoted by $\mathcal{M}_{r_1 \to \omega \Rightarrow r_2}$ s.t. $(s, \eta) \in L(\mathcal{M}_{r_1 \to \omega \Rightarrow r_2}) \Leftrightarrow \eta \in L(r_2)$ and $\eta \in s_{r_1 \to \omega}$. Let $\mathcal{M}_1(r_2)$ be the FST that accepts the identity relation $\{(s, s) \mid s \in L(r_2)\}$. Let $\mathcal{M}_{r_1 \to \omega}$ be the FST shown in Lemma 4.5, i.e., $(s, \eta) \in L(\mathcal{M}_{r_1 \to \omega})$ iff $\eta \in s_{r_1 \to \omega}$. $\mathcal{M}_{r_1 \to \omega \Rightarrow r_2}$ can then be constructed as $\mathcal{M}_{r_1 \to \omega} \mid\mid \mathcal{M}_1(r_2)$. Similarly, for the pure reluctant semantics, $\mathcal{M}^-_{r_1 \to \omega \Rightarrow r_2}$ can be constructed as $\mathcal{M}^-_{r_1 \to \omega} \mid\mid \mathcal{M}_1(r_2)$, where $\mathcal{M}^-_{r_1 \to \omega}$ is defined in Theorem 5.23. $\mathcal{M}^+_{r_1 \to \omega \Rightarrow r_2}$ can be defined similarly for the greedy semantics (using Theorem 5.42).

Equivalence 6.7 For any SISE of the form $\mu_{r_1 \to \omega} \equiv r_2$ where $\mu \in E$, $r_1, r_2 \in R$, and $\omega \in \Sigma^*$. ρ is a solution to $\mu_{r_1 \to \omega} \equiv r_2$ iff it is a solution to $\mu \equiv r$ where $L(r) = \{s \mid (s, \eta) \in L(\mathcal{M}_{r_1 \to \omega \Rightarrow r_2})\}$.

The $L(r)$ in the above Equivalence can be computed by projecting $\mathcal{M}_{r_1 \to \omega \Rightarrow r_2}$ to its input tape, which results in a finite state machine, representing a regular language. The same applies to the pure greedy and reluctant semantics, using $\mathcal{M}^+_{r_1 \to \omega \Rightarrow r_2}$ and $\mathcal{M}^-_{r_1 \to \omega \Rightarrow r_2}$.

[14] We abuse the notation here because xr_1 is not a standard string expression. For this equation, we are looking for the solution of x and two words $w_1 \in r_1$ and $w_2 \in r_2$ s.t. $x \circ w_1 = w_2$.

[15] Here $\rho(\mu)$ represents the result of replacing in μ each variable v with its value defined by ρ. Notice that it could be a set of words if there is any declarative replacement involved.

```
1 function computeSolutionPool(μ ≡ r)
2  switch (μ):
3     case x ∈ V: return {(x, r)}
4     case r1 ∈ R: if L(r1) ∩ L(r) ≠ ∅ return ∅ o.t. return ⊥
6     case μ[i, j]: return computeSolutionPool(μ ≡ Σ^i(r ∩ Σ^{j−i})Σ*)
7     case μ_{r1→ω}: return computeSolutionPool(μ ≡ {s | (s, η) ∈ L(M_{r1→ω} ⇒r)})
8     case μ+_{r1→ω}: return computeSolutionPool(μ ≡ {s | (s, η) ∈ L(M+_{r1→ω} ⇒r)})
9     case μ−_{r1→ω}: return computeSolutionPool(μ ≡ {s | (s, η) ∈ L(M−_{r1→ω} ⇒r)})
10    case μν:
11       Let r1 be approx(μ) and r2 be approx(ν)
12       return computeSolutionPool(μ ≡ r/r2) ∪ computeSolutionPool(ν ≡ r\r1)
```

Fig. 9. Algorithm for computing solution pool

6.2. Recursive algorithm for computing solution pool

Based on Equivalences 6.3, 6.6, and 6.7, one can develop a recursive algorithm for generating the solution pool for all variables in a SISE. The algorithm is shown in Fig. 9. Function `computeSolutionPool()` returns a set of tuples, with each tuple representing a solution pool (note: not a solution) for a variable. We use $\perp$ to represent "no solution". When applying any set operation (e.g., intersection and union) on $\perp$, the result is $\perp$. Note that $\perp$ is not the same as empty set $\emptyset$.

Example 6.8 Consider the following SISE equation where $x \in V$ and $a, b, c \in \Sigma$:

$$x^{+}_{a^{+}\to b} \circ (y[2,3]) \equiv b^{+}c \tag{2}$$

According to the algorithm (line 10), the equation is reduced to two sub-problems:

$$x^{+}_{a^{+}\to b} \circ (\Sigma^{*}[2,3]) \equiv b^{+}c \tag{3}$$

$$\Sigma^{*}{}^{+}_{a^{+}\to b} \circ (y[2,3]) \equiv b^{+}c \tag{4}$$

We tackle Eq. 3 first. Using Equivalence 6.6, Eq. 3 can be transformed into the following:

$$x^{+}_{a^{+}\to b} \equiv b^{+}c/\Sigma^{*}[2,3] = b^{+}c/\Sigma = b^{+} \tag{5}$$

To solve Eq. 5, a finite state transducer $\mathcal{M}^{+}_{a^{+}\to b}$ is needed and then $\mathcal{M}^{+}_{a^{+}\to b \Rightarrow b^{+}}$ is constructed as $\mathcal{M}^{+}_{a^{+}\to b} \,||\, Id(b^{+})$. By projecting $\mathcal{M}^{+}_{a^{+}\to b \Rightarrow b^{+}}$ to its input tape, we have $\{(x, (a \mid b)^{+})\}$ as the solution pool of Eq. 3. Similarly, solving Eq. 4 results in $\{(y, \Sigma^{2}c\Sigma^{*})\}$. This eventually yields a variable solution pool $\{(x, (a \mid b)^{+}), (y, \Sigma^{2}c\Sigma^{*})\}$ according to line 12 in Fig. 9.

The validity of the algorithm in Fig. 9 is stated in the following theorem.

Theorem 6.9 *For any SISE $\mu \equiv r$ and any variable v in μ, the following are true:*

1. *Let ρ be the set of tuples returned by* `computeSolutionPool`$(\mu \equiv r)$. *When $\rho \neq \perp$, for each variable v in μ: ρ contains one and only one tuple on v, and it is $(v, sp(v))$.*
2. *$sp(v)$ is a regular language.*

Proof. The conclusion results from the following arguments: (a) Each variable v appears only once in LHS (by Definition 3.4). This results in the conclusion that there is at most one tuple related to v in ρ. (b) Given any SISE equation E_1 and let E_2 be the equation resulted from any of the equation transformations outlined in Fig. 9. A variable mapping ϱ is a solution to E_1 iff it is a solution to E_2. (c) The same set of solutions always induce the same solution pool for each variable involved. (d) The RHS of any SISE during the solution process is always a regular language. When it boils down to case 1, i.e., line 3 of Fig. 9, a variable is always assigned a regular solution pool. This leads to statement (2) in the theorem. □

Theorem 6.10 *The worst complexity of the algorithm in Fig. 9 is $O(|\mu| \times 2^{8\times 2^{|\mu|}})$.*

```
StrExp term1 = StrExp.fromRegExpConst(new RegExpConst("uname='"));
StrExp term3 = StrExp.fromRegExpConst(new RegExpConst("' pwd='"));
StrExp x = StrExp.fromVar(new Variable("x"));
x = StrExp.replaceAll(x, "'", "''");
x = StrExp.substring(x, 0, 5);
StrExp lhs = StrExp.concat(term1, x);
lhs = StrExp.concat(lhs, term3);
RegExpConst rhs = new RegExpConst("uname='([^']|'')*'");
SISE slse = new SLSE(lhs,rhs);
Solution sol = se.solveMax();
SolutionItem si = sol.findVarByName("x");
Automaton au = ((RegExpConst) si.value).getAutomaton();
if(!au.run("aaaa'")){
  fail("error!");
}
```

Listing 4. Solve SISE Using SUSHI

In `computeSolutionPool` the most expensive computation is the case $\mu^{+}_{r_1 \to \omega}$, which needs $\mathcal{M}^{+}_{r_1 \to \omega} \Rightarrow_r$, a composition of eight transducers. Among them, $\mathcal{A}_3(r_1)$ is the largest transducer, whose size is $2^{2^{|r_1|}}$. Since $|r_1| < |\mu|$, we can approximate the worst complexity of the $\mu^{+}_{r_1 \to \omega}$ case as $O(2^{8 \times 2^{|\mu|}})$. Finally, given $\mu \equiv r$, `computeSolutionPool` is called recursively for at most $|\mu|$ times. This leads to the complexity result in Theorem 6.10.

Given the solution pool, a concrete solution can be generated by concretizing the valuation of a variable one by one using a counter loop on the number of variables in the equation. In each iteration, nondeterministically instantiate one variable using a value contained in its variable solution pool. Thus a new SISE equation is obtained. Solving this equation would lead to the solution pool to be used in the next iteration. Starting from the initial solution pool, the concretization process will always terminate with a concrete solution generated.

7. SUSHI constraint solver

SISE constraint solving is implemented in a Java library called SUSHI, which can be called as a back-end constraint solver by model checkers and symbolic execution engines. The source code of SUSHI is available at [Fu09]. This section presents the implementation details of SUSHI.

7.1. SUSHI API interface

SUSHI consists of two Java packages: the automaton utilities and the constraint solver. The automaton package provides the support of finite state transducer. It includes FST operations such as concatenation, Kleene star, projection, composition, etc. A finite state automaton is directly modeled using the `dk.brics.automaton` package [Mø]. Listing 4 presents a sample program for solving a string equation shown below:

$$\texttt{uname='} \circ x^{-}_{\to''}[0, 5] \circ \texttt{' pwd='} \equiv \texttt{uname='([^']|'')*'}$$

The program first constructs the LHS, which involves two string operations: `substring` and `replaceAll` on variable x. The RHS is a regular expression. Calling `solveMax` returns a `Solution`, which is a collection of solution pools. For each variable, the solution pool is represented using an FSA encoded using `dk.bricks.Automaton`.

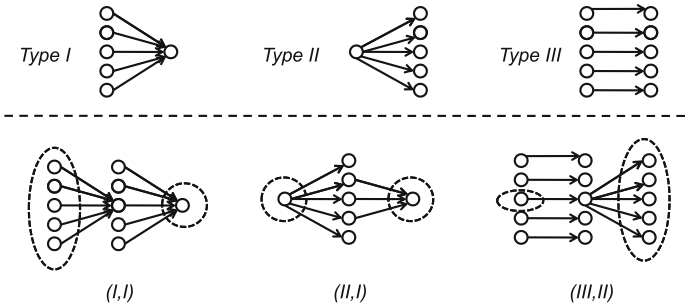


Fig. 10. SUSHI FST transition set

7.2. Compact representation of FST

In practice, to perform inspection on the user input of web applications, SUSHI has to handle a large alphabet represented using 16-bit Unicode. If FST transitions are represented explicitly, the performance (in both time and memory) would be undesirable. We present a symbolic encoding of FST that allows SUSHI to scale for large constraints. We use a compact representation called transition set.

Definition 7.1 A *SUSHI FST Transition Set* (SFTS) $\mathcal{T}$ is represented as a tuple $(q, q', \phi : \varphi)$, where q, q' are the source and destination states. The *input charset* $\phi = [n_1, n_2]$ with $0 \leq n_1 \leq n_2$ represents a range of input characters, and the *output charset* $\varphi = [m_1, m_2]$ with $0 \leq m_1 \leq m_2$ represents a range of output characters. $\mathcal{T}$ includes a set of transitions with the same source and destination states: $\mathcal{T} = \{(q, q', a : b) \mid a \in \phi \text{ and } b \in \varphi\}$. If $|\phi| > 1$ and $|\varphi| > 1$, it is required that $\phi = \varphi$, and $\mathcal{T} = \{(q, q', a : a) \mid a \in \phi\}$. For ϕ and φ, ϵ is represented using $[-1, -1]$.

Excluding the ϵ transitions, there are three categories of SFTSes:

1. *Type I:* $|\phi| > 1$ and $|\varphi| = 1$, thus $\mathcal{T} = \{(q, q', a : b) | a \in \phi \text{ and } \varphi = \{b\}\}$.
2. *Type II:* $|\phi| = 1$ and $|\varphi| > 1$, thus $\mathcal{T} = \{(q, q', a : b) | b \in \varphi \text{ and } \phi = \{a\}\}$.
3. *Type III:* $|\varphi| = |\phi| > 1$, thus $\mathcal{T} = \{(q, q', a : a) \mid a \in \phi\}$.

The top of Fig. 10 gives an intuitive illustration of these SFTS types (which relate the input and output chars). They can capture some frequently used replacement operations in practice.

Example 7.2 In the following, we list some typical examples of SFTS. Here $\backslash u0061$ and $\backslash u007A$ are the Unicode values for a and z, respectively.

1. To remove input symbols other than lower case English characters can be represented using two type I SFTSes $(q, q', [\backslash u0000, \backslash u0060] : \epsilon)$ and $(q, q', [\backslash u007B, \backslash uFFFF] : \epsilon)$.
2. Identity relation $Id(\Sigma)$ can be represented using $(q, q', [\backslash u0000, \backslash uFFFF] : [\backslash u0000, \backslash uFFFF])$.
3. Type II SFTS $(q, q', \backslash u0061 : [\backslash u0061, \backslash u007A])$ denotes a non-deterministic replacement that substitutes a lower case character for a.

In the following, we use $(q, q', [a, z] : [a, z])$ to denote $(q, q', [\backslash u0061, \backslash u007A] : [\backslash u0061, \backslash u007A])$ for short. □

The algorithm for supporting FST operations (such as union, Kleene star) should be customized correspondingly. In the following, we take FST composition as one example. Let $\mathcal{A} = (\Sigma, Q, q, F, \delta)$ be the composition of $\mathcal{A}_1 = (\Sigma, Q_1, q_0^1, F_1, \delta_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, s_0^2, F_2, \delta_2)$. Given $\tau_1 = (t_1, t_1', \phi_1 : \varphi_1)$ in $\mathcal{A}_1$ and $\tau_2 = (t_2, t_2', \phi_2 : \varphi_2)$ in $\mathcal{A}_2$, where $\varphi_1 \cap \phi_2 \neq \emptyset$, an SFTS $\tau = (s_1, s_2, \phi : \varphi)$ is defined for $\mathcal{A}$ s.t. $s_1 = (t_1, t_2)$, $s_2 = (t_1', t_2')$, and the input/output charset of τ is defined following the table in Fig. 11. For all cases except (I,II), the algorithm produces one SFTS given a pair of SFTSes from $\mathcal{A}_1$ and $\mathcal{A}_2$. For example, when both τ_1 and τ_2 are type I, we have $\phi = \phi_1$ and $\varphi = \varphi_2$. Only the (I,II) case produces a set of $|\phi_1| \times |\varphi_2|$ SFTSes (each with a singleton input/output charset). The bottom part of Fig. 10 shows the intuition of the algorithm. The dashed circles represent the corresponding input/output charset.

Type of τ_1	Type of τ_2	input of τ	output of τ
I	I	ϕ_1	φ_2
II	I	ϕ_1	φ_2
III	I	$\varphi_1 \cap \phi_2$	φ_2
I	II	$\{x\}$ where $x \in \phi_1$	$\{y\}$ where $y \in \varphi_2$
II	II	ϕ_1	φ_2
III	II	$\varphi_1 \cap \phi_2$	φ_2
I	III	ϕ_1	φ_1
II	III	ϕ_1	$\varphi_1 \cap \phi_2$
III	III	$\varphi_1 \cap \phi_2$	$\varphi_1 \cap \phi_2$

Fig. 11. FST composition algorithm for SFTS encoding

Type	SFTS of $\mathcal{A}_1$	SFTS of $\mathcal{A}_2$	Resulting SFTS
(I,I)	$(q, q', [a, b] : [c, c])$	$(p, p', [c, d] : [e, e])$	$(r, r', [a, b] : [e, e])$
(II,I)	$(q, q', [a, a] : [c, d])$	$(p, p', [c, d] : [e, e])$	$(r, r', [a, a] : [e, e])$
(III,I)	$(q, q', [a, b] : [a, b])$	$(p, p', [b, d] : [e, e])$	$(r, r', [b, b] : [e, e])$
(I,II)	$(q, q', [a, b] : [c, c])$	$(p, p', [c, c] : [d, e])$	$\{(r, r', [a, a] : [d, d]),\ (r, r', [a, a] : [e, e]),$ $(r, r', [b, b] : [d, d]),\ (r, r', [b, b] : [e, e])\ \}$

Fig. 12. SFTS composition examples

Example 7.3 Figure 12 lists four examples of the application of the SFTS composition algorithm. In each example, $\mathcal{A}_1$ and $\mathcal{A}_2$ are both single-transition finite state transducers. The resulting FST has one SFTS for all except the (I,II) case. The source/destination states of the resulting SFTS is defined as $r = (q, p)$ and $r' = (q', p')$.

8. Evaluation

Recall that the SISE theory supports two features, i.e., *unbounded string length* and *regular replacement operations*, which could lead to undecidability in a generic string constraint system. Syntactic restrictions are used to ensure the decidability of the solution algorithm. We are interested in the impacts of these design decisions: (1) How would the syntactic restriction affect the applicability of SISE? and (2) What is the efficiency of the SUSHI constraint solver? This section presents the evaluation results. All experiments are conducted on a Linux Ubuntu 10.04 system with an Intel 2.0GHz Xeon CPU and 3GB RAM.

8.1. Evaluation of applicability

To evaluate the applicability of SUSHI, we rely on the Kaluza dataset generated by the Kudzu toolset [SAH+10]. Kudzu, designed by Saxena et al., uses symbolic execution to extract string constraints from the client-side scripts of 18 web applications and it runs the Kaluza solver. The set of constraints are publicly available on the Kaluza tool website [SAMS10].

Kaluza dataset consists of four categories of constraints: 19985 small satisfiable constraints (referred to as `SAT_small` in our later discussion), 1835 big satisfiable constraints (`SAT_big`), 11761 small unsatisfiable constraints (`UNSAT_small`), and 21469 big unsatisfiable constraints (`UNSAT_big`). Each Kaluza constraint is essentially a conjunction of atomic string constraints, though in practice the Kaluza solver supports one layer of disjunction of conjunctions using `IF-ELSE` without nesting. An atomic Kaluza constraint can be string concatenation, linear integer constraints on variable length, and equality comparison on strings. All other string operations, such

as `substring`, `charAt`, `replacement`, and `indexOf` are translated to the form of basic core constraint.[16] The Kaluza solver first calls an integer constraint solver to decide (instantiate) the length of each string variable, then models each string variable as a vector of bits, and uses a SAT solver to generate their contents. Thus, Kaluza solves constraints using a *bounded length* approach.

Our evaluation experiment is designed upon the Kaluza dataset, and it is set up as follows. First, we parse a Kaluza constraint specification, translate it to a SISE equation (or a conjunction of SISE equations). Then the SISE constraints are fed to SUSHI solver for solution. To increase confidence, we use SUSHI to generate a concrete variable solution (if there is any), append it to the original Kaluza specification, and supply it to the Kaluza constraint solver. The results of the two solvers are then compared.

8.1.1. Kaluza to SISE

Kaluza constraints are translated to a conjunction of SISE equations using a three-step algorithm. Here, SISE equations are extended slightly to accommodate Kaluza. Each string expression μ in the LHS of a SISE may be associated with a range restriction $R(\mu)$. To solve such an extended SISE equation, The original SISE solution algorithm in Fig. 9 remains the same, except that the approximation of μ is now $approx(\mu) \cap R(\mu)$. The conversion algorithm is briefly described as below:

1. (Concat Graph) A concatenation graph of string variables is established.
2. (Integer Constraint Projection) The conjunction of all integer constraints are projected to each string variable using existential quantification elimination, which is accomplished using the Presburger arithmetic constraint solver Omega [Pug94]. Then all length constraints (on single variables) are eventually translated to a regular expression. The algorithm then verifies if the conjunction of the projected integer constraints is equivalent to the original set of integer constraints. If the equivalence test fails, the conversion effort is declared as failed (reported as the `IntProj` failure).
3. (Translation) For each root node in the concatenation graph, a SISE constraint is constructed. An inspection check is made to make sure that each string variable occurs only in one SISE equation and once. Otherwise, the conversion fails (reported as the `SingleOcc` failure).

Example 8.1 Consider a simple Kaluza constraint presented on the left of Fig. 13. It is a conjunction of four atomic constraints. In Kaluza, "." represents string concatenation. Its equivalent SISE constraint, as the result of the conversion algorithm, is presented on the right of Fig. 13.

Given the Kaluza constraint specification, the conversion algorithm first produces two trees in the concatenation graph. It then collects all integer constraints in the specification and let it be $I_0 : |T_4| = |T_5|$. The algorithm then conservatively builds another constraint called I_1, which is implied by the concatenation and string equality comparison. I_1 is defined as follows.

$$I_1 : |T_1| = |T_2| + |T_3| \wedge |T_4| = |T_5| + |T_6| \wedge \bigwedge_{1 \le i \le 6} |T_i| \ge 0.$$

Then $I_0 \wedge I_1$ is projected to each string variable using existential elimination. For example, for variable T_6, its projected integer constraint is obtained using $\exists |T1|, |T2|, |T3|, |T4|, |T5| : I_0 \wedge I_1$. This yields $|T_6| = 0$ for $T6$ and similarly, $|T_i| \ge 0$ ($i \in \{1, 2, 3, 4, 5\}$) for all other string variables. Readers can verify that $I_1 \wedge |T_6| = 0 \wedge \bigwedge_{1 \le i \le 5} |T_i| \ge 0$ is equivalent to $I_0 \wedge I_1$. This suggests that the constraint projection is successful.

The projected integer constraint $|T_6| = 0$ is translated to a range restriction $R(T_6) \equiv \Sigma^0$ in the constructed SISE equation, and it is used to restrict the range of T_6 in the solution. We call T_6 "tagged" in the concat graph. Similar is T_4 because it has an associated constraint (K4).

The final translation from the two trees to SISE is straightforward. Note that only one SISE (i.e., $S1$ in Fig. 13) is generated and $S2$ is the range restriction for T_6. This is because all untagged root nodes are removed by the optimization procedure. During the process that translates the SUSHI solution back to Kaluza, the values of these pruned nodes are recovered. □

[16] The translation of regular replacement operations in Kaluza is approximated.

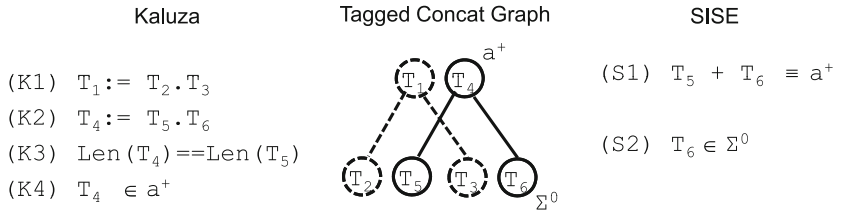


Fig. 13. An example of conversion from Kaluza to SISE

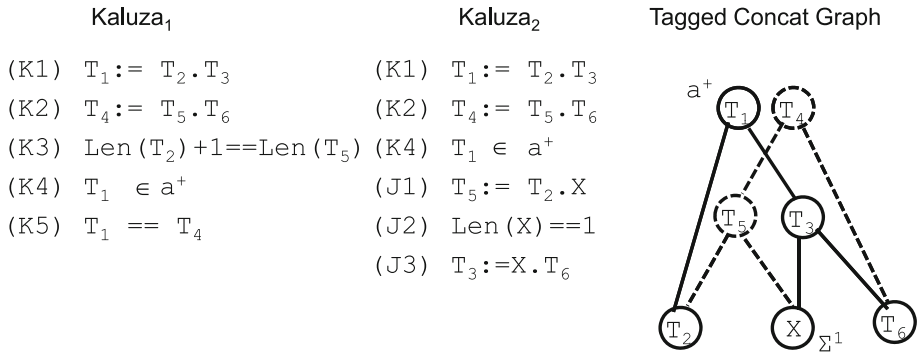


Fig. 14. An example of syntactic transformation

DataSet	Count	Success					Failure			Exception
		I	II	III	IV	Total	IntProj	SingleOcc	Total	
SAT_small	19985	92.12%	3.33%	4.31%	0.01%	99.77%	0.00%	0.00%	0.00%	0.23%
UNSAT_small	11761	85.90%	8.44%	5.00%	0.33%	99.68%	0.00%	0.00%	0.00%	0.32%
SAT_big	1835	10.08%	3.49%	72.04%	0.05%	85.67%	0.00%	0.00%	0.00%	14.33%
UNSAT_big	21469	4.50%	3.00%	38.73%	17.80%	64.04%	11.97%	5.47%	17.44%	18.52%

Fig. 15. Results of conversion from Kaluza to SISE

In practice, the algorithm fails on many real Kaluza specifications included in the dataset. Additional syntactic transformation is needed to transform a Kaluza specification into an equivalent one, but without generating `IntProj` and `SingleOcc` violations on SISE constraints.

Example 8.2 The Kaluza example named $Kaluza_1$ in Fig. 14 cannot be directly translated to SISE. Its integer constraint $K3$, when projected to T_2 and T_5, has the binding between $|\ T_2\ |$ and $|\ T_5\ |$ lost. In addition, SISE is not able to model $K5$.

A syntactic transformation is applied to remove $K3$ and $K5$. Based on $K1$, $K2$, $K3$ and $K5$, we are able to introduce a new string variable X whose length is 1. Then the atomic constraints are restructured to form the new specification $Kaluza_2$, which is equivalent to $Kaluza_1$. In $Kaluza_2$, the integer constraint $J2$ can be projected without information loss. Then after pruning untagged nodes, additional `SingleOcc` violations are removed. This makes the conversion algorithm applicable to $Kaluza_2$. □

8.1.2. Experimental results

We run experiments on all of the four Kaluza datasets. It takes about 80 hours of CPU time on a Dell workstation with an Intel 2.0GHz Xeon CPU. Figure 15 presents the success rate of the conversion efforts from Kaluza to SISE. The first two columns are the dataset name and the number of SISE constraints used in the experiment. The next four columns present the ratio of success cases. When the conversion is successful, we classify the resulting SUSHI constraints into four categories: (I) pure SUSHI constraints (without any range restriction attached to any string expression), (II) extended SUSHI constraints (with range restrictions), (III) SUSHI resulted from

DataSet	Size			Cost (milliseconds)			
	Kaluza	Vars	SISE	Conversion	SUSHI	Kaluza	Verification
SAT_small	7.78	2.24	0.27	106.15	6.61	364.48	221.11
UNSAT_small	9.94	3.15	0.46	300.11	9.39	451.21	-
SAT_big	275.55	175.40	7.85	15368.01	19546.96	1688.42	2104.43
UNSAT_big	182.57	145.78	4.30	13650.77	1094.33	2111.28	-

Fig. 16. Cost of conversion from Kaluza to SISE

syntactic transformation on the original Kaluza constraints, and (IV) no SUSHI constraints generated due to early discovery of unsatisfiable integer constraints. The failure scenarios consist of two cases: integer constraint projection error and violation of the single occurrence restriction. The `Exception` column includes those cases that are caused by various run time errors, e.g., Kaluza reports syntax error on the input constraint[17] and some rare cases of Omega constraint solver crashes. Most of the exceptions correspond to Omega solver time-outs when existential elimination operations are performed to project integer constraints for a Kaluza input. In these experiments, the Omega calculator is set to terminate after 300 seconds.

The conversion algorithm works very well for the `SAT_small` and `UNSAT_small` datasets, with over 99% of the small constraints successfully converted. About 86% of the `SAT_big` constraints are converted, and most of them belong to Category III, i.e., syntactic transformation is needed first. The conversion rate is the lowest for `UNSAT_big` (only 64%), and the conversion of 12% of `UNSAT_big` fails at the integer constraint projection stage.

Figure 16 presents the running cost of the experiment. Columns 2 and 4 are the average size (i.e., the number of atomic constraints) of each Kaluza input and the mean size (i.e., the number of SISE equations) of the resulting SISE constraint, respectively. Column 3 shows the average number of variables contained in a Kaluza input. It is interesting to notice that the average SISE size is below 1.0 for `SAT_small` and `UNSAT_small`. This is due to the effects of root node pruning during the conversion. Columns 5, 6, 7, 8 display the average cost of conversion from Kaluza to SUSHI, constraint solving using SUSHI, constraint solving using Kaluza, and the final verification process (using Kaluza to solve the constraint appended with a concrete solution). The cost of SUSHI constraint solving is in general much lower than that of Kaluza (except for `SAT_big`), however, this would be an unfair comparison, because the conversion cost is not included (where integer constraints are handled). For the `SAT_big` and `UNSAT_big` datasets, the conversion cost is large because of the existential elimination operations, which are costly.

In conclusion, it is not a good idea to directly apply SUSHI to solving Kaluza constraints, because the full power of SUSHI (e.g., its ability to handle string operations such as replacement and substring extraction) is not taken advantage of. In the Kaluza data set, hundreds of variables and atomic constraints may be generated for one single string operation such as replacement. These variables incur the great cost of integer constraint projection during the conversion process. A more reasonable comparison in terms of constraint solving cost is presented in Sect. 8.2.2.

8.2. Evaluation of efficiency

8.2.1. Scalability

We are interested in the performance of SUSHI as a constraint solver. Figure 17 lists five SISE equations for stress-testing the SUSHI package. Note that each equation is parametrized by an integer n (ranging from 1 to 40). For example, when n is 30, the size of the RHS of `eq4` is 60.

The sample set covers all string operations we discussed earlier. In Fig. 18, the first two diagrams present the size of the FST (the number of states and the number of transitions) used in the solution process, the third diagram is the size of the FSA used for representing solution pools, and the last shows the time spent on running each test. As shown in Fig. 18, SUSHI scales well in most cases. The figure also suggests that the solution cost of a SISE equation mainly depends on the complexity of the automata structure of the resulting solution pool (e.g., readers can compare the cost of `eq4` and `eq5`). In addition, the experimental data (see Fig. 18a, b) indicate that to model greedy regular replacement (e.g., `eq4`) is expensive because the modeling process involves composition of seven transducers.

[17] The `UNSAT_big` dataset contains a small set of constraints that Kaluza solver fails to parse.

ID	Equation
eq1	$x \circ \texttt{a}\{n,n\} \equiv (\texttt{a}\|\texttt{b})\{2n,2n\}$
eq2	$x[n,2n] \equiv \texttt{a}\{n,n\}$
eq3	$x \circ \texttt{a} \circ y[0,n] \equiv \texttt{b}\{n,n\}\texttt{ab}\{n,n\}$
eq4	$x^{+}_{\texttt{a}^{+}\rightarrow \texttt{b}\{n,n\}} \equiv \texttt{b}\{2n,2n\}$
eq5	$\texttt{uname='} \circ x^{-}_{\texttt{'},\texttt{''}}[0,n] \circ \texttt{' pwd='} \equiv \texttt{uname='[^'|'']*'}$

Fig. 17. Sample SUSHI equations

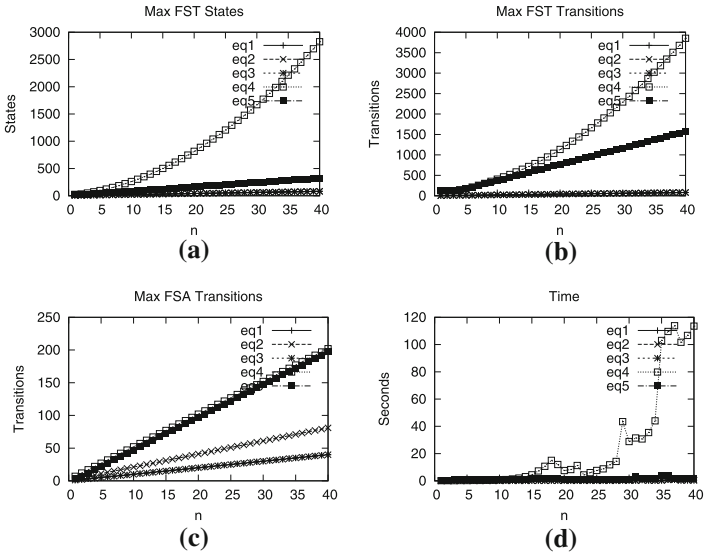


Fig. 18. Constraint solving cost using SUSHI

8.2.2. Comparative study with Kaluza

It is interesting to compare the performance of SUSHI and Kaluza when they are given the same set of string constraints at the same "abstract" level. In this experiment, we use the five sample SISE equations listed in Fig. 17 and the experiment is set up as follows, with a 300-second time-out for each constraint solving operations involved.

1. Given a SISE equation, we first obtain the running cost of SUSHI.
2. The SISE equation is then translated to an equivalent Kaluza specification. During the translation, we will have to bound the number of matches on a search pattern (if there is a regular replacement operator). The translation is always successful because Kaluza is strictly more expressive than SISE when length is bounded. The only exception is that the regular replacement semantics is approximated in Kaluza. For example, given the number of matches up to n, $s^{+}_{r\rightarrow\omega} \in r_2$ is essentially translated to a formula in the form of $\bigvee_{0\leq k\leq n} U(k)$, where $U(k)$ is defined as below. Notice that A_i is used in both s and t.

$$s = X_1 \circ X_2 \circ X_3 \ldots \circ X_k \circ X_{k+1} \wedge \bigwedge_{1\leq i\leq k} (X_i = A_i \circ B_i \wedge A_i \in \Sigma^* - \Sigma^* r \Sigma^* \wedge B_i \in r) \wedge X_{k+1} \notin r \wedge$$

$$t = Y_1 \circ Y_2 \circ Y_3 \ldots \circ Y_k \circ Y_{k+1} \wedge \bigwedge_{1\leq i\leq k} (Y_i = A_i \circ C_i \wedge A_i \in \Sigma^* - \Sigma^* r \Sigma^* \wedge C_i = \omega) \wedge Y_{k+1} = X_{k+1} \wedge t \in r_2$$

In practice, the disjunction of conjunctions in Kaluza is expressed using a collection of `IF-ELSE` statements.

3. We then convert the Kaluza specification back to SUSHI and obtain its running cost. This is to compare with the performance of SUSHI on the original SISE equation.

n	eq1				eq2				eq3				eq4				eq5			
	S_1	K_1	C	S_2	S_1	K_1	C	S_2	S_1	K_1	C	S_2	S_1	K_1	C	S_2	S_1	K_1	C	S_2
1	2	802	3	2	40	401	510	0	5	1102	615	3	111	301	209	3	233	19329	1155	143
5	9	1402	3	7	4	401	509	0	15	1102	614	11	81	301	208	13	273	8410	1856	333
9	21	1402	4	19	6	501	509	0	30	1201	616	31	175	301	209	30	366	19719	2740	646
13	39	1402	5	37	10	501	511	1	51	1202	619	40	792	301	208	60	486	63757	3981	1136
17	64	1402	6	58	21	501	611	1	77	1202	515	70	4113	301	209	98	626	30624	5865	1811
21	94	1402	6	85	35	501	510	1	107	1202	618	97	8167	401	211	144	795	-1	8443	2748
25	129	1503	6	120	53	502	612	1	144	1202	621	130	7411	401	209	207	975	-1	11881	4003
29	174	1507	9	169	108	504	512	1	184	1302	622	161	16973	401	219	293	1174	-1	16876	5710
33	357	1810	8	220	194	601	668	1	233	1402	662	203	36545	401	227	382	1434	-1	22509	7749
37	320	2103	9	274	359	701	564	1	307	1620	659	248	74869	501	211	478	1700	-1	66326	25397

Fig. 19. Comparative study of SUSHI and Kaluza constraint solving cost (in milliseconds)

Figure 19 presents the running cost (in terms of time in milliseconds) of Kaluza and SUSHI on the five sample equations, parameterized with n from 1 to 37 at a step of 4. For each SISE equation, column S_1 represents the time used by SUSHI to compute a solution pool. K_1 is the cost of Kaluza solver, C is the conversion cost from the translated Kaluza back to SISE, and S_2 is the running cost of SUSHI on the re-generated SISE equations from the Kaluza input. -1 in Fig. 19 denotes time out.

As shown in Fig. 19, the SUSHI constraint solving cost (S_1) on the five sample equations are compatible with the ones shown in Fig. 18. The most surprising finding is that S_2 (the cost of SUSHI on re-generated SISE constraints) is often smaller than S_1. This is because S_2 does not include the time for exploring the entire solution pool. Take $eq4$ as an example. Consider $eq4$: $x^{+}_{\mathtt{a}^+ \to \mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n, 2n\}$. Its shortest solution is ab^n. When solving a Kaluza input with `IF-ELSE` structures, SUSHI adopts a depth-first search in generating conjunction of atomic Kaluza constraints (and then converting them to a conjunction of SISE equations and then solving them). The search completes early when the shortest solution is found. For S_1, the whole solution space has to be explored and a solution pool (representing all possible concrete solution values for x) has to be generated.

Comparing SUSHI and Kaluza, we find that Kaluza scales better than SUSHI on $eq1$ to $eq4$ (K_1 vs. S_1), because it tries to search for a single solution. Comparing the cost for finding a single solution (K_1 vs. $C + S_2$), SUSHI wins on all the five sample equations, especially with a big margin on $eq5$ (Kaluza is timed out starting from $n = 21$). This is as expected. Recall that $eq5$ is a simplification of the password bypassing attack in Sect. 2. It is hard to find a proper value for x (because x has to contain the *right* number of single quotes to get them chopped by the substring extraction operation). It will take Kaluza time to search a working solution.

In summary, the solution pool approach works better when there are few solutions (i.e., the constraint is hard to solve). The search algorithm (e.g., adopted by Kaluza) works better in other cases.

8.3. Preliminary application of SUSHI

We demonstrate how *reusable* attack pattern rules are represented in SISE. We show how SUSHI is useful in security analysis. The application is motivated by the fact that most of the vulnerability reports do not come with a working exploit. This could be caused by several reasons: (1) the poster would not like to provide exploits to hackers, (2) it is time consuming to craft a working exploit, and (3) the vulnerability does not actually allow an effective attack (e.g., the overflowed buffer is too small to allow any shell code). SUSHI can be used to either find a working exploit or discharge case (3) for a specific attack pattern.

In the following we give one example of analyzing one recent XSS vulnerability [Lab09] in Adobe Flex SDK 3.3. A file named `index.template.html` is used for generating wrappers of application files in a FLEX project. It takes a user input in the form of "`window.location`" (URL of the web page being displayed), which is later built into the `embedAttrs`. The user input is eventually written into the DOM structure of the HTML file using `document.write(str)`.

Rule	Regular Expression Pattern
XSS	`.*<script>alert('XSS found!')</script>.*`
EffectiveScript	$\overline{\texttt{.*[a-zA-Z0-9\_]+ *= *"[^"]*<script.*>.*}}$
MatchTag	$\overline{\texttt{.*<embed[^<>]*>.*}} \cap \overline{\texttt{.*</embed[^<>]*>.*}}$

Fig. 20. Rules in RHS for modeling XSS attack

The unfiltered input could lead to XSS (a taint analysis [NTGG+05] could identify the vulnerability). However, to precisely craft a working exploit is still not a trivial job, as several constraints have to be satisfied before the injected JavaScript code could work. For example, the injected JavaScript tag should not be contained in the value of an HTML attribute (otherwise it will not be executed). In addition, the resulting HTML should remain syntactically correct, at least until the parser reaches the injected JavaScript code. To simply insert a JavaScript function will not work.

SUSHI can help generating the attack string precisely. In fact, SUSHI generates the following attack string which is, first of all *working*, and is *shorter* (if not the shortest) than the exploit given in the original securitytracker post [Lab09].

```
\"<script>alert('XSS found!')</script>
```

In the following, we briefly describe how the SISE equation is constructed for generating the exploit. The LHS of the equation is a concatenation of constant words (generated by a manual simulation of symbolic execution on the target program), and the unsanitized user input. The RHS is a conjunction of a number of attack patterns and filter rules as shown in Fig. 20. They are reusable patterns for characterizing a working XSS exploit.

1. The `XSS` pattern in Fig. 20 requires that the JavaScript `alert()` function eventually shows up in the combined output.
2. the `EffectiveScript` rule forbids the JavaScript snippet to be embedded in any HTML attribute definition (thus ineffective).
3. The `MatchTag` rule requires that an HTML beginning tag must be matched by an ending tag (in the example, it is the "`<embed>`" tag).

Clearly, these SISE constraints used for specifying XSS attacks are reusable.

9. Related work

String analysis, i.e., analyzing the set of strings that could be produced by a program, emerged as a novel technique for analyzing web applications, e.g., compatibility check of XHTML [CMS03a], security vulnerability scanning [FLP+07, GSD04], and web application verification [YBI09, BTV09].

In general, there are two interesting directions of string analysis: (1) *forward analysis*, which computes the image (or its approximation) of the program states as constraints on strings and other primitive data types; and (2) *backward analysis*, which usually starts from the negation of a property and computes backward. Most of the related work (e.g., [CMS03a, CMS03b, Min05, KM06, YBI09, BTV09]) belongs to forward analysis. The work presented in this paper is *backward*.

It is worthy of note that, unlike symbolic model checking on Boolean programs (e.g., using BDD) and integer programs (e.g., using Presburger arithmetic), where backward analysis can be easily leveraged from forward analysis via the use of existential quantification, there is a gap between the forward and backward image computations for strings. Concerning forward analysis, the main focus is on fixpoint computation (or approximation). For example, Christensen et al. [CMS03b] used the Mohri–Nederhof algorithm [MN01] to approximate from context-free to regular languages. Yu, Bultan, and Ibarra achieved forward fixpoint computation via widening technique for multi-tape automata [YBI09]. The backward analysis of string equation systems can be very different. Both forward and backward analyses have pros and cons. Forward analysis is able to discharge vulnerabilities, i.e., to prove a system is free of a certain vulnerability, due to the use of over-approximation. However, it might have

false positives, i.e., to report a vulnerability that actually does not exist. The backward string analysis, adopted by this research, is able to generate attack strings as hard-evidence for a vulnerability. However, it suffers from false-negatives, i.e., there are cases that vulnerabilities are ignored by the analysis. It will be interesting in our future work to combine the benefits of these two approaches.

SISE can be regarded as a variation of the *word equation* problem [Lot02]. A word equation $L = R$ is an equation where both L and R are concatenation of string variables and constants. Note that in a word equation, only word concatenation is allowed. In SISE, various popular `java.regex` operations are supported. This determines that the traditional technique for solving word equations, such as the Makanin's algorithm [Mak77], cannot be applied here. Concerning complexity, it is proved by Makanin that the word equation problem is decidable and NP-hard [Mak77]. However, extension of word equations can easily lead to undecidability. For example, the $\forall\exists^3$-theory of concatenation (according to [Lot02]) is known to be undecidable. SISE is decidable by imposing syntactic restrictions on the occurrence of variables.

There are recently several emerging backward string constraint solvers. The closest work to ours is the HAMPI string constraint solver [KGG+09], which also supports a backward analysis. HAMPI solves string constraints with context-free components, which are essentially unfolded to regular language within a certain bound. HAMPI, however, does not support string replacement, nor does it support regular replacement, which limits its ability to reason about sanitization procedures. In addition, the unfolding of context-free components limits its scalability. Similarly, Hooimeijer and Weimer's work [HW09] in the decision procedure for regular constraints does not support regular replacement. Another close work to ours is Yu's automata based forward/backward string analysis [YAB09]. Yu et al. use language based replacement [YBCI08] to handle regular replacement. Imprecision is introduced in the over-approximation during the language based replacement. Conversely, our analysis considers the delicate differences among the typical regular replacement semantics. This allows further reduction of false negatives, as shown in Sect. 2. The BEK solver [HLM+11] supports an imperative language for modeling string sanitization functions, and relies on symbolic finite transducers. The difference is that BEK intends to model the character level operations directly, while SUSHI concentrates on the high level regular replacement operations.

Excluding Makanin's algorithm, there are two popular methodologies taken by recent string constraint solvers: (1) bit-blasting, taken by e.g., [KGG+09, BTV09, SAH+10]; and (2) automata modeling, adopted by this work and [YBCI08, YAB09, YBI09]. The basic idea of bit-blasting is to translate a string equation into a SAT problem, by modeling a string as a vector of bits. Many string operations, such as indexing, substring extraction, and concatenation, can be modeled easily. Concerning length predicate (which is not modeled in this work), the bit-blasting approach has a much more straightforward modeling. However, capturing length predicate is possible in automata and has been shown to work in [YBCI08]. On the contrary, string replacement (especially precise modeling of various regular replacement) is more convenient using automata. It is interesting to note that, as shown by Veanes, Bjørner, and de Moura, finite and push-down automata can be modeled using SMT [VBdM10] and hence be used for solving string constraints. This may bridge the gap between the two approaches.

Finite state transducers are the major modeling tool used for handling regular replacement in this paper. This is mainly inspired by [KK94, Moh97, KCGS96] in computational linguistics, for processing phonological and morphological rules. In [KCGS96], an informal discussion was given for the semantics of left-most longest matching of string replacement. This paper has given the formal definition of replacement semantics and has considered the case where an empty word is included in the regular search pattern. Compared with [KK94] where FST is used for processing phonological rules, our approach is lighter given that we do not need to consider the left and right context of re-writing rules in [KK94]. Thus more DFST can be used, which certainly has advantages over NFST, because DFST is less expressive. For example, in modeling the reluctant semantics, compared with [KK94], our algorithm does not have to non-deterministically insert begin markers and it does not need extra filters, thus is more efficient. In practice, SUSHI uses a symbolic representation of finite state transducers, which speeds up its operations. Hooimeijer and Veanes recently find that Binary Decision Diagrams (BDD) can further improve the performance of symbolic automata in string analysis [HV11]. There are other ways of handling regular replacement. For example, Minamide in [Min05] shows how to construct a context-free grammar to model the approximation of regular replacement with backreferences. During the analysis, Minamide uses a finite state transducer to model a regular replacement without backreferences first. Then by tracking the non-terminals of the context-free grammar, Minamide shows that it is possible to handle backreferences. However, the analysis is imprecise and it does not distinguish the various semantics such as the left-most, greedy and reluctant matching. The problem of regular replacement is closely related to regular substring matching, and

the type-theoretic axiomatization of regular expressions can be applied [HN11], e.g., for handling backreferences and named capturing groups which are currently not supported by SUSHI. It is worthy of note that variations of finite state transducers have potentials in this area. Recently, Alur and Černý propose in [Av11] that a stream transducer, equipped with a finite collection of string variables and guarded transitions, can perform several interesting actions such as string reverse, and the functional equivalence of such transducers is decidable.

SUSHI is a component of a more general framework called SAFELI, proposed in [FLP+07, FQ08], for automatically discovering command injection attacks on web applications. The basic idea is to rely on symbolic execution [Kin76] to generate string constraints. Then the constraints are solved by SUSHI and the attack is replayed. Symbolic execution is out of the scope of this paper. We refer interested readers to JavaSye [FQ08], a primitive symbolic execution engine we developed for the Java platform. Note that JavaSye is not the only choice of front-end symbolic execution engines for SUSHI. There are many off-the-shelf tools available, e.g., Java PathFinder and its derivatives [BHPV00], and Symstra [XMSN05]. It is possible to integrate SUSHI with these tools.

Solving string constraints is one of the many directions for tackling command injection attacks. Taint analysis [NTGG+05] tracks the source of data and stops injection attempts at run time. A variety of intrusion detection approaches are developed based on forward string analysis [CMS03b] (see [GSD04] and [HO05]). Black-box testing (e.g., [HHLT03]) is more widely used with vulnerability scanners. SQL randomization [BK04] tracks SQL keywords using a revised SQL parser. Many of the aforementioned efforts are run-time protection, while this research intends to secure web applications in the development stage before they are deployed. In addition, the string constraint solving technique can be applied to defeating other existing and emerging attacks.

10. Conclusion

This paper introduces a general framework called string equation for modeling attack patterns. We show that a fragment called simple linear string equation (SISE) can be solved using an approach based on automata. Finite state transducer is used for precisely modeling several different semantics of regular substitution. The SUSHI constraint solver is implemented and it is applied to analyzing security of web applications. The experimental results show that SUSHI works efficiently in practice. Future directions include expanding the solver to consider context-free components and incorporating temporal logic operators. We also plan to integrate SUSHI with APOGEE [FQP+08], an automated web programming grading tool, which is used to increase the awareness of web security in computer science education.

Acknowledgements

We would like to thank Fang Yu, Tevfik Bultan, and Oscar Ibarra for the discussion that inspired this research. We are grateful to the anonymous reviewers for their constructive comments on the experimental evaluation of SUSHI. Prateek Saxena has provided great help in answering our questions on Kaluza. This paper is based upon the work supported by the National Science Foundation under grants NSF-DUE 0836859 and 0837020.

Appendix A. Regular replacement with ϵ in search pattern

Replacement with ϵ in search pattern is often counter intuitive. For example, `"a".replaceAll("a*?","b")` in Java yields bab.[18] It is valuable to formally model such rare cases of regular replacement, because it helps to perform a thorough program analysis on sanitization procedures.

The procedural replacement algorithm, intuitively, works in a loop. It inspects every index (including the one after the last character of the input word). At each index, it looks for the match of the search pattern, and it performs the replacement if there is any. For example, given input word a and search pattern $r = a^*$, $a^-_{r\to b}$ inspects indices 0 and 1, finds two matches of r (i.e., ϵ), and produces bab. We thus formally define the reluctant and greedy replacement semantics in Definition A.1 and A.6. It is tested that they are commensurate with the behaviors of `java.util.regex`.

[18] Here `*?` represents the reluctant semantics of the $*$ operator.

Definition A.1 Let $s, \omega \in \Sigma^*$ and $r \in R$ with $\epsilon \in r$.

$$s^{-}_{r\to\omega} = \begin{cases} \omega & \text{if } s = \epsilon. \\ \omega a \mu^{-}_{r\to\omega} \text{ where } s = a\mu \text{ and } a \in \Sigma & \text{otherwise.} \end{cases}$$

□

Example A.2 Let $s, \omega \in \Sigma^*$ and $r \in R$. Consider the following three cases, all of which are verified using `java.util.regex`. (i) If $s = aa$, $r = a^*$, and $\omega = c$, then $s^{-}_{r\to\omega} = cacac$. (ii) If $s = ba$, $r = a^*$, and $\omega = c$, then $s^{-}_{r\to\omega} = cbcac$. (iii) If $s = ba$, $r = b \,|\, a^*$, and $\omega = c$, then $s^{-}_{r\to\omega} = cbcac$.

Lemma A.3 *Let $\mu \in \Sigma^*$ and $r \in R$ with $\epsilon \in r$. η is the r-begin-marked output of μ iff $|\,\eta\,| = 2\,|\,\mu\,| + 1$, and $\forall_i\ 0 \le i < |\,\mu\,| : \eta[2i] = \#\ \wedge\ \eta[2i+1] = \mu[i]$, and $\eta[|\,\eta\,| - 1] = \#$.*

Lemma A.4 *Let $\mu, \eta, \omega \in \Sigma^*$, $n = |\,\mu\,|$, and $r \in R$ with $\epsilon \in r$. $\mu^{-}_{r\to\omega} = \eta$ iff η can be written as $\eta_1 \circ \cdots \circ \eta_n \circ \omega$ where $\forall_i\ 1 \le i \le n : \eta_i = \omega\,\mu[i-1]$.*

Similar to Lemma 5.22, we have the following lemma for the reluctant replacement transducer $\mathcal{A}_4(r, \omega)$ when $\epsilon \in r$. It can be inferred from Lemmas A.3 and A.4

Lemma A.5 *Given $r \in R$ with $\epsilon \in r$, and $\omega \in \Sigma^*$, for any r-begin-marked word $\kappa \in (\Sigma \cup \{\#\})^*$ and any word $\zeta \in \Sigma^*$: $(\kappa, \zeta) \in L(\mathcal{A}_4(r, \omega))$ iff $\kappa = \#\ \wedge\ \zeta = \omega$, or both of the following are true:*

- (F1′) *κ can be written as $\#a\mu$ such that $a \in \Sigma$.*
- (F2′) *ζ can be written as $\omega a \eta$, and (μ, η) is accepted by $\mathcal{A}_4(r, \omega)$.*

The definition of the greedy replacement with ϵ in search pattern, and the related properties are listed in the following.

Definition A.6 Let $s, \omega \in \Sigma^*$ and $r \in R$ with $\epsilon \in r$. $s^{+}_{r\to\omega}$ is defined using one of the following three rules:

1. if $s = \epsilon$, then $s^{+}_{r\to\omega} = \omega$.
2. if $s \in (r - \epsilon)\Sigma^*$, then $s^{+}_{r\to\omega} = \omega\mu^{+}_{r\to\omega}$, where $s = \beta\mu$ s.t. $\beta \in r - \epsilon$, and $\forall_{m,n\ s.t.\ \mu=mn}\ m \neq \epsilon \Rightarrow \beta m \notin r$.
3. if $s \neq \epsilon\ \wedge\ s \notin (r - \epsilon)\Sigma^*$, then $s^{+}_{r\to\omega} = \omega a \mu^{+}_{r\to\omega}$, where $s = a\mu$. □

Example A.7 Let $s, \omega \in \Sigma^*$ and $r \in R$. Consider the following three cases. (i) If $s = aa$, $r = a^*$, and $\omega = c$, then $s^{+}_{r\to\omega} = cc$. (ii) If $s = ba$, $r = a^*$, and $\omega = c$, then $s^{+}_{r\to\omega} = cbcc$. (iii) If $s = ba$, $r = b \,|\, a^*$, and $\omega = c$, then $s^{+}_{r\to\omega} = ccc$.

For $\epsilon \in r$, the notion of r-greedy-marking is defined below.

Definition A.8 Let $r \in R$ with $\epsilon \in r$, for any $\kappa \in \Sigma^*$ and $\zeta \in \Sigma_2^*$, ζ is said to be a *r-greedy-marking* of κ iff one of the following three conditions is satisfied:

1. If $\kappa = \epsilon$, then $\zeta = \#\$$.
2. If $\kappa \in (r-\epsilon)\Sigma^*$, then let $\kappa = \beta\mu$ s.t. $\beta \in r - \epsilon$ and $\forall_{m,n\ s.t.\ \mu=mn}\ m \neq \epsilon \Rightarrow \beta m \notin r - \epsilon$. Let η be a r-greedy marking of μ. Then $\zeta = \#\beta\$\eta$.
3. If $\kappa \neq \epsilon\ \wedge\ \kappa \notin (r - \epsilon)\Sigma^*$, let $\kappa = a\mu$ where $a \in \Sigma$, and η be a r-greedy marking of μ. Then $\zeta = \#\$a\eta$. □

Appendix B. Complete proofs in section 5

Appendix B.1. Proof of Lemma 5.4

Lemma 5.4 For any $r \in R$, $\mu \in \Sigma^*$ and $\eta \in (\Sigma \cup \{\$\})^*$, $(\mu, \eta) \in L(\mathcal{A}_1(r))$ iff all of the following are satisfied:

1. $\mu \in L(r)$, $|\eta| > 0$, and $\eta[|\eta| - 1] = \$$; and,
2. $\mu = \pi_\Sigma(\eta)$; and,
3. for each $0 \leq i < |\eta|$: $\eta[i] = \$$ iff (i) $\pi_\Sigma(\eta[0, i]) \in L(r)$, and (ii) when $i > 0$, $\eta[i-1] \neq \$$.

Proof. (Direction $\Rightarrow$) If $(\mu, \eta) \in L(\mathcal{A}_1(r))$, condition (1) can be inferred by converting an acceptance run of (μ, η) on $\mathcal{A}_1$ to an acceptance run of μ on DFSA(r). Condition (2) results from the fact that in $\mathcal{A}_1(r)$ only the $(\epsilon, \$)$ transitions (from f to f') generate \$ on the output tape.

The proof of condition (3) is shown as below. We first prove the $\Rightarrow$ direction. If $\eta[i] = \$$, by simulating a partial run (similarly to the argument of condition (1)), one can argue that $\pi_\Sigma(\eta[0, i])$ is accepted by DFSA(r). Then $\eta[i-1] \neq \$$ (no consecutive \$) is based on the fact that there could not exist two consecutive $(\epsilon, \$)$ transitions in $\mathcal{A}_1(r)$. This is because the destination state of a $(\epsilon, \$)$ transition in $\mathcal{A}_1$ (i.e., f' in the algorithm) will never be a final state in DFSA(r) (i.e., the source state of another $(\epsilon, \$)$ transition in $\mathcal{A}_1(r)$).

Now we prove the $\Leftarrow$ direction of condition (3). When both (i) and (ii) are true, it can be shown that $\eta[i-1]$ is a letter in Σ, and the partial run of $(\pi_\Sigma(\eta[0, i]), \eta[0, i])$[19] reaches a final state f in DFSA(r). From the construction algorithm, it is clear that f has exactly one transition, and it is $(\epsilon, \$)$. This proves that $\eta[i]$ is \$.

(Direction $\Leftarrow$) We can prove the following proposition (A1) first.

A1: Let $\mathcal{A}_1 = (\Sigma \cup \{\$\}, Q_1, q_0^1, F_1, \delta_1)$, and q_0^0 be the initial state of DFSA(r). If (μ, η) satisfies conditions 2, 3 and $\mu \in PREFIX(r)$, then there exists $q_j^1 \in Q_1$ s.t. $q_0^1 \rightsquigarrow^*_{(\mu,\eta)} q_j^1$ in $\mathcal{A}_1(r)$. If $\eta[|\eta| - 1] = \$$, then $q_j^1 \in F_1$; otherwise, q_j^1 is the state imported from DFSA(r) in the construction algorithm of $\mathcal{A}_1$, and $q_0^0 \rightsquigarrow^*_\mu q_j^1$ in DFSA(r).

A1 can be proved by induction on the length of η. The key of the proof is based on the fact that there are no consecutive \$ signs (from condition 3(ii)).

From A1, we can infer that if conditions 1,2, and 3 are satisfied, (μ, η) is accepted by $\mathcal{A}_1$. This completes the proof of Direction $\Leftarrow$. □

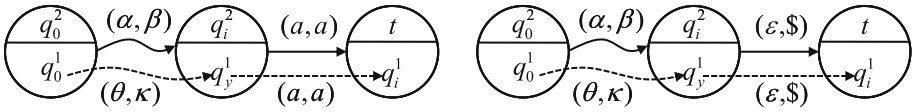
Appendix B.2. Proof of Lemma 5.7

Lemma 5.7 For any $r \in R$ and $\mu \in \Sigma^*$ there exists one and only one r-end-marked output of μ.

Proof. We prove by induction on the length of μ. For the base case, when $|\mu| = 0$, the lemma holds vacuously. For the inductive step, assume that $\mu = \nu \circ a$ (where $a \in \Sigma$), and ζ is the r-end-marked output of ν. Then, we construct μ's r-end-marked output (let it be η) as follows. If $\mu \in \Sigma^* r$, then $\eta = \zeta\, a\$$; otherwise $\eta = \zeta\, a$.

Now we prove by contradiction that η is the *only one* r-end-marked output for μ. Assume that there is another r-end-marked output of μ (letting it be η'). It follows from Definition 5.6 that $\pi_\Sigma(\eta) = \pi_\Sigma(\eta') = \mu$. Let ω be the *longest* common prefix of η and η', i.e., there exists ω_1 and ω_2 s.t. $\eta = \omega\omega_1$ and $\eta' = \omega\omega_2$. Let $n = |\omega|$. Now think about $\eta[n]$, i.e., the first letter of ω_1. There are two cases to consider: (i) If $\eta[n] \in \Sigma$, it follows that $\eta'[n]$ has to be the same as $\eta[n]$ if $\eta'[n] \in \Sigma$ (otherwise $\pi_\Sigma(\eta) \neq \pi_\Sigma(\eta')$). However, when $\eta[n] = \eta'[n]$, it conflicts with the assumption that ω is the *longest* common prefix. (ii) If $\eta[n] = \$$, according to condition (2) of Definition 5.6, $\eta[n-1] \neq \$$ and $\pi_\Sigma(\omega) \in \Sigma^* r$. It leads to $\eta'[n] = \$$, which again conflicts with the assumption on ω. This completes the proof. □

[19] Note that $\eta[i-1]$ is the last letter of $\eta[0, i]$.



(a) **(b)**

Fig. 21. Proof idea of Lemma 5.10

Appendix B.3. Proof of Lemma 5.10

Lemma 5.10 For any $r \in R$, let $\mathcal{A}_1(r)$ be $(\Sigma \cup \{\$\}, Q_1, q_0^1, F_1, \delta_1)$, and $\mathcal{A}_2(r)$ be $(\Sigma \cup \{\$\}, Q_2, q_0^2, F_2, \delta_2)$. For any $t \in Q_2$, any $\mu \in \Sigma^*$, and any $\eta \in (\sigma \cup \{\$\})^*$ s.t. $q_0^2 \rightsquigarrow^*_{(\mu,\eta)} t$ in $\mathcal{A}_2(r)$, both of the following are true:

1. For any $q_i^1 \in \mathcal{B}(t)$, there exists $\nu \dot{\prec} \mu$ and $\zeta \in (\Sigma \cup \{\$\})^*$ s.t. $q_0^1 \rightsquigarrow^*_{(\nu,\zeta)} q_i^1$ in $\mathcal{A}_1(r)$.
2. For any $\nu \dot{\prec} \mu$ s.t. $\nu \in PREFIX(r)$, there exists $\zeta \in (\Sigma \cup \{\$\})^*$ and $q_i^1 \in \mathcal{B}(t)$ s.t. $q_0^1 \rightsquigarrow^*_{(\nu,\zeta)} q_i^1$ in $\mathcal{A}_1(r)$.

We prove by induction on the length of η. The base case vacuously holds for both statements, because $\mathcal{B}(q_0^2) = \{q_0^1\}$ and $q_0^1 \rightsquigarrow^*_{(\epsilon,\epsilon)} q_0^1$ in $\mathcal{A}_1(r)$.

Proof of Statement 1: Given q_i^1 in $\mathcal{B}(t)$, the objective is to find the pair (ν, ζ) that reaches q_i^1 in $\mathcal{A}_1(r)$. The proof idea is presented in Fig. 21a.

Consider the last transition in the partial run of (μ, η) on $\mathcal{A}_2(r)$. If it is a Case 1 transition, let it be $(q_i^2, t, a : a)$. We can safely write μ and η as $\mu = \alpha a$, and $\eta = \beta a$. Now by case 1 Rule, for q_i^1 there must be a $q_y^1 \in \mathcal{B}(q_i^2)$ s.t. $(q_y^1, q_i^1, a : a) \in \delta_1$.[20] Since $q_0^2 \rightsquigarrow^*_{(\alpha,\beta)} q_i^2$ in $\mathcal{A}_2(r)$, the induction assumption applies, i.e., we can find a pair (θ, κ) s.t. $q_0^1 \rightsquigarrow^*_{(\theta,\kappa)} q_y^1$. Now $(\theta a, \kappa a)$ is what we need because $q_0^1 \rightsquigarrow^*_{(\theta,\kappa)} q_y^1 \rightsquigarrow_{(a,a)} q_i^1$. Hence, $q_0^1 \rightsquigarrow^*_{(\theta a,\kappa a)} q_i^1$. Similar argument applies to Case 2.

Proof of Statement 2: The key is to find the ζ and q_i^1 in the statement.

If the last element of η is \$ (Case 2), let $\eta = \beta\$$. We can infer that there exists $q_i^2 \in Q_2$ s.t. $q_0^2 \rightsquigarrow^*_{(\alpha,\beta)} q_i^2 \rightsquigarrow_{(\epsilon,\$)} t$. The scheme of the proof is shown in Fig. 21b. Given $\nu \dot{\prec} \alpha$, by applying the induction assumption we now have a $q_y^1 \in \mathcal{B}(q_i^2)$ and $\kappa \in (\Sigma \cup \{\$\})^*$ s.t. $q_0^1 \rightsquigarrow^*_{(\nu,\kappa)} q_y^1$ in $\mathcal{A}_1(r)$. Now, if $q_y^1 \notin F_0$[21], according to Case 2 Rule of $\mathcal{A}_2(r)$, it is also contained in $\mathcal{B}(t)$. Then κ and q_y^1 already satisfy our needs (i.e., κ is the ζ and q_y^1 is the q_i^1 we are trying to find for statement 2). If $q_y^1 \in F_0$, then there must be a q_i^1 s.t. $(q_y^1, q_i^1, \epsilon : \$) \in \delta_1$. In this case, $\kappa\$$ and q_i^1 satisfy our needs for the proof, i.e., $q_0^1 \rightsquigarrow^*_{(\nu,\kappa\$)} q_i^1$ and $q_i^1 \in \mathcal{B}(t)$.

When $\eta[|\eta| - 1] \in \Sigma$ (Case 1), the proof is more complex and it needs the fact that ν is a prefix of a word in r. Let $\mu = \alpha a$ and $\eta = \beta a$, similarly we have $q_i^2 \in Q_2$ s.t. $q_0^2 \rightsquigarrow^*_{(\alpha,\beta)} q_i^2 \rightsquigarrow_{(a,a)} t$ in $\mathcal{A}_2(r)$. We use the notations in Fig. 21a in the following proof.

For any $\nu \dot{\prec} \mu$ (which is a prefix of a word in r), we can write it as $\nu = \theta a$. By the induction assumption, there exists $q_y^1 \in \mathcal{B}(q_i^2)$ and $\kappa \in (\Sigma \cup \{\$\})^*$ s.t. $q_0^1 \rightsquigarrow^*_{(\theta,\kappa)} q_y^1$. Now since ν is a prefix of a word in r, θa has a partial run on DFSA(r), and the partial run of θ is a prefix of it. Thus, let q_0^0 and δ_0 be the initial state and transition relation of DFSA(r), there exists q_k^0, q_i^0 s.t. $q_0^0 \rightsquigarrow^*_\theta q_k^0 \rightsquigarrow_a q_i^0$ in DFSA(r). Now the question is: would q_k^0 be the same as q_y^1? By the construction algorithm of $\mathcal{A}_1(r)$, if $\theta \notin r$, we can conclude that q_y^1 is the q_k^0 during the construction of $\mathcal{A}_1(r)$. If $\theta \in r$, we can infer that $q_k^0 \in F_0$, and q_y^1 is the corresponding final state in F_1. (q_y^1 cannot be contained in F_0, otherwise the state q_i^2 would not have an out-going (a, a), according to the Case 2 Rule, in $\mathcal{A}_2(r)$.) Thus, there is a transition $(a : a)$ from q_y^1 to q_i^0 in $\mathcal{A}_1(r)$. Note that q_i^0, a state in DFSA(r), is imported to the state set of $\mathcal{A}_1(r)$ according to the construction algorithm. In the following, we write it as q_i^1 to indicate that it is a state in $\mathcal{A}_1(r)$. Finally, we have $q_0^1 \rightsquigarrow^*_{(\theta,\kappa)} q_y^1 \rightsquigarrow_{(a,a)} q_i^1$, and $q_i^1 \in \mathcal{B}(t)$. This completes the proof of statement 2. □

[20] We only consider the case that q_i^1 is not the initial state q_0^1 of $\mathcal{A}_1(r)$. When q_i^1 is q_0^1, readers can verify that the claim vacuously holds.

[21] F_0 is the set of final states of DFSA(r).

Appendix B.4. Proof of Lemma 5.11

Lemma 5.11 For any $r \in R$, any $\mu \in \Sigma^*$, and any $\eta \in (\Sigma \cup \{\$\})^*$: $(\mu, \eta) \in L(\mathcal{A}_2(r))$ iff η is the r-end-marked output of μ.

Proof The proof by induction will be split into two parts (one for each direction). We assume that when $|\mu| + |\eta| \leq n$, $(\mu, \eta) \in L(\mathcal{A}_2(r))$ iff η is the r-end-marked output of μ. The base case for $\epsilon \notin r$, i.e., $\mu = \eta = \epsilon$ vacuously holds, because $(\epsilon, \epsilon) \in L(\mathcal{A}_2(r))$. The base case for $\epsilon \in r$, i.e., $\mu = \epsilon$ and $\eta = \$$, also holds as the word pair is accepted by $\mathcal{A}_2(r)$.

(*Direction* $\Leftarrow$*:*) We show that *if* η *is the* r*-end-marked output of* μ*, then* $(\mu, \eta) \in L(\mathcal{A}_2(r))$.

For the inductive step, similar to the proof of Lemma 5.10, we need to consider two cases, based on the last element of η. We consider one case here (when $\eta[|\eta| - 1] \in \Sigma$) and the proof of the other uses the same technique.

Let $\eta = \beta a$ (where $a \in \Sigma$). Because βa is the r-end-marked output of μ, by Definition 5.6, $\pi_\Sigma(\beta a) = \mu$. Thus the last element of μ is a, and let $\mu = \alpha a$. We can further infer that β is the r-end-marked output of α. This leads to $(\alpha, \beta) \in L(\mathcal{A}_2(r))$, using the inductive assumption. Therefore, there exists $q_i^2 \in F_2$ s.t. $q_0^2 \rightsquigarrow^*_{(\alpha,\beta)} q_i^2$ in $\mathcal{A}_2(r)$.

We now prove that $\mathcal{B}(q_i^2)$ does not contain any element in F_0 by contradiction. Assume $\mathcal{B}(q_i^2) \cap F_0 \neq \emptyset$, according to Case 2 Rules, there is an $(\epsilon, \$)$ transition that leads to a final state of $\mathcal{A}_2(r)$. This implies that $(\alpha, \beta\$) \in L(\mathcal{A}_2(r))$. Now by the induction (on the combined length of α and β and using the $\Rightarrow$ direction), $\beta\$$ is the r-end-marked output of α. This contradicts with the fact that βa is the r-end-marked output of αa, as on the index $|\beta|$ of η, once the sufficient and necessary conditions are satisfied (i.e., $\alpha \in \Sigma^* r$ and $\beta[|\beta| - 1] \neq \$$), the element can only be \$ (and cannot be a). Thus we have $\mathcal{B}(q_i^2) \cap F_0 = \emptyset$.

Now since $\mathcal{B}(q_i^2) \cap F_0 = \emptyset$, Case 1 Rule applies. There must exist a $t \in Q_2$ s.t. $(q_i^2, t, a : a) \in \delta_2$. Note that we still need to prove that $t \in F_2$ (i.e., $\mathcal{B}(t) \cap F_0 = \emptyset$). This could be achieved using contradiction (showing that η is not the r-end-marked output of μ because another \$ is needed at the end of η, using Lemma 5.10). In summary, we have constructed the following acceptance run for $(\mu, \eta) = (\alpha a, \beta a)$:

$$q_0^2 \rightsquigarrow^*_{(\alpha,\beta)} q_i^2 \rightsquigarrow_{(a,a)} t$$

Thus, $(\mu, \eta) \in L(\mathcal{A}_2(r))$. The case (when $\eta[|\eta| - 1] = \$$) can be proved similarly.

(*Direction* $\Rightarrow$*:*) We show that *if* $(\mu, \eta) \in L(\mathcal{A}_2(r))$ *then* η *is the r-end-marked output of* μ.

According to Definition 5.6, we need to prove that η satisfies the following four conditions: *(c1)* $\pi_\Sigma(\eta) = \mu$; and, *(c2)* $\forall_{0 \leq i < |\eta|}$, if $\pi_\Sigma(\eta[0, i]) \in \Sigma^* r$ and $\eta[i-1] \neq \$$ (when $i > 0$), then $\eta[i] = \$$; and, *(c3)* $\forall_{0 \leq i < |\eta|}$, if $\eta[i] = \$$, then $\pi_\Sigma(\eta[0, i]) \in \Sigma^* r$ and $\eta[i-1] \neq \$$ (when $i > 0$); and, *(c4)* if $\pi_\Sigma(\eta) \in r$, then $\eta[|\eta| - 1] = \$$.

(c1) is immediately available based on the fact that there are only two types of transitions in $\mathcal{A}_2(r)$: (b, b) for $b \in \Sigma$, and $(\epsilon, \$)$.

(c2) is established as follows. Consider $(\pi_\Sigma(\eta[0, i]), \eta[0, i])$, which is a prefix of (μ, η). Since $\mathcal{A}_2(r)$ is a DFST, there exists a partial run of $(\pi_\Sigma(\eta[0, i]), \eta[0, i])$, which is the prefix of the acceptance run for (μ, η). Thus, there is a $q_i^2 \in Q_2$ s.t. $q_0^2 \rightsquigarrow^*_{(\pi_\Sigma(\eta[0,i]),\eta[0,i])} q_i^2$ in $\mathcal{A}_2(r)$.

Now using Lemma 5.10(2), we can infer that there exists $q_j^1 \in \mathcal{B}(q_i^2)$, and $\nu \in r$ s.t. $\nu \doteq \pi_\Sigma(\eta[0, i])$ (this is possible because $\pi_\Sigma(\eta[0, i]) \in \Sigma^* r$), and $\zeta \in (\Sigma \cup \{\$\})^*$ s.t. $q_0^1 \rightsquigarrow^*_{(\nu,\zeta)} q_j^1$ in $\mathcal{A}_1(r)$. By the construction algorithm of $\mathcal{A}_1(r)$, we can infer that q_j^1 is either in F_0 or F_1. We now refute the possibility that $q_j^1 \in F_1$, because that would imply that the last transition taken in the run of (ν, ζ) on $\mathcal{A}_1(r)$ would be $(\epsilon, \$)$. This conflicts with the fact that $\eta[i-1] \neq \$$.

Now since $q_j^1 \in F_0 \cap \mathcal{B}(q_i^2)$, the only transition out of q_i^2 is $(\epsilon, \$)$. This concludes the proof for $\eta[i] = \$$.

(c3) is proved similarly as (c2). We first show that there exists a run $q_0^2 \rightsquigarrow^*_{(\pi_\Sigma(\eta[0,i]),\eta[0,i])} q_i^2 \rightsquigarrow_{(\epsilon,\$)} q_j^2$. Then using Lemma 5.10(1), we show that $\pi_\Sigma(\eta[0, i]) \in \Sigma^* r$. The rest of the proof is to show that there will never be two consecutive $(\epsilon, \$)$ transitions in $\mathcal{A}_2(r)$ (which leads to $\eta[i-1] \neq \$$). This can be inferred from the fact that for any $t \in F_2 : \mathcal{B}(t) \cap F_0 = \emptyset$. Thus, the destination state of an $(\epsilon, \$)$ transition cannot be the source state of another $(\epsilon, \$)$ transition in $\mathcal{A}_2(r)$.

For (c4), the proof is very similar to that of (c2) and we omit the details here. □

Appendix B.5. Proof of Theorem 5.23

Theorem 5.23 Given any $r \in R$ and $\omega \in \Sigma^*$, and let $\mathcal{M}^-_{r\to\omega}$ be $\mathcal{A}_3(r) \mid\mid \mathcal{A}_4(r,\omega)$, then for any $\kappa, \zeta \in \Sigma^*$: $(\kappa,\zeta) \in L(\mathcal{M}^-_{r\to\omega})$ iff $\zeta = \kappa^-_{r\to\omega}$.

Proof We prove the case that $\epsilon \notin r$. The $\epsilon \in r$ case can be proved using the same technique, using Lemmas A.3, A.4, and A.5.

(*Direction* $\Rightarrow$): The proof goal is *if* $(\kappa,\zeta) \in L(\mathcal{M}^-_{r\to\omega})$ *then* $\zeta = \kappa^-_{r\to\omega}$. The base case where $\kappa \in \Sigma^* - \Sigma^* r \Sigma^*$ is obvious. We now study the case where κ has at least one match of r. We prove by induction on the number of matches of r in κ.

According to Definition 4.2, there exists θ s.t. $(\kappa,\theta) \in L(\mathcal{A}_3(r))$ and $(\theta,\zeta) \in L(\mathcal{A}_4(r,\omega))$. By Lemma 5.15, θ must be the r-begin-marked output of κ, thus $\pi_\Sigma(\theta) = \kappa$. Then by applying Lemma 5.22, we have θ and ζ satisfying conditions F1 and F2, i.e., they can be written as:

$$\theta = \nu\#\beta\,\mu, \zeta = \nu\#\omega\eta$$

Here β, μ, and η satisfy conditions F1 and F2, e.g., $\beta \in \texttt{reluc}(r_\#)$. Since $\pi(\theta) = \kappa$ and $\nu \in \Sigma^* - \Sigma^* r \Sigma^*$, κ could also be written as

$$\kappa = \nu\pi_\Sigma(\beta)\pi_\Sigma(\mu) \tag{6}$$

Now we try to prove $\kappa^-_{r\to\omega} = \zeta$ by showing that κ and ζ conform to the recursive case of Definition 3.2. In summary, we need to show:

1. $\nu \notin \Sigma^* r \Sigma^*$. This is already proved.
2. $\pi_\Sigma(\beta) \in r$. It follows from the fact that $\beta \in \texttt{reluc}(r_\#)$, by Lemma 5.22.
3. *for every* x, y, u, t, m, n *with* $\nu = xy$, $\pi_\Sigma(\beta) = ut$, and $\pi_\Sigma(\mu) = mn$: *(i) if* $y \neq \epsilon$ then $yu \notin r$ and $y\beta m \notin r$; *and (ii) if* $t \neq \epsilon$ *then* $u \notin r$. We show first that (ii) holds and (i) can be proved similarly. Due to Eq. 6, there exists x', y', u', t', m', n' s.t. $x = x'$, $y = y'$, $u = \pi_\Sigma(u')$, $t = \pi_\Sigma(t')$, $m = \pi_\Sigma(m')$, $n = \pi_\Sigma(n')$, $\nu = x'y'$, $\beta = u't'$, and $\mu = m'n'$. By F1 of Lemma 5.22, we have if $t' \neq \epsilon$, then $u' \notin r_\#$. This implies that if $t \neq \epsilon$, then $u \notin r$. To see why, consider the case when $\pi_\Sigma(t') \neq \epsilon$. This implies that $t' \neq \epsilon$, which leads to $u' \notin r_\#$. Thus $\pi_\Sigma(u') \notin r$. Hence, we have proved (ii), and (i) is proved similarly.
4. $\eta = \pi_\Sigma(\mu)^-_{r\to\omega}$. The proof relies on the induction assumption. Using Lemma 5.22 we have $(\mu,\eta) \in L(\mathcal{A}_4(r,\omega))$. Then, we just need to show $(\pi_\Sigma(\mu),\mu) \in L(\mathcal{A}_3(r))$ so that $(\pi_\Sigma(\mu),\eta) \in L(\mathcal{M}^-_{r\to\omega})$, which leads to $\eta = \pi_\Sigma(\mu)^-_{r\to\omega}$ using the induction assumption.
   We prove that μ is the r-begin-marked output of $\pi_\Sigma(\mu)$ (i.e., $(\pi_\Sigma(\mu),\mu) \in L(\mathcal{A}_3(r))$. We need to show that both E1 and E2 of Definition 5.14 hold. E1 (every # is followed by a match of r and no consecutive # signs) is implied by the fact that θ is r-begin-marked (where μ is its suffix). For E2, we need to show that $\mu[0] = \#$ if $\mu \in r\Sigma^*$. This can be proved by contradiction. Assume that $\mu[0] \neq \#$ but $\mu \in r\Sigma^*$. Since θ is r-begin-marked, it follows that the # before the match of r is the last element of β. Now this conflicts with $\beta \in \texttt{reluc}(r_\#)$, as stated in Lemma 5.22(F1), because $\beta[0, |\beta| - 1]$ (i.e., dropping the last #) is a shorter match of $r_\#$. Therefore, the assumption cannot be true and E2 holds.

(Direction $\Leftarrow$): We show that *if* $\kappa^-_{r\to\omega} = \zeta$ *then* $(\kappa,\zeta) \in L(\mathcal{M}^-_{r\to\omega})$.

Similarly, we concentrate on the case where there is at least one match of r in κ. Notice that there exists one and only one r-begin-marked output of κ, and let it be θ. According to Lemma 5.15, $(\kappa,\theta) \in L(\mathcal{A}_3(r))$. Thus, to show $(\kappa,\zeta) \in L(\mathcal{M}^-_{r\to\omega})$ we only need to prove $(\theta,\zeta) \in L(\mathcal{A}_4(r,\omega))$. This is accomplished via induction on the number of replacements performed on κ.

We now try to construct a run for (θ,ζ) on $\mathcal{A}_4(r,\omega)$. In the following, we study the relations between κ, θ, and ζ. According to Definition 3.2, κ and ζ can be written as $\kappa = \nu\beta\mu$ and $\zeta = \nu\omega\eta$ that satisfy the conditions which enforce β to be the earliest and reluctant match of r (i.e., $\beta \in r$, and for every x, y, u, t, m, n with $\nu = xy$, $\beta = ut$, and $\mu = mn$: (i) if $y \neq \epsilon$ then $yu \notin r$ and $y\beta m \notin r$; and (ii) if $t \neq \epsilon$ then $u \notin r$), and $\eta = \mu^-_{r\to\omega}$. We could write θ as the following form:

$$\theta = \nu\#\beta'\mu'$$

Such a form is possible—we only need to show that the first # of θ is right after ν. This is based on the following facts: (1) θ is the r-begin-marked output of κ, (2) $\nu \in \Sigma^* - \Sigma^* r \Sigma^*$, and (3) and β is the earliest match of r in κ. Now we require that β' is the shortest substring of θ starting at # s.t. $\beta = \pi_\Sigma(\beta')$. Due to this requirement, $\beta' \in \texttt{reluc}(r_\#)$.

We now construct the first portion of the acceptance run. Given $v \in \Sigma^* - \Sigma^* r \Sigma^*$, $\beta \in \texttt{reluc}(r)$, we have the following partial run for $(v\#\beta', v\omega)$ in $\mathcal{A}_4(r, \omega)$:

$$\gamma_1 : f_1^4 \rightsquigarrow^*_{(v,v)} f_1^4 \rightsquigarrow_{(\#,\epsilon)} s_1^4 \rightsquigarrow_{(\beta',\omega)} s_2^4 \rightsquigarrow_{(\epsilon,\epsilon)} f_1^4$$

Now by Definition 3.2, $\eta = \mu^-_{r\to\omega}$, and this leads to $(\mu, \eta) \in \mathcal{M}^-_{r\to\omega}$ by the induction assumption. Then consider μ' and η. Using an argument similar to (4) of Direction $\Rightarrow$, we can show that μ' is the r-begin-marked output of μ, thus $(\mu, \mu') \in L(\mathcal{A}_3(r))$. Therefore, we have $(\mu', \eta) \in \mathcal{A}_4(r, \omega)$ (using Definition 4.2). This leads to the following run in $\mathcal{A}_4(r, \omega)$:

$$\gamma_2 : f_1^4 \rightsquigarrow^*_{(\mu',\eta)} f_1^4$$

It follows that $\gamma_1 \circ \gamma_2$ is the acceptance run for (θ, ζ) on $\mathcal{A}_4(r, \omega)$. This concludes the proof for $(\kappa, \zeta) \in L(\mathcal{M}^-_{r\to\omega})$. □

Appendix B.6. Proof of Lemma 5.41

Lemma 5.41 For any $r \in R$, any $\kappa \in \Sigma^*$ and $\eta \in \Sigma_2^*$. η is the r-greedy-marking of κ iff $(\kappa, \eta) \in L(\mathcal{M}_r^+)$.

Proof (*Direction* $\Rightarrow$:) we first show that *if* η *is the* r*-greedy-marking of* κ *then* $(\kappa, \eta) \in L(\mathcal{M}_r^+)$. This results from the following two facts: (1) $(\kappa, \eta) \in L(\mathcal{A}_3(r) \,||\, \mathcal{A}_5(r) \,||\, \mathcal{A}_6)$ because $\mathcal{A}_5(r)$ optionally inserts a \$ after each match of r (so η is one of the output words). (2) (η, η) is accepted by all of the $\mathcal{A}_7(r)$, $\mathcal{A}_8(r)$, $\mathcal{A}_9(r)$ filters. This is because η is the r-greedy-marking and it will pass all the filters.

(*Direction* $\Leftarrow$:) The proof goal is *if* $(\kappa, \eta) \in L(\mathcal{M}_r^+)$, *then* η *is the* r*-greedy-marking of* κ. By Lemma 5.38 and directing $\Rightarrow$ we have: for any word $\alpha \in \Sigma^*$, there exists one and only one r-greedy marking (letting it be α'), and $(\alpha, \alpha') \in L(\mathcal{M}_r^+)$. Based on the above, We only need to show the following proposition (J1) is true for accomplishing the proof goal:

(J1) If both (κ, η) *and* (κ, η') *are accepted by* $\mathcal{M}_r^+$, *then* $\eta = \eta'$.

Since $\mathcal{M}_r^+$ is a composition of a sequence of transducers, and each transducer can be regarded as an input/output device. One can write the process of reaching η and η' as the following, using subscripts to indicate the output of a particular transducer.

$$\xrightarrow{\kappa} \mathcal{A}_3(r) \xrightarrow{\eta_3} \mathcal{A}_5(r) \xrightarrow{\eta_5} \mathcal{A}_6 \xrightarrow{\eta_6} \mathcal{A}_7(r) \xrightarrow{\eta_7} \mathcal{A}_8(r) \xrightarrow{\eta_8} \mathcal{A}_9(r) \xrightarrow{\eta}$$
$$\xrightarrow{\kappa} \mathcal{A}_3(r) \xrightarrow{\eta'_3} \mathcal{A}_5(r) \xrightarrow{\eta'_5} \mathcal{A}_6 \xrightarrow{\eta'_6} \mathcal{A}_7(r) \xrightarrow{\eta'_7} \mathcal{A}_8(r) \xrightarrow{\eta'_8} \mathcal{A}_9(r) \xrightarrow{\eta'}$$

Based on Lemma 5.16, one can infer that $\eta_3 = \eta'_3$. Since $\mathcal{A}_7(r)$, $\mathcal{A}_8(r)$, and $\mathcal{A}_9(r)$ are "filters" (each of them accepts a language which is a subset of the identity relation). We have: $\eta_6 = \eta_7 = \eta_8 = \eta$, and $\eta'_6 = \eta'_7 = \eta'_8 = \eta'$. Thus, if $\eta \neq \eta'$, they have to depart right after $\mathcal{A}_5(r)$ (which nondeterministically introduces end markers).

We prove J1 by contradiction. Assume that $\eta \neq \eta'$. Let index i be the first index that η differs from η', i.e., $\eta[i] \neq \eta'[i]$ and $\forall_x\, 0 \leq x < i : \eta[x] = \eta'[x]$. Then there are seven cases to discuss (note that the length of η may be different from that of η'):

- **Case 1** (ϵ, a)**:** $|\,\eta\,| = i$ and $\eta'[i] \in \Sigma$ (letting it be a); or symmetrically $|\,\eta'\,| = i$ and $\eta[i] \in \Sigma$.
- **Case 2** $(\epsilon, \#)$**:** $|\,\eta\,| = i$ and $\eta'[i] = \#$; or symmetrically $|\,\eta'\,| = i$ and $\eta[i] = \#$.
- **Case 3** $(\epsilon, \$)$**:** $|\,\eta\,| = i$ and $\eta'[i] = \$$; or symmetrically $|\,\eta'\,| = i$ and $\eta[i] = \$$.
- **Case 4** (a, b)**:** $\eta[i] \in \Sigma$ and $\eta'[i] \in \Sigma$, however $\eta[i] \neq \eta'[i]$.
- **Case 5** $(b, \#)$**:** $\eta[i] \in \Sigma$ and $\eta'[i] = \#$; or symmetrically $\eta'[i] \in \Sigma$ and $\eta[i] = \#$.
- **Case 6** $(a, \$)$**:** $\eta[i] \in \Sigma$ and $\eta'[i] = \$$; or symmetrically $\eta'[i] \in \Sigma$ and $\eta[i] = \$$.
- **Case 7** $(\#, \$)$**:** $\eta[i] = \#$ and $\eta'[i] = \$$; or symmetrically $\eta'[i] = \#$ and $\eta[i] = \$$.

We now show by contradiction that none of the above cases holds.

Cases 1 and 4: We can first discharge these two cases using Lemma 5.40. Take Case 4 (a, b) as an example: we have $\forall_x\, 0 \leq x < i : \eta[x] = \eta'[x]$, and $\eta[i], \eta'[i] \in \Sigma$, and $\eta[i] \neq \eta'[i]$. It follows that $\pi_\Sigma(\eta) \neq \pi_\Sigma(\eta')$. This contradicts with Lemma 5.40, which requires that both of $\pi_\Sigma(\eta)$ and $\pi_\Sigma(\eta')$ should be equal to κ.

Cases 3 and 7: As an example, we prove that Case 3 $(\epsilon, \$)$ is impossible. The assumption is that $|\,\eta\,| = i$ and $\eta'[i] = \$$. According to Lemma 5.27, $\mathcal{A}_6$ ensures that # and \$ markers are paired in η. This leads to the contradiction that the \$ at index i of η' does not have a corresponding # in η'. Case 7 can be proved using the same technique.

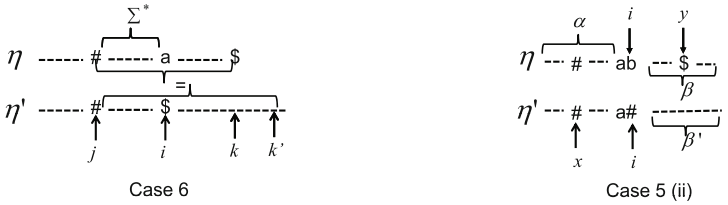


Fig. 22. Proof idea of Lemma 5.41

Cases 6: We show that Case 6 $(a, \$)$ is impossible. The idea of the proof is presented in Fig. 22.

The assumption is that $\eta[i] = a$ and $\eta'[i] = \$$, and $\forall_x\, 0 \leq x < i : \eta[x] = \eta'[x]$. By Lemma 5.27, in η' there is a preceding # to pair with $\eta'[i] = \$$, and let it be located at index j, and we also have $\eta'[j+1, i] \in \Sigma^*$. It follows that in η we also have $\eta[j] = \#$ and $\eta[j+1, i] \in \Sigma^*$. Now study η, the pairing \$ of the # at index j must be located at an index greater than i (and let it be k). Note that $\eta[j+1, k]$ captures the substring (match of r) between the marker pair at indices j and k. Now go back and observe η': by Lemma 5.38, there must exist $k' > i$ s.t. $\pi_\Sigma(\eta'[j+1, k']) = \eta[j+1, k]$, because $\eta[0, j] = \eta'[0, j]$ and $\pi_\Sigma(\eta) = \pi_\Sigma(\eta')$. Notice that $\eta'[j+1, i] \in \Sigma^*$ is a *strict* prefix of $\pi_\Sigma(\eta'[j+1, k'])$, and both of them are instances of r. Thus the \$ at index i of η' is improperly marked (for a shorter match), and this conflicts with the fact that η' passes longest match filter $\mathcal{A}_9(r)$. Thus case 6 could not hold.

Case 2: The proof has been provided in Sect. 5.2.9.

Case 5: We prove that Case 5 $(b, \#)$ does not hold. The assumption is that $\eta[i] = b$, $\eta'[i] = \#$, and $\eta[0, i] = \eta'[0, i]$.

Consider $\eta[i-1]$, there are three possibilities: (i) $\eta[i-1] = \#$, (ii) $\eta[i-1] \in \Sigma$, and (iii) $\eta[i-1] = \$$. We address these cases one by one.

- **(i):** $\eta[i-1] = \#$ could not be true, because it leads to $\eta'[i-1] = \#$. Given $\eta'[i] = \#$, this cannot hold because after passing $\mathcal{A}_6$ all # and \$ should be paired.
- **(ii):** Now we reject case (ii) by contradiction. The proof idea is shown in Fig. 22.
  Let $\eta[i-1] = a$ where $a \in \Sigma$. Based on the previous discussion, η and η' can be written as the following, where $\alpha, \beta, \beta' \in \Sigma_2^*$ (see Fig. 22):

  $$\eta = \alpha a b \beta,\ \eta' = \alpha a \# \beta'$$

  Then we trace the source of the characters a, b, and # in the above formula. Note that there might be multiple a's, b's, and #'s in η and η'. In our following discussion, we use a, b, and # to specifically refer to $\eta'[i-1]$ (i.e., a), $\eta[i]$ (i.e., b), and $\eta'[i]$ (i.e., #). We use indices a_i, b_i, $\#_i$ to indicate the index of the specific a, b, # in an intermediate output η_i. For example, $\eta_3[a_3]$ is the a in η_3, and $\#'_5$ indicates the index of the # character in η'_5. Using the fact $\pi_\Sigma(\eta) = \pi_\Sigma(\eta')$, we have the following observation:

  1. $a_3 = a'_3 < \#_3 = \#'_3 < b_3 = b'_3$. This results from the fact that $\eta_3 = \eta'_3$.
  2. $a_5 < \#_5 < b_5$ and $a'_5 < \#'_5 < b'_5$.
  3. $a_6 < b_6$ and $a'_6 < \#'_6 < b'_6$. Note that there does not exist $\#_6$, because it is consumed by $\mathcal{A}_6$.

  Now consider $\eta_5[a_5+1, b_5]$ (the substring of η_5 which starts from the next character after the a and ends at the letter before the b). First of all, it cannot contain any symbol from Σ (otherwise, a and b will not be neighbors in η). Second, it has to contain the # (see formula 2 above). Third, it cannot contain more than one #, because otherwise it conflicts with the fact that $\mathcal{A}_3(r)$ generates no consecutive # signs. The other choices are \$ signs, and they cannot appear consecutively. Thus $\eta_5[a_5+1, b_5]$ has only four choices: #, \$#, \$#\$, #\$.
  Notice that $\eta_5[\#_5]$ disappears from η_6, and the only way it could disappear is to traverse the $(q_1^6, q_1^6, \# : \epsilon)$ in $\mathcal{A}_6$. \$ cannot immediately precede #, because in $\mathcal{A}_6$ none of the transitions with \$ on the input tape reaches state q_1^6. Similarly, we refute the case of \$ right after # in $\eta_5[a_5+1, b_5]$, because the \$ will travel on $(q_1^6, q_0^6, \$: \$)$ and appear between a and b in η. Thus, the only option for $\eta_5[a_5+1, b_5]$ is #. This leads to the fact that $a\#b$ is a substring of η_5.

Let us observe the run of η_5 on $\mathcal{A}_6$. All of the a, #, b in $\eta_5[a_5, b_5 + 1]$ are taking self-loop transitions on q_1^6. So there must be one # before a for entering q_1^6, and one \$ after b for entering final state q_0^6. Let the # and \$ be the element x and y in η (see Fig. 22). It follows that $x < i - 1$ and $y > i$, and $\eta[x + 1, y] \in \Sigma^*$, which implies $\eta'[x, i] \in \#\Sigma^*$. Now for η', there are two # signs (at x and i), between which there is no \$ sign. This conflicts with Lemma 5.27 that markers should be paired.

- **(iii):** We finally refute the case that $\eta[i - 1] = \$$. In this case let $\eta = \alpha\$b\beta$ and $\eta' = \alpha\$\#\beta'$. Consider $\eta[i - 2]$: it cannot be #, otherwise $\eta'[i - 2] = \#$ and $\eta'[i] = \#$ would have been consecutive # signs in η'_3. Similarly, $\eta[i - 2] \neq \$$. Thus, the only choice is $\eta'[i - 2] \in \Sigma$ (and let it be c).
  Let $\alpha = \gamma c$. Then $\eta = \gamma c\$b\beta$ and $\eta' = \gamma c\$\#\beta'$. Now we argue that $\pi_\Sigma(b\beta) = \pi_\Sigma(\beta') \in r\Sigma^*$. This is because (1) $\pi_\Sigma(\eta) = \pi_\Sigma(\eta')$ and $\eta[0, i] = \eta'[0, i]$ imply $\pi_\Sigma(b\beta) = \pi_\Sigma(\beta')$; and (2) $\eta'[i] = \#$ and the # must precede a match of r.

Given that $L_2(1)$, the first filter expression in $\mathcal{A}_8(r)$, is defined as $\overline{\Sigma_2^*[\hat{}\#]\$\Sigma\Sigma_2^* \cap r_{\#,\$}\Sigma_2^*}$, we have $\eta \cap L_2(1) = \emptyset$. Intuitively, it means that the $b\beta$ in η is a match of r, however, it does not have a preceding # in η. This conflicts with the fact that η has already passed the filter $\mathcal{A}_8(r)$.

In summary, we have discharged all of the seven cases by contradiction. The proof is complete. □

References

[Anl02] Anley C (2002) Advanced SQL injection in SQL server applications. Next generation security software

[APV07] Anand S, Pasareanu CS, Visser W (2007) JPF-SE: a symbolic execution extension to Java pathfinder. In: Proceedings of the 13th international conference on tools and algorithms for construction and analysis of systems (TACAS), pp 134–138

[Av11] Alur R, Černý P (2011) Streaming transducers for algorithmic verification of single-pass list-processing programs. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), pp 599–610

[BHPV00] Brat G, Havelund K, Park S, Visser W (2000) Java path finder: second generation of a Java model checker. In: Workshop on advances in verification

[BK04] Boyd SW, Keromytis AD (2004) SQLrand: preventing SQL injection attacks. In: Proceedings of the 2nd applied cryptography and network security conference (ACNS). Lecture notes in computer science, vol 3089. Springer, pp 292–302

[BS88] Büchi JR, Senger S (1998) Definability in the existential theory of concatenation and undecidable extensions of this theory. Zeitschr f math Logik und Grundlagen d Math 34:337–342

[BTV09] Bjørner N, Tillmann N, Voronkov A (2009) Path feasibility analysis for string-manipulating programs. In: Proceedings of the 15th international conference on tools and algorithms for the construction and analysis of systems (TACAS). Springer, pp 307–321

[CF10] Chaudhuri A, Foster JS (2010) Symbolic security analysis of ruby-on-rails web applications. In: Proceedings of the 17th ACM conference on computer and communications security (CCS), pp 585–594

[CGP+06] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR (2006) EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM conference on computer and communications security (CCS), pp 322–335

[Chr06] Christey SM (2006) Dynamic evaluation vulnerabilities in PHP applications. http://seclists.org/fulldisclosure/2006/May/35

[CMS03a] Christensen AS, Møller A, Schwartzbach MI (2003) Extending Java for high-level web service construction. ACM Trans Program Lang Syst 25(6):814–875

[CMS03b] Christensen AS, Møller A, Schwartzbach MI (2003) Precise analysis of string expressions. In: Proceedings of the 10th international static analysis symposium (SAS), pp 1–18

[CPM+10] Caballero J, Poosankam P, McCamant S, Babic D, Song D (2010) Input generation via decomposition and re-stitching: finding bugs in Malware. In: Proceedings of the 17th ACM conference on computer and communications security (CCS), pp 413–425

[FL10a] Fu X, Li CC (2010) A string constraint solver for detecting web application vulnerability. In: Proceedings of the 22nd international conference on software engineering and knowledge engineering (SEKE), pp 535–542

[FL10b] Fu X, Li CC (2010) Modeling regular replacement for string constraint solving. In: Proceedings of the 2nd NASA formal methods symposium (NFM), pp 67–76

[FLP+07] Fu X, Lu X, Peltsverger B, Chen S, Qian K, Tao L (2007) A static analysis framework for detecting SQL injection vulnerabilities. In: Proceedings of 31st annual international computer software and applications conference (COMPSAC), pp 87–96

[FQ08] Fu X, Qian K (2008) SAFELI: SQL injection scanner using symbolic execution. In: Proceedings of the 2008 workshop on testing, analysis, and verification of web services and applications, pp 34–39

[FQP+08] Fu X, Qian K, Peltsverger B, Tao L, Liu J (2008) APOGEE: automated project grading and instant feedback system for web based computing. In: Proceedings of the 39th SIGCSE technical symposium on computer science education (SIGCSE), pp 77–81

[Fu09] Fu X (2009) SUSHI: a solver for single linear string equations. http://people.hofstra.edu/Xiang_Fu/XiangFu/projects.php

[GSD04] Gould C, Su Z, Devanbu PT (2004) JDBC checker: a static analysis tool for SQL/JDBC applications. In: Proceedings of the 26th international conference on software engineering (ICSE), pp 697–698

[HHLT03] Huang YW, Huang SK, Lin TP, Tsai CH (2003) Web application security assessment by fault injection and behavior monitoring. In: Proceedings of the 12th international world wide web conference (WWW), pp 148–159

[HLM+11] Hooimeijer P, Livshits B, Molnar D, Saxena P, Veanes M (2011) Fast and precise sanitizer analysis with BEK. In: Proceedings of the 20th USENIX security symposium (to appear)
[HN11] Henglein F, Nielsen L (2011) Regular expression containment: coinductive axiomatization and computational interpretation. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), pp 385–398
[HO05] Halfond W, Orso A (2005) AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering (ASE), pp 174–183
[HP] HP WebInspect (2011) https://download.hpsmartupdate.com/webinspect/. Accessed July 2011
[HU79] Hopcroft JE, Ullman JD (1979) Introduction to automata theory, languages, and computation. Addison-Wesley
[HV11] Hooimeijer P, Veanes M (2011) An evaluation of automata algorithms for string analysis. In: Proceedings of the 12th international conference on verification, model checking, and abstract interpretation (VMCAI), pp 248–262
[HW09] Hooimeijer P, Weimer W (2009) A decision procedure for subset constraints over regular languages. In: Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation (PLDI), pp 188–198
[HW10] Hooimeijer P, Weimer W (2010) Solving string constraints lazily. In: Proceedings of the 25th IEEE/ACM international conference on automated software engineering (ASE), pp 377–386
[JM08] Jurafsky D, Martin JH (2008) Speech and language processing (2e). Prentice Hall
[KCGS96] Karttunen L, Chanod J-P, Grefenstette G, Schille A (1996) Regular expressions for language engineering. Nat Lang Eng 2:305–328
[KGG+09] Kiezun A, Ganesh V, Guo PJ, Hooimeijer P, Ernst MD (2009) HAMPI: a solver for string constraints. In: Proceedings of the 18th international symposium on testing and analysis (ISSTA), pp 105–116
[KGJE09] Kiezun A, Guo PJ, Jayaraman K, Ernst MD (2009) Automatic creation of SQL injection and cross-site scripting attacks. In: Proceedings of the 31st international conference on software engineering (ICSE), pp 199–209
[Kin76] King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394
[KK94] Kaplan RM, Kay M (1994) Regular models of phonological rule systems. Comput Linguist 20(3):331–378
[KM06] Kirkegaard C, Møller A (2006) Static analysis for Java servlets and JSP. In: Proceedings of the 13th international static analysis symposium (SAS), pp 336–352
[Lab09] Labs@gdssecurity.com. (2009) Adobe Flex SDK Input Validation Bug in 'index.template.html' Permits Cross-Site Scripting Attacks. http://www.securitytracker.com/alerts/2009/Aug/1022748.html
[Lot02] Lothaire M (2002) Algebraic combinatorics on words. Cambridge University Press
[Mak77] Makanin GS (1977) The problem of solvability of equations in a free semigroup. Math USSR-Sbornik 32(2):129–198
[Min05] Minamide Y (2005) Static approximation of dynamically generated Web pages. In: Proceedings of the 14th international conference on World Wide Web (WWW), pp 432–441
[MKK07] Moser A, Kruegel C, Kirda K (2007) Exploring multiple execution paths for Malware analysis. In: Proceedings of the 2007 IEEE symposium on security and privacy (S&P), pp 231–245
[MN01] Mohri M, Nederhof MJ (2001) Regular approximation of context-free grammars through transformation. Robustness Lang Speech Technol 153–163
[Mø] Møller A (2009) The dk.brics.automaton package. http://www.brics.dk/automaton/. Accessed July 2009
[Moh97] Mohri M (1997) Finite-state transducers in language and speech processing. Comput Linguist 23(2):269–311
[New00] Newsham T (2000) Format string attacks. Bugtraq mailing list. http://seclists.org/bugtraq/2000/Sep/0214.html
[NTGG+05] Nguyen-Tuong A, Guarnieri S, Greene D, Shirley J, Evans D (2005) Automatically hardening Web applications using precise tainting. In: Proceedings of the 20th IFIP international information security conference (SEC), pp 295–308
[Pug94] Pugh W (1994) The Omega project. http://www.cs.umd.edu/projects/omega/
[Raf01] Rafail J (2001) Cross-site scripting vulnerabilities. CERT Coordination Center, Carnegie Mellon University. http://www.cert.org/archive/pdf/cross_site_scripting.pdf
[RE97] Rozenberg G, Salomaa A (ed) (1997) Handbook of formal languages. Word, language, grammar, vol 1. Springer
[SAH+10] Saxena P, Akhawe D, Hanna S, Mao F, McCamant S, Song D (2010) A symbolic execution framework for JavaScript. In: Proceedings of the 31st IEEE symposium on security and privacy (S&P), pp 513–528
[SAMS10] Saxena P, Akhawe D, McCamant S, Song D (2010) Kaluza constraint solver. http://webblaze.cs.berkeley.edu/2010/kaluza/
[Shi04] Shiflett C (2004) Security corner: cross-site request forgeries. http://shiflett.org/articles/cross-site-request-forgeries
[SL] Sullo C, Lodge D (2010) Nikto. http://www.cirt.net/nikto2. Accessed July 2010
[VBdM10] Veanes M, Bjørner N, de Moura L (2010) Symbolic automata constraint solving. In: Proceedings of the 17th international conference of logic for programming, artificial intelligence, and reasoning (LPAR), pp 640–654
[XMSN05] Xie T, Marinov D, Schulte W, Notkin D (2005) Symstra: a framework for generating object-oriented unit tests using symbolic execution. In: Proceedings of the 11th international conference on tools and algorithms for the construction and analysis of systems (TACAS), pp 365–381
[YAB09] Yu F, Alkhalaf M, Bultan T (2009) Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In: Proceedings of the 24th IEEE/ACM international conference on automated software engineering (ASE), pp 605–609
[YAB10] Yu F, Alkhalaf M, Bultan T (2010) Stranger: an automata-based string analysis tool for PHP. In: Proceedings of the 16th international conference on tools and algorithms for the construction and analysis of systems (TACAS), pp 154–157
[YBCI08] Yu F, Bultan T, Cova M, Ibarra OH (2008) Symbolic string verification: an automata-based approach. In: Proceedings of the 15th SPIN workshop on model checking software (SPIN), pp 306–324
[YBI09] Yu F, Bultan T, Ibarra OH (2009) Symbolic string verification: combining string analysis and size analysis. In: Proceedings of the 15th international conference on tools and algorithms for the construction and analysis of systems (TACAS), pp 322–336. Springer

Received 15 July 2010
Accepted in revised form 10 October 2011 by Jim Woodcock