Simple Linear String Constraints

Xiang Fu, Michael C. Powell, Michael Bantegui, and Chung-Chih Li

1Department of Computer Science
Hofstra University
Hempstead, NY 11549, USA
Xiang.Fu@hofstra.edu, michael.powellcs@gmail.com, mbante2@gmail.com

2School of Information and Technology
Illinois State University
Normal, IL 61790, USA
cli2@ilstu.edu

Abstract. Modern web applications often suffer from command injection attacks such as Cross-Site Scripting and SQL Injection. Even equipped with sanitation code, many systems can still be penetrated due to the existence of software bugs (see e.g., the Samy Worm). It is desirable to automatically discover such vulnerabilities, given the bytecode of a web application. One solution would be symbolically executing the target system and constructing constraints for matching path conditions and attack patterns. The solution to such constraints is an attack signature, based on which, the attack process can be replayed. Constraint solving is the key to the success of symbolic execution. For web applications, string constraints receive most of the attention because web applications are essentially text processing programs. The study of string constraints has to balance between expressiveness and complexity.

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 greedy, reluctant, declarative, and finite replacement), which are frequently used by text processing programs. The semantics of these replacement operations are precisely modeled using finite state transducers, with various automata techniques developed 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 all feasible 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, which allows the constraint solver to support a 16-bit Unicode alphabet in practice. The algorithm is implemented in a Java constraint solver called SUSHI.

Keywords: String Analysis; Symbolic Execution; Constraint Solving; Verification; Testing; Security; Vulnerability Detection.
1. Introduction

Automatic vulnerability detection is a key problem in computer security research. In the context of web security analysis, we are interested in a variation of the problem: Assume that the binary or source code of a web application is available, can those “malicious” program inputs be automatically synthesized? In particular, discovering software bugs in user input sanitation procedures is challenging. “Corner case” bugs often lead to delicate security vulnerabilities.

A recent emerging paradigm for detecting software vulnerability is the application of the symbolic execution technique proposed by J. C. King in the 1970’s [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 problems that crash operating systems. Moser, Krügel, and Kinda adopt dynamic symbolic execution for malware analysis [MKK07]. In the area of web application security, there are several parallel efforts, e.g., JDBC Checker [GSD04] by Gould, Su, and Devanbu, finding server side vulnerabilities [YBI09, YAB10, YAB09] by Yu and Bultan, the HAMPI constraint solver [KGJE09, KGG09] by Kiežun et al, DRPLE [HW09] by Hooimeijer and Weimer, and the recent work [SAH10] by Saxena et al. in finding vulnerabilities of client-side JavaScript programs. SAFEI/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 to compute the program inputs of interests. 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 very sensitive to the matching of script tags. In this paper, we dab the problem of solving constraints related to strings as “string constraint solving”.

The major contribution of this paper is 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 difference is that a SISE constraint supports various frequently seen string operators such as substring, concatenation, and string substitution. 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], Kahuza [SAH+10], “Pure Library Language” of .Net [BT09], 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 [BT09] (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 step. This is quite different from, e.g., the solution of word equations using Makanin’s algorithm [Mak77]. Given a set of strings $R$ and a string operation $f$ (e.g., $\text{substring}$ and $\text{charAt}$), 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., [BT09, 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 sanitation. 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
protected void processRequest(
    HttpServletRequest request ...)
throws ServletException{
    PrintWriter out = response.getWriter();
    try {
        String sUserName = request.getParameter("sUsername");
        String sPwd = request.getParameter("sPwd");
        Connection conn = DriverManager.getConnection("...");
        Statement stmt = conn.createStatement();
        String strCmd = "SELECT * FROM users WHERE username='" + massage(sUserName) + "' AND pwd='" + massage(sPwd) + "'";
        ResultSet srs = stmt.executeQuery(strCmd);
        if(srs.next()){
            out.println("Welcome "+ sUserName);
            out.println("Login fail!");
        }else{
            out.println("Login fail!");
        }
    } catch (Exception exc) {...}
}

Listing 1. Vulnerable Authentication

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 find delicate SQL injection and XSS attacks. It can be extended for discovering other command injection attacks such as request forgery attack [Shi04], format string attack [New00], and PHP injection attack [Chr06].

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

2. Motivating Examples

This section presents two application examples. The first example (originally introduced in [FLP07]) shows that it is possible to discover a password bypassing attack using the string constraint solving technique. The second demonstrates the need for modeling various regular replacement semantics, because imprecise modeling can lead to false negatives during a security analysis.

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 there are many symbolic execution frameworks emerging recently. The following are a few examples: JPF-SE [APV07] for Java, Kudzu [SAH10] for JavaScript, BitFuzz [CPM10] for X86 binaries, and RubyX [CF10] for Ruby-on-Rails.
2.1. Example 1: SQL Injection for Password Bypassing

2.1.1. Vulnerable Sanitation

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 sanitation function named massage. 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 'a' and continued with eight single quotes). The second string (value of sPwd) is 14 characters long (starting with a single quote character).

![Listing 2: Malicious SQL Query](image)

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.

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 and let them be x and y. Then, by executing the statements one by one, at line number 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 (T1 to T5), 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. T1: constant word SELECT * FROM users WHERE uname='a'...
2. T2: term x[0,16]
3. T3: constant word ' AND pwd='
4. T4: term y[0,16]
5. T5: 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 T2, x[0,16] represents the sanitation performed by massage on sUname. It replaces every single quote with its escaping form (two single...
quote characters) and then performs a substring operation on the user input. Here a greedy replacement semantics is used, which is denoted using “*” in the formula. Similar is $y^0_{\pi}$, which applies the same sanitation on $sPwd$. Now by associating the symbolic string expression with pre-defined 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:

$$\begin{align*}
T_1 \circ T_2 \circ T_3 \circ T_4 \circ T_5 & \equiv \text{SELECT} \cdot \text{FROM users WHERE} \text{uname} = \text{''} (\text{''}|\S) \cdot \text{OR} \cdot \text{uname} < ' ' \cdot \text{''} \cdot \text{''} \cdot \text{''} \\
\end{align*}$$

Intuitively, the SISE equation asks the following question: after all the sanitation 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 1.4 seconds on a laptop computer with a 3.06 GHz CPU and 4GB RAM.

2.1.3. Discussion

The first example shows the power of the constraint solving techniques. Black-box vulnerability scanning tools such as Nikto [CIR] and WebInspect [SPI] 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 unlikely discover the bugs either. Only when the control/data flow information is analyzed by the vulnerability detection tool, such deeply hidden vulnerability can 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.

#### 2.2.1. Precise Modeling of Regular Replacement Matters

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

At line 3, the programmer 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>` the call on line 3 returns `<script>`.

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

$$\text{msg}_{\text{postMessage}} = \text{msg} = \text{``<script.*?><script.*?</script.*?>>```}$$

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

```sql
<\script><\script>alert('a')</script><\script>
```
Readers can verify that applying replacement to the solution string yields `<script>alert('a')</script>.

The trick of the attack is that after the reluctant replacement kills the shortest match (i.e., `<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 actually vulnerable sanitation.

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 Simple Linear String Equation (SISE), starting with the formalization of string operators and string expressions.

3.1. Preliminaries

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

3.2. String Operator Semantics

SISE supports a representative collection of string operators available in popular programming languages. This set is denoted using $O = \{\circ, [i,j], x_{r\rightarrow w}, x_{r\leftarrow w}, x_{r\rightarrow w}^+, x_{r\leftarrow w}^+\}$. They are concatenation ($\circ$), substring ([i,j]), declarative replacement ($x_{r\rightarrow w}$), reluctant regular replacement ($x_{r\leftarrow w}$), and greedy regular replacement ($x_{r\rightarrow w}^+$). For $s \in \Sigma^*$, $s[i,j]$ denotes the substring of $s$ starting from index $i$ up to index $j - 1$ (included).\(^2\) If $r \in R$ (i.e., $r$ is a regular expression), then let $r[i,j] = \{s[i,j] \mid s \in r\}$. Regular string replacement has several different semantics. Intuitively, the greedy semantics tries to match the longest string and the reluctant tries to match the shortest. Both the reluctant and greedy procedures use left-most matching, while the declarative semantics may produce multiple output words given one input word. The following example demonstrates their difference.

Example 3.1. Consider the following two cases. (i) If $s = aab$, $r = (a|ab)$, and $\omega = c$, then $s_{r \rightarrow w} = \{cc, ab\}$, and $s_{r \leftarrow w} = s_{r \rightarrow w}^+ = cc$. (ii) If $s = aaa$, $r = a^+$, and $\omega = b$, then $s_{r \rightarrow w} = \{b, bb, bbb\}$, $s_{r \leftarrow w} = bbb$, and $s_{r \rightarrow w}^+ = b$. \(\square\)

\(^2\) String index starts from 0, i.e., $s[0]$ represents the first character of $s$, and $s[|s| - 1]$ is its last element, where $|s|$ is the length of $s$. 
$s^\rightarrow_{r\omega}$ and $s^\leftarrow_{r\omega}$ are uniquely defined for any $s \in \Sigma^*$ and $r \in R$, because they are left-matching (processed one by one from 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 $\epsilon \notin r$.

$$s^\rightarrow_{r\omega} = \begin{cases} \{s\} & \text{if } s \notin \Sigma^*r\Sigma^*; \\ \{\nu r^\leftarrow_{r\omega}\mu r^\rightarrow_{r\omega} \mid s = \nu/\beta\mu, \beta \in r\} & \text{otherwise.} \end{cases}$$

If $s^\rightarrow_{r\omega} = \{s\}$, then let $s^\leftarrow_{r\omega} = s$; otherwise, define

- $s^\leftarrow_{r\omega} = \nu\omega\mu^\leftarrow_{r\omega}$ where $s = \nu\beta\mu$ 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 $y\beta m \notin r$; and if $t \neq \epsilon$ then $u \notin r$.
- $s^\rightarrow_{r\omega} = \nu\omega\mu^\rightarrow_{r\omega}$ where $s = \nu\beta\mu$ 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 $y\beta m \notin r$.

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 $s^\rightarrow_{r\omega}$, the requirement "if $y \neq \epsilon$ then $yu \notin r$ and $y\beta m \notin r" enforces that $\beta$ is the left-most matching of the regular search pattern $r$, i.e., there is no earlier matching of $r$ than $\beta$.

### 3.3. String Expression

Intuitively, a string expression is an expression over $\Sigma$ with occurrences of variables in $V$ and operators in $O$ (such as $\circ$, $[i, j]$, and $x^\rightarrow_{r\omega}$). Formally, it is defined as below.

**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 $\mu^\leftarrow_{r\omega}$, $\mu^\rightarrow_{r\omega}$, $\mu^\rightarrow_{r\omega}$ $\in E$ 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 $\mu$. 

Note that variables do not appear in Right Hand Side (RHS) 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, limiting the applicability of SISE in practice. In §9, 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 in practice.

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

---

3. 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 $s^\rightarrow_{r\omega} = bab$, $s^\leftarrow_{r\omega} = bb$. 

---
Definition 3.5. Let $\varphi$ be a SISE in the form of $\mu \equiv r$ and let $\tilde{V}$ be the set of variables in $\mu$. A solution to $\varphi$ is a function $\rho : \tilde{V} \to \Sigma^*$ 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 $\mu$.4

4. Modeling Declarative and Finite Regular Replacement Semantics

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 cannot 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.

4.1. Augmented Finite State Transducer

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

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

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, which is a subset of $\Sigma^* \times \Sigma^*$. Non-deterministic FST (NFST) is strictly more expressive than deterministic FST (DFST). FST is closed under concatenation, union, Kleene star, and composition; but 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 $L(M_3) = \{(\mu, \nu) \mid (\mu, \eta) \in L(M_1) \text{ and } (\eta, \nu) \in L(M_2) \text{ for some } \eta \in \Sigma^*\}$. □

Let $L_1$ and $L_2$ be two languages. If an FST, $M$, accepts $(\omega_1, \omega_2)$ 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 $\delta$ augmented to $Q \times R \rightarrow 2^Q$, where $R$ is the set of regular relations over $\Sigma$. □

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. It soon leads to the equivalence of AFST and FST models. 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) $Q \times R \times \Sigma^* \rightarrow 2^Q$; and (2) $Q \times \{Id(r) \mid r \in R\} \rightarrow 2^Q$ where $Id(r) = \{\omega \mid \omega \in L(r)\}$. 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 Note that some string operators such as $S_{r \rightarrow \omega}$ might produce multiple words. Thus, $\rho(\mu)$ is a set of words.
4.2. Declarative Regular Replacement

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

1. Transition 1 \( \rightarrow 2 \) 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 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 \( \epsilon \) is an element of \( \Sigma^* - \Sigma^r \Sigma^* \), in 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 \( \omega \in \Sigma^* \), \( L(M_{r \rightarrow \omega}) = \{(s, \eta) \mid s \in \Sigma^* \text{ and } \eta \in s_{r \rightarrow \omega}\} \).

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

Then we prove that for each \( \eta \in s_{r \rightarrow \omega} \): \((s, \eta) \in L(M_{r \rightarrow \omega}) \). This could also be proved by induction on the length of \( \eta \). 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 \( M_{r \rightarrow \omega} \). 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 \( M_{r \rightarrow \omega} \) respectively. From the structure of \( M_{r \rightarrow \omega} \) 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 \((s, \eta)\). The transition \( 2 \rightarrow 3 \) replaces \( \beta \) with \( \omega \). This immediately leads to \((s, \eta) \in L(M_{r \rightarrow \omega}) \).

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 fragment of the problem subsumes constant word replacement (i.e., replacement without using regular search patterns).

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 \( \epsilon \) on the input, then this is the only transition from \( q \).

**Definition 4.6.** An \( \text{FST} \ A = (\Sigma, Q, s_0, F, \delta) \) 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 $\$$ \not\in \Sigma$ be an end marker. Given a finite regular expression $r \in R$ with $\epsilon \notin r$ and $\omega_2 \in \Sigma^*$, there exist DFST $A^-$ and $A^+$ s.t. for any $\omega_1, \omega_1 \in \Sigma^*$: $\omega_1 = \omega_1^{-r}_{-\omega_2}$ iff $(\omega\$$, \omega_1\$$) \in L(A^-)$; and, $\omega_1 = \omega_1^r_{-\omega_2}$ iff $(\omega\$$, \omega\$$) \in L(A^+)$. 

**Proof:** We briefly describe how $A^+$ is constructed for $\omega_1^r_{-\omega_2}$, similar is $A^-$. 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 $A^+$ is defined as a quintuple $(\Sigma \cup \{\$$\}, Q, q_0, F, \delta)$. The set of states $Q = \{q_1, \ldots, q_{\Sigma^{\leq n}}\}$ has $|\Sigma^{\leq n}|$ elements, and let $B : \Sigma^{\leq n} \to Q$ be a bijection. Let $q_0 = B(\epsilon)$ 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 $\beta = B^{-1}(q)$: (case 1) if $a \neq \$$ and $|\beta| < n$, then $b = \epsilon$ and $q' = B(\beta a)$; (case 2) if $a \neq \$$ and $|\beta| = n$: if $\beta \notin r\Sigma^*$, then $b = \beta(0)$ and $q' = B(\beta(1)|\beta)(a)$; otherwise, let $\beta = \mu \nu$ where $\mu$ is the longest match of $r$, then $b = \omega_2$ and $q' = B(\nu a)$; (case 3) if $a = \$$, then $b = \beta(1)_r^{-r}_{-\omega_2}\$$$ and $q' = q_0$. 

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 $\omega_2$ on the output tape. The bijection $B$ is feasible because of the bounded length of $r$. Note that the $\$$ (appended to $\omega$ and $\omega_1$) is necessary in the lemma. It is used to terminate the running of 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 Figure 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 output and the buffered string becomes ab. Now at state 4, if the next input character is a, the buffered string becomes ab_a 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, and the following is an example. Assume there exists a DFST $A$ s.t. $L(A) = \{(\omega, \omega_1) \mid \omega_1 = \omega_{n+r-b-c}\}$. It is known that $(a^n, a^n) \in L(A)$ and $(a^n b, c) \in L(A)$. Since $A$ is deterministic, and $a^n$ is a prefix of $a^n b$, it leads to the contradiction that $a^n$ (as the result of $a^n_{-n+r-b-c}$) is a prefix of $c$ (as the result of $a^n_{n+r-b-c}$). Thus the assumption cannot be true. It is worthy of note that the same relation $\{(a^n b, c), (a^n, a^n)\}$ 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.

We fix some notations first. A finite state automaton (FSA) is defined as a quintuple $(\Sigma, Q, q_0, F, \delta)$. Here $\Sigma$, $Q$, $q_0$, $F$ are the alphabet, set of states, initial state, and set of final states, respectively. $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function. Deterministic and nondeterministic FSA are denoted using DFSA and NFSA, respectively.

$(q, a, q') \in \delta$ denotes a transition from state $q$ to $q'$ that is labeled with $a \in \Sigma$. Let $w = a_1a_2...a_n$, we say $w$ has a run from state $q$ to $q'$, written as $q \sim_w^* q'$, if there exists $q_0, q_1, ..., q_n \in Q$ s.t. $q = q_0$, and $q' = q_n$, and $\forall 1 \leq i \leq n : (q_{i-1}, a_i, q_i) \in \delta$. For any state $q$ of a DFSA, for any word $w \in \Sigma^*$, there is at most one $q' \in Q$ s.t. $q \sim_w^* q'$. The concept of run can be naturally extended to FST.

5.1. Modeling Reluctant Replacement Semantics

5.1.1. Overview

The FST model of reluctant replacement $S_{r-\omega}$, as shown in Figure 3, is a composition of two transducers: $A_3(r)$ (begin marker transducer) and $A_4(r, \omega)$ (reluctant replacement transducer). The subscript of each FST (e.g., 3 in $A_3(r)$) 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 word. Take the input word $dcabc$ in Figure 3 as an example, it has two matches of pattern $ab^+|ca^+$. They are $ca$ (starting from the second character) and $ab$ (starting from the third character). $A_3(r)$ marks the beginning of each match and produces one and only one output word $d\#c\#abc$. Then the reluctant replacement transducer $A_4(r, \omega)$ searches $\#$ from left to right. Whenever it discovers a $\#$, $A_4(r, \omega)$ enters the mode of substitution, which replaces the shortest match of $r$ with $\omega$ (i.e., $\epsilon$ in Figure 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 $A_3(r)$ 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 $r$). $A_3(r)$ makes sure that any run with an incorrect “look-ahead” decision will be rejected eventually. The construction of $A_3(r)$ needs two additional steps, which are contained in the road-map shown below. The entire algorithm consists of four steps:

- **Step 1**: Given $S_{r-\omega}$, let $r' = \text{reverse}(r)$. Construct an end marking DFST $A_1(r')$ which marks the end of any match of pattern $r'$.
- **Step 2**: $A_1(r')$ is restricted in that its projection to the input tape accepts $r'$ only. It is extended to $A_2(r')$ so that any input word will be accepted. In the case there is no match of $r'$, the output word is the same as input.
- **Step 3**: Reverse all transitions of $A_3(r')$. This leads to the begin marker $A_3(r)$ which marks the beginning of pattern $r$.
- **Step 4**: Replacement transducer $A_4(r, \omega)$ is constructed by integrating several techniques such as enforcing the reluctant semantics. Then $M_{r-\omega} = A_3(r)||A_4(r, \omega)$ is the FST model of $S_{r-\omega}$.
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 $A_1$, and $q_f$ is the fifth state in $A_2$.

### 5.1.2. Step 1 (End Marking DFST)

The objective of this step is to construct a DFST (let it be $A_1(r)$) that marks the end of of every match of a regular search pattern $r$.

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

**Construction Algorithm of $A_1(r)$**: Given regular search pattern $r$, we first construct a deterministic FSA (let it be $\text{DFSA}(r)$) that accepts $r$. Then we modify each final state $f$ of $\text{DFSA}(r)$ as below: (1) make $f$ a non-final state, (2) create a new final state $f'$ and establish a transition $\epsilon$ from $f$ to $f'$, (3) for any outgoing transition $(f, a, s)$ create a new transition $(f', a, s)$ and remove that outgoing transition from $f$ (keeping the $\epsilon$ transition). Thus the $\epsilon$ transition is the only outgoing transition of $f$. Then convert the FSA into a DFST as below: for an $\epsilon$ transition, its output is $\$ (end marker); for every other transition, its output is identical to the input symbol.

Take $A_1$ in Figure 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 5.1 follow directly from the construction algorithm.

**Lemma 5.2.** $A_1(r)$ is a DFST.

**Corollary 5.1.** For any state $q_i$ in $A_1(r)$, $\mu \in \Sigma^*$ and $\eta \in (\Sigma \cup \{\$\})^*$, there is at most one state $q_j$ s.t. $q_i \sim^* (\mu, \eta) q_j$ in $A_1(r)$.

Lemma 5.3 describes the major property of $A_1(r)$. Notice that condition (3) specifies that $A_1(r)$ never produces consecutive $\$ signs on its output tape. The proof of Lemma 5.3 is available in Appendix B.1.

**Lemma 5.3.** For any $r \in R$, $\mu \in \Sigma^*$ and $\eta \in (\Sigma \cup \{\$\})^*$, $(\mu, \eta) \in L(A_1(r))$ iff all the following are satisfied:

1. $\mu \in L(r)$ 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)

$A_1(r)$ has one limitation: for any $(\mu, \eta)$ accepted by $A_1(r)$, $\mu$ 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 $A_2(r)$) 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 5.4. Let $\Sigma = \{a, b, c, d\}$ and the regular search pattern $r$ be $cb^+a^+$. $A_2$ in Figure 4 is the generic end marker for $r$ (and $\Sigma$). For example, (ccbbab, ccbbabSaS) is accepted by $A_2$. Note that the input word, ccbbab, is not an instance of $cb^+a^+$, and the match actually starts from the second $c$ character. After each match of the search pattern, a $\$$ is appended. For another example, (dcba, dcbaSaS) $\in L(A_2)$.

Definition 5.5. Given $r \in R$ and $\zeta \in (\Sigma \cup \{\$$\})^*$, $\zeta$ 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) $|\zeta[x - 1]| \neq \$$ \text{ or } x = 0.$
- $(D2)$ If $\pi_\Sigma(\zeta) \in \Sigma^*r$ then $|\zeta| > 0$ and $|\zeta[|\zeta| - 1]| = \$$.

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

Lemma 5.6. For any $r \in R$ and $\mu \in \Sigma^*$ there exists one and only one $r$-end-marked output.

The proof of Lemma 5.6 is presented in Appendix B.2. It relies on the fact that no duplicate $\$$ signs exist in an end-marked output.

Construction Algorithm of $A_2(r)$: The generic end marker $A_2(r)$ works by tracing the running of a word pair $(\mu, \eta)$ on $A_1(r)$. It monitors states in $A_1$ that could be reached by any suffix of $(\mu, \eta)$. This is accomplished using a bijection function $B$ from the set of states of $A_2(r)$ to the power-set of the set of states of $A_1(r)$.

The construction process is formally presented as follows. Given $A_1(r) = (\Sigma \cup \{\$$\}, Q_1, q_0^1, F_1, \delta_1)$ as described in Step 1, $A_2(r)$ 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 (however, in SUSHI there is no need to explicitly represent those states that are unreachable from $q_0^2$). A labeling function $B : Q_2 \rightarrow 2^{Q_1}$ is defined as a bijection such that $B(q_0^2) = \{q_0^1\}$. Let $F_0$ be the set of final states of DFSA$(r)$, from which $A_1(r)$ is constructed from. The set of final states is defined as: $F_2 = \{q \mid B(q) \cap F_0 = \emptyset\}$.

The transition relation of $A_2$ 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. $B(t) \cap F_0 = \emptyset$, then for any $a \in \Sigma$: $(t, t', a : a) \in \delta_2$ iff $B(t') = \{s' \mid \exists s \in B(t) \text{ s.t. } (s, s', a : a) \in \delta_1\} \cup \{q_0^1\}$. Notice that for all $a \in \Sigma$, there exists such a $t' \in Q_2$ because $B(t')$ includes $q_0^1$ (the initial state of $A_1(r)$). This makes $A_2(r)$ accept any word on its input tape.

2. Case 2 Rule: If $t \notin F_2$, i.e. $B(t) \cap F_0 \neq \emptyset$, $t$ has exactly one transition of the form $(t, t', \epsilon : \$$) \in \delta_2$ where $t' = B^{-1}(\{q \mid q \in (Q_1 - F_0) \text{ or } q' \in B(t) : (q', q, \epsilon : \$$) \in \delta_1\})$. Note that $t'$ is a final state in $A_2$, because $B(t') \cap F_0 = \emptyset$.

Example 5.7. $A_2$ in Figure 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, $B(1) = \{1\}$, $B(2) = \{2, 1\}$, and $B(3) = \{3, 1\}$.

Running (ccbb, ccbb) on $A_2$ results in a partial run to state 3. For (ccbb, ccbb), $B(3)$ tracks all possible states in $A_1$ that could be reached by the suffixes of its input word, and they are ccbb, cbba, bb, b, and $\epsilon$. Among them, only cb and $\epsilon$ 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 (ccbb, ccbb)) and 1 (by $(\epsilon, \epsilon)$). $^7$ This is the intuition of having $B(3) = \{3, 1\}$ 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 $B(4) = \{4, 1\}$. This is because after appending (a, a) to (ccbb, ccbb), the word pair (ccba, ccba) reaches state 4 in $A_1$. The new suffix $(\epsilon, \epsilon)$ 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 $B$ to $\{4, 1\}$ in $A_1$.

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

---

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

$^6$ Some $t \in Q_2$ might have a transition leading to $B(\$$), which has a self loop $(\epsilon, \$$)$ to itself, but never reaches any final state. Such transitions are not represented in SUSHI.

$^7$ Other input words, i.e., ccbb and bb, make $A_1$ stuck and do not reach any states in $A_1$. 
Lemma 5.8. For any $r \in R$, $A_2(r)$ is a DFST.

We formalize the properties of $A_2$ using a collection of lemmas. In particular, Lemma 5.9 states that $A_2(r)$ simulates $A_1$ on running suffixes of input words. Lemma 5.10 describes $A_2$’s function as a generic end marker. The proofs are available in Appendices B.3 and B.4.

Lemma 5.9. For any $r \in R$, let $A_1(r)$ be $(\Sigma \cup \{\$\}, Q_1, q_0^1, F_1, \delta_1)$, and $A_2(r)$ 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. $q_0^2 \sim^{\ast}_{(\mu, \eta)} t$ in $A_2(r)$, both of the following are true:\footnote{Notice that the combination of the two conditions is not equivalent to the statement that $B(t)$ is always equal to $\{q_1^t | \exists \nu \prec \mu, \zeta \in (\Sigma \cup \{\$\})^* \text{ s.t. } q_0^t \sim^{\ast}_{(\nu, \zeta)} q_1^t \}$.}

1. For any $q_1^t \in B(t)$, there exists $\nu \prec \mu$ and $\zeta \in (\Sigma \cup \{\$\})^*$ s.t. $q_0^t \sim^{\ast}_{(\nu, \zeta)} q_1^t$ in $A_1(r)$.
2. For any $\nu \prec \mu$ s.t. $\nu \in \textit{PREFIX}(r)$, there exists $\zeta \in (\Sigma \cup \{\$\})^*$ and $q_1^t \in B(t)$ s.t. $q_0^t \sim^{\ast}_{(\nu, \zeta)} q_1^t$ in $A_1(r)$.

Lemma 5.10. For any $r \in R$, any $\mu \in \Sigma^*$, and any $\eta \in (\Sigma \cup \{\$\})^*$, $(\mu, \eta) \in L(A_2(r))$ iff $\eta$ is the r-end-marked output of $\mu$.

Corollary 5.2. For any $r \in R$, and any $\mu \in \Sigma^*$, there exists one and only one $\eta \in (\Sigma \cup \{\$\})^*$ s.t. $(\mu, \eta) \in L(A_2(r))$.

5.1.4. Step 3 (Begin Marker of Regular Pattern)

From $A_2(r)$ it is straightforward to construct a “reversed” transducer which marks the beginning of any match of the search pattern $\textit{reverse}(r)$ in an input word.

Example 5.11. $A_3$ shown in Figure 5 is a reverse of $A_2$ in Figure 4. $A_3$ marks the beginning for pattern $r = a^+b^+c$ in any input word on alphabet $\{a, b, c, d\}$. For example, $(aabbcc, \#a\#abc) \in L(A_3)$.

Construction Algorithm of $A_3(r)$: Let $A_2(\textit{reverse}(r))$ be a quintuple $(\Sigma \cup \{\$\}, Q_2, q_0^2, F_2, \delta_2)$. $A_3(r)$ is defined as $(\Sigma \cup \{\$\}, Q_3, q_0^3, F_3, \delta_3)$, where $Q_3 = Q_2 \cup \{q_0^3\}$ and $F_3 = \{q_0^3\}$. $\delta_3$ is the minimal set of transitions derived from $\delta_2$, satisfying the following three rules:

1. For any $(t, t', a : a) \in \delta_2$: $(t', t, a : a)$ is contained in $\delta_3$.
2. For any $(t, t', \varepsilon : \#) \in \delta_2$: $(t', t, \varepsilon : \#)$ is contained in $\delta_3$.
3. For any $t \in F_2$ : there exists $(q_0^3, t, \varepsilon : \varepsilon)$ in $\delta_3$.

Definition 5.12. Given $r \in R$ and $\zeta \in (\Sigma \cup \{\$\})^*$, $\zeta$ 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 $(1)$ $\pi_{\Sigma}(\zeta[x + 1, |\zeta|]) \in r\Sigma^*$; $^9$ and $(2)$ $\zeta[x + 1] \neq \#$ or $x = |\zeta| - 1$.

\footnote{Note that $\zeta[x, |\zeta|]$ is the suffix of $\zeta$ starting from index $x$, with right bound $|\zeta|$ (not included). For $\zeta[|\zeta|, |\zeta|]$, it generates $\varepsilon$.}
• (E2) If \( \pi_\Sigma(\zeta) \in r\Sigma^* \) then \( \zeta[0] = \# \).\(^{10}\)

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

\[ \tag*{\Box} \]

**Lemma 5.13.** For any \( r \in R \) there exists an FST \( A_3(r) \) s.t. for any \( \mu \in \Sigma^* \) and \( \eta \in (\Sigma \cup \{\#\})^* \): \((\mu, \eta) \in L(A_3(r)) \) iff \( \eta \) is the \( r\)-begin-marked output of \( \mu \).

Notice that \( A_3(r) \) is nondeterministic. However, due to the construction and the deterministic \( A_2(r) \), \( A_3(r) \) can always make the “smart” decision to enforce there is one and only one run which “correctly” inserts the label \( \# \). Any incorrect insertion will never reach a final state. The nondeterminism gives \( A_3(r) \) the “look ahead” ability.

**5.1.5. Step 4 (Reluctant Replacement)**

The purpose of the last transducer is to perform the “reluctant replacement”. There are three tasks to accomplish: (1) filtering extra \( \# \) symbols, (2) enforcing reluctant semantics, and (3) procedural replacement. We address them one by one.

**Task 1: Filtering Extra \# Symbols.**

**Example 5.14.** To see the needs of the filtering operation, consider the following example. Let \( r = ca|aa \), \( \mu = caa \), and \( \omega = d \). The \( r\)-begin-marked output of \( \mu \) is \( \#c\#aa \). Reluctant replacement \( \mu \rightarrow \omega \) 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 5.15.** Let \( L \) be a regular language on \( \Sigma \), and \( x \notin \Sigma \) be a special symbol. \( L_x \), the interspersed language of \( L \) with \( x \), is defined as \( L_x = \{ \omega \mid \omega \in (\Sigma \cup \{x\})^* \text{ and } \pi_\Sigma(\omega) \in L \} \).

\( L_x \) in Definition 5.15 is a regular language. To filter extra \( \# \) is accomplished using a transition of \((L_x : L)\) in AFST.

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

\[ \tag*{\Box} \]

**Lemma 5.17.** For any \( r \in R \), \( reluc(r) \) is a regular language.

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

Notice that \( reluc(r) \) does not find the “shortest” instance of \( r \), but the “reluctant” matches. To see the difference, consider \( r = a^+|cda|^+ \). The shortest instance of \( r \) is \( a \). But in the process of reluctant replacement, given input word \( ecda|bc \), \( cda \) is the “reluctant match”, because starting at position 1, it is the “shortest” match.

**Task 3: Procedural Replacement and Construction Algorithm of \( A_4(r, w) \).** Given \( r \in R \) and \( \omega \in \Sigma^* \),
we now construct a transducer \(\mathcal{A}_1(r, \omega)\), which given a \(r\)-begin-marked word on its input tape, generates the result of reluctant replacement on its output tape.

The design of \(\mathcal{A}_1(r, \omega)\) is shown on the right half of Figure 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, \(\mathcal{A}_1(r, \omega)\) enters the replacement mode, which replaces the reluctant match of \(r\) with \(\omega\) (and also removes extra \# in the match). Piping with \(\mathcal{A}_3(r)\) leads to the precise modeling of reluctant replacement.

**Definition 5.18.** Given any \(r \in R\) and \(\omega \in \Sigma^*\), \(AFST \mathcal{A}_1(r, \omega)\) is defined as a quintuple \((\Sigma \cup \{\$\}, Q_1, f_1^1, F_1, \delta_1)\) where \(Q_1 = \{f_1^1, s_1^1, s_2^1\}\), \(F_1 = \{f_1^1\}\), and \(\delta_1\) consists of the following four transitions: \(\tau_1 : (f_1^1, f_1^1, \text{Id}(\Sigma))\), and \(\tau_2 : (s_1^1, s_2^1, \text{reluc}(r\#) : \omega)\), and \(\tau_3 : (s_1^1, s_2^1, \epsilon : \epsilon)\).

We now prove that \(\mathcal{A}_2(r, \omega)\) 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 5.19.** Given \(r \in R\) with \(\epsilon \notin r\), and \(\omega \in \Sigma^*\), for any \(r\)-begin-marked word \(\kappa \in (\Sigma \cup \{\#\})^*\) and \(\zeta \in \Sigma^*\): \((\kappa, \zeta) \in L(\mathcal{A}_2(r, \omega))\) iff \(\kappa = \zeta \in \Sigma^* - \Sigma^* r \Sigma^*\) or both of the following are true:

- \((F1)\) \(\kappa\) 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 \(ym \notin r\#\); and if \(t \neq \epsilon\) then \(u \notin r\#\).
- \((F2)\) \(\zeta\) can be written as \(\nu\omega\eta\), and \(\nu\) \(\omega\eta\) is accepted by \(\mathcal{A}_1(r, \omega)\).

**Proof:** The base case where \(\kappa = \zeta\) and \(\kappa \in \Sigma^* - \Sigma^* r \Sigma^*\) holds because the run of \((\kappa, \zeta)\) on \(\mathcal{A}_2(r, \omega)\) is a self loop of the transition \((f_1^1, f_1^1, \text{Id}(\Sigma))\). We now concentrate on the case where state \(s_1^1\) is visited at least once. Notice that the fact \(\kappa\) is \(r\)-begin-marked is essential in the proof.

**Direction \(\Rightarrow:\)** We first prove that if \((\kappa, \zeta) \in L(\mathcal{A}_2(r, \omega))\) then \(\kappa\) and \(\zeta\) satisfy \(F1\) and \(F2\). Let a run \(\gamma\) of \((\kappa, \zeta)\) on \(\mathcal{A}_1(r, \omega)\) be written as follows:

\[
\mathcal{f}_1^1 \sim_{(\nu, \epsilon)} \mathcal{f}_1^1 \sim_{(\# , \epsilon)} \mathcal{s}_1^1 \sim_{(\beta, \omega)} \mathcal{s}_2^1 \sim_{(\epsilon, \epsilon)} \mathcal{f}_1^1 \sim_{(\mu, \eta)} \mathcal{f}_1^1
\]

In the above “presentation” of \(\gamma\), we require that the “\#” before \(s_1^1\) and the \(s_2^1\) before \((\epsilon, \epsilon)\) are both their first occurrence in \(\gamma\). Then we show that the \(\beta, \mu, \eta\) resulted from this writing of \(\gamma\) satisfy \(F1\) and \(F2\).

From the structure of \(\mathcal{A}_2(r, \omega)\), it follows that \(\nu \in \Sigma^*\) and \(\beta \in \text{reluc}(r\#)\). We need to further prove that \(\nu \in \Sigma^* - \Sigma^* r \Sigma^*\), this is available by inferring from the fact that \(\kappa\) is \(r\)-begin-marked (otherwise, there would be a \# inside \(\nu\) ). For \(F2\), \((\mu, \eta) \in L(\mathcal{A}_2(r, \omega))\) holds because the last segment of \(\gamma\), i.e., \(f_1^1 \sim_{(\mu, \eta)} f_1^1\), 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 \(ym \notin r\#\); and, \((G2)\) if \(t \neq \epsilon\) then \(u \notin r\#\). From \(\gamma\), we can infer that \(\beta \in \text{reluc}(r\#)\), which leads to \(G2\). \(G1\) is proved using the fact that \(\kappa\) is \(r\)-begin-marked, and there does not exist an earlier match.

**Direction \(\Leftarrow:\)** It is straightforward to prove that if \(\kappa\) and \(\zeta\) satisfy \(F1\) and \(F2\), then \((\kappa, \zeta) \in L(\mathcal{A}_2(r, \omega))\). The run of \((\kappa, \zeta)\) on \(\mathcal{A}_1(r)\) is constructed using induction.

**Lemmas 5.19 and A.5 leads to Theorem 5.20.** The complete proof can be found in Appendix B.5.

**Theorem 5.20.** Given any \(r \in R\) and \(\omega \in \Sigma^*\), and let \(\mathcal{M}_{\omega^{-}}\) be \(\mathcal{A}_3(r)||\mathcal{A}_4(r, \omega)\), then for any \(\kappa, \zeta \in \Sigma^*\): \((\kappa, \zeta) \in L(\mathcal{M}_{\omega^{-}})\) iff \(\zeta = \kappa_{\omega^{-}}\).

### 5.2. Modeling Greedy Replacement Semantics

#### 5.2.1. Overview

Modeling greedy replacement \(S_{\omega^{-}}\) 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.

\(\text{Note that by the last condition, } \beta \in \text{reluc}(r\#).\)
1. Begin marker transducer $A_3(r)$, which inserts a # before each match of $r$.
2. Nondeterministic end marker transducer $A_5(r)$, which nondeterministically insert a $ after each match of $r$.
3. Pairing filter $A_6$, which makes sure that each # is paired with a corresponding $.
4. Match filter $A_7(r)$, which checks if the substring between each pair of # and $ signs is a match of $r$.
5. Begin marker protector $A_8(r)$, which avoids early removing of begin markers by $A_6$.
6. Longest match filter $A_9(r)$, which ensures the match of $r$ is the longest one starting from a given position in the input word.
7. Replacement transducer $A_{10}(r, \omega)$ 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.

Example 5.21. Let $r = a^+$ and $\omega = x$. Figure 6 displays the process of applying the seven transducers on input word aabab. In this example, each transducer is regarded as an input/output device.

- Step 1: begin marker transducer $A_3(r)$ inserts a # before each match of $r$, and it yields output word #a#ab#ab.
- Step 2: Nondeterministic end marker $A_5(r)$ 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 $A_6$ tries to filter extra # signs, and pair # and $ signs. If the effort fails, the input word will be rejected. For example, given input word #a#ab#a$b, $A_6$ yields output word #aab$a$b – all the # signs between the first # and $ are filtered. For another example, input word #a#ab$a$b is rejected because there is no way to find a matching $ for the last #. In summary, all the output words of $A_6$ 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 $A_7(r)$ rejects those input words where any substring between a pair of begin and markers is not a match of $r$. For example, #a$ab$a$b is rejected at this step.
- Step 5: Begin marker protector $A_8(r)$ has no application here. It is used when $r$ contain * operators. We will show one example later.
- Step 6: Longest match filter $A_9(r)$ rejects those input words that do not conform to the greedy semantics. For example, #a$#a$b#a$b is rejected here because there could be a longer match aa starting from the first #.
• Step 7: Replacement transducer $A_{10}(r, \omega)$ replaces each substring inside a pair of # and $ with $\omega$. For example, given input word $\#aab\#b\#a\#b$, it generates $xbb$. Note that $xbb$ is the one and only one output word of $aabab$, generated by the composition of the seven transducers.

\[\square\]

5.2.2. Step 1: Begin Marker $A_3(r)$.

The begin marker is defined in §5.1.4.

5.2.3. Step 2: Nondeterministic End Marker $A_5(r)$.

The construction algorithm of nondeterministic end marker $A_5(r)$ is very similar to that of $A_2(r)$ (the generic end marker in §5.1.3). There are two differences: (1) the $ after a match of $ is optional; and (2) the input word now contains # (introduced by $A_3(r)$ in Step 1). (1) is handled by adding an ($, \epsilon$) transition in the Case 2 rule (adapted from $A_2(r)$). (2) is solved by adding (#, #) transitions to each state.

Construction Algorithm of $A_5(r)$: Given $A_1(r) = (\Sigma \cup \{\}$, $Q_1$, $q^1_0$, $F_1$, $\delta_1)$ as described in §5.1.2, $A_5(r)$ is a quintuple $(\Sigma \cup \{\}$, $Q_5$, $q_0^5$, $F_5$, $\delta_5)$. $Q_5$ is a set of $|2^{Q_1}|$ states, and similarly define a labeling function $B : Q_5 \to 2^{Q_1}$ s.t. $B(q_0^5) = \{q_0^1\}$. $F_5 = \{\} \cup \{a \in \Sigma \mid L(A_2(r)) \subseteq \{a\}\}$.

1. **Case 1 Rule:** If $t \notin F_5$, i.e., $B(t) \cap F_0 = \emptyset$, then for any $a \in \Sigma$: $(t, t', a : a) \in \delta_2$ iff $B(t') = \{s' \mid \exists s \in B(t) \text{ s.t. } (s, s', a : a) \in \delta_1\} \cup \{q_0^1\}$.
2. **Case 2 Rule:** If $t \notin F_5$, i.e., $B(t) \cap F_0 \neq \emptyset$, $t$ has exactly two transitions: $(t, t', \epsilon : \epsilon)$ and $(t, t', \epsilon : \epsilon)$ where $t' = B^{-1}(\{q \mid q \in (Q_1 - F_0) \text{ or } \exists q' \in B(t) : (q', q, \epsilon : \epsilon) \in \delta_1\})$.
3. **Rule 3:** $(t, t, # : #:) \in \delta_5$.

**Lemma 5.22.** For any $r \in R$, any $r$-begin-marked word $\kappa \in (\Sigma \cup \{\})^*$, and $\zeta \in \Sigma^*$, $(\kappa, \zeta) \in L(A_5(r))$ iff both of the following conditions are satisfied:

- (11) 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 $x = 0$.
- (12) $\kappa = \pi_{\Sigma \cup \{\}}(\zeta)$.

Notice that $\Sigma_2 = \Sigma \cup \{\}, 11$ in Lemma 5.22 is very similar to D1 in Definition 5.5 (for $r$-end-marked output), except that here only if (instead of iff) is used. The proof of Lemma 5.22 can be adapted from the proof of of Lemma 5.10.

\[\text{Note that Case 1 and Case 2 rules cannot be adopted simultaneously. Rule 3 applies to every state.}\]
5.2.4. Step 3: Pair Filter $A_6$.

Then we need a filter to remove extra markers so that every $ is paired with a #. This is accomplished via a transducer $A_6$ (note that $A_6$ is not parameterized by $r$).

Construction Algorithm of $A_6$: As shown in Figure 7, $A_6$ always swaps between two stages: (1) waiting for #, and (2) waiting for $. While waiting for #, it removes any $ encountered and generates an identical character for each input symbol in $\Sigma$. While waiting for $\$, it removes any # encountered. $A_6$ ensures that there are equal number 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 $L$. This is accomplished later using additional filters.

Formally, $A_6$ is defined using a quintuple $(\Sigma \cup \{#, \}, Q_6, q_0^6, F_6, \delta_6)$ where $Q_6 = \{q_0^6, q_1^6\}$, $F = \{q_0^6\}$, and $\delta_6$ consists of the following transitions: $(q_0^6, q_0^6, Id(\Sigma))$, and $(q_0^6, q_0^6, \$ : \varepsilon)$, and $(q_0^6, q_1^6, #: #:)$, and $(q_1^6, q_1^6, #: #:)$, and $(q_1^6, q_1^6, Id(\Sigma))$, and $(q_1^6, q_0^6, \$: $)$.

Example 5.23. We list four pairs of input and output words that are accepted by $A_6$ in the following: $(#ab#cSc#ab$, and $(#abSc#a#b$, and $(#abc#a#b$, and $(#abcc#a#b$, and $(#abcc#a$c#ab$, and $(#abc#c#ab$. The following are two examples of input words, pairing with any output words, will not be accepted by $A_6$: $#abcSc#a$, and $abc#c#ab$.

Lemma 5.24 states that for each # in the output word of $A_6$, there is a pairing $\$, and there are no other markers in between the pair.

Lemma 5.24. $A_6$ is a DFST. For any $\mu, \eta \in \Sigma_2^+$, if $(\mu, \eta) \in L(A_6)$ then the following are true for $\eta$:

1. $\pi_\Sigma(\mu) = \pi_\Sigma(\eta)$.
2. $\forall i, 0 \leq i < |\eta| : \eta[i] = # \Rightarrow \exists j, i < j < |\eta| : \eta[j] = $ \& $ \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] = #: #: \& \forall k, j < k < i : \eta[k] \in \Sigma$.

5.2.5. Step 4: Match Filter $A_7(r)$

Match filter makes sure that the substring between each pair of # and $ is a match of $r$. It is formally defined in Definition 5.25. For regular languages, let $\cap$ and $\Pi$ 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 5.25. Let $L_1 = \Sigma_2^+(r \cap \Sigma^*)\Sigma_2^+$. The match filter $A_7(r)$ is an FST that accepts $Id(L_1)$.

Note that in the formula of $L_1$, $r$ is defined as $\Sigma_2^+ - r$. Thus, $r \cap \Sigma^*$ is essentially $\Sigma^* - r$. Let DFSA($L_1$) be the DFSA that accepts $L_1$. $A_7(r)$ 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 5.26. Let $r = ab^+bc$ and $\alpha = ab#b#b$. $\alpha$ is accepted by $L_1$ because the substrings contained in the two #*'s pairs are instances of $r$. Notice the use of $\Pi \cap \Sigma^*$ in the formula of $L_1$, it avoids treating the substring $b#b$ of $\alpha$ (i.e., $\alpha[3, 9]$) as a substring between a pair of any (but not the closest neighboring) begin/end markers. For another example, $aa#aba#b#b \notin L_1$ 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 $A_8(r)$

Occasionally $A_6$ might treat “good” begin markers as the extra to filter. The begin marker protector $A_8(r)$ is defined to avoid such cases. It is defined in Definition 5.27. We explain the design decision in Examples 5.28 and 5.29.

---

13 The claim assumes that $A_7(r)$ is applied after $A_6$, where all markers have been paired.
Definition 5.27. Let \( L_2 = \Sigma_2^{\ast} \cap \left( \Sigma \Sigma_2^{\ast} \cap r_{\#} . s \Sigma_2^{\ast} \right) \cap \Sigma_2^{\ast} \cap \left( \Sigma \cap r_{\#} . s \right) \). The begin marker protector \( A_8(r) \) 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 \( \epsilon \).

Example 5.28. Let \( \alpha = bab \). The output of \( a_\wedge a_\wedge \ldots \) should be \( cbcbc \), and the correct intermediate marking (i.e., the input word to the replacement transducer \( A_{10}(r, \omega) \)) should be \( \#b\#a\#b\#\#. \) Notice that there are four matches of \( \alpha \), and they are the \( \epsilon \) before the first \( b \) in \( \alpha \), the first \( a \), the \( \epsilon \) before the second \( b \), and the \( \epsilon \) after the second \( b \). We show that, without the proper protection provided by \( L_2(1) = \Sigma_2^{\ast} \cap \left( \Sigma \Sigma_2^{\ast} \cap r_{\#} . s \Sigma_2^{\ast} \right) \), an additional word \( cbcbc \) (missing a \( c \) before the second \( b \) in the output word) might be generated.

Consider \( \beta = \#b\#a\#b\#\$. It is an incorrect marking because the \( \epsilon \) 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. \( (bab, \#b\#a\#b\#) \in L(A_3(r)) \) because \( A_3(r) \) has successfully identified the beginning position of all the four matches of \( \alpha \).
2. \( (\#b\#a\#b\#, \#b\#a\#b\#\$) \in L(A_5(r)). \#b\#a\#b\#\$ is one of many available output words for the given input. This is because \( A_5(r) \) nondeterministically insert \$ markers after the match of \( r \). In this case, the \$ for the second \# is not inserted.
3. \( (\#b\#a\#b\#\#, \#b\#a\#b\#\$) \in L(A_6). \) Now the interesting part is: the missing \$ for the second \#, actually causes the third \# being deleted by the pair filter \( A_6 \)!

Now we explain the design idea of \( L_2(1) \). It is used to refine 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 \( \beta \) (i.e., \( \#b\#a\#b\#\$ \)) and \( L_2(1) \) (i.e., \( \Sigma_2^{\ast} \cap \left( \Sigma \Sigma_2^{\ast} \cap r_{\#} . s \Sigma_2^{\ast} \right) \)). Here \( r \) is \( a^* \) and its match is the \( \epsilon \) before the second \( b \) in \( \alpha \). This \( \epsilon \) is not preceded by the needed \#. Note that in \( L_2(1) \), the preceding and succeeding letters of the \$, i.e., [\( \#^\ast \# \) and \( \Sigma \)] are used to eliminate all possibilities of \# preceding \( r \). They match the highlighted \( a \) and \( b \) in \( \#b\#a\#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 5.38.

Example 5.29. The greedy replacement \( a_\wedge a_{\wedge \ldots} \) 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 \( A_3(r)||A_5(r)||A_6 \), which missed the second match of \( a^* \).

\( L_2(2) = \Sigma_2^{\ast} \cap \left( \Sigma \cap r_{\#} . s \right) \) refutes such a case by stating that: if \( \epsilon \in r \) (this is implied by the \( (\$ \cap r_{\#} . s) \) in the formula), then the intermediate marking (i.e., the input to \( A_{10}(r, \omega) \)) 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 5.38 (Case 2).

5.2.7. Step 6: Longest Match Filter \( A_9(r) \)

The longest match filter \( A_9(r) \) ensures that each substring between a marker pair is the longest match of \( r \), starting from the begin marker. \( A_9(r) \) is effective based on the assumption that \( A_7(r) \) and \( A_8(r) \) have already been applied.

Definition 5.30. Let \( L_3 = \Sigma_2^{\ast} \cap \left( (\Sigma^\ast \Sigma^\ast \cap r_{\#} . s) \Sigma_2^{\ast} \right) \). The longest match filter \( A_9(r) \) is an FST that accepts Id(\( L_3 \)).

\(^{14}\) Readers can verify the correctness of this argument using prevalent regex language packages, such as java.util.regex.
In L₃, the $r_{#,s}$ models a longer match of $r$ which contains a shorter match. This is described by $(\Sigma^*(\Sigma^+_{#,s}))$. Here the $\Sigma^*$ before the $\#$ in $\Sigma^*(\Sigma^+_{#,s})$ is the shorter match. The $(\Sigma^+_{#,s})$ denotes the difference between the longer match and the shorter match, and it has at least one character in $\Sigma$. This is because $(\Sigma^+_{#,s})$ refers to a word in $\Sigma^+$ interspersed with begin/end markers, i.e., for any $x \in (\Sigma^+_{#,s})$, $|\pi(x)| > 0$.

In summary, L₃ rejects an improper marking $\#$ (highlighted in $\Sigma^*(\Sigma^+_{#})\Sigma^*$). The $\#$ improperly marks a shorter match of $r$ while another longer match exists and starts at the same #.

Example 5.31. Let $r = a^+$ and $\beta = c#a\#a\$. $\beta$ will be rejected by filter L₃. Here $c$ in $a\#a\#a\$ matches the $\Sigma^*$ in $\Sigma^*(\Sigma^+_{#})\Sigma^*$. $a\#a\#$ is the longer match, and it corresponds to both the $r_{#,s}$ and $(\Sigma^*(\Sigma^+_{#}))$ in the formula. Here the improper $\#$ (marking a shorter match) is highlighted in both $a\#a\#$ and $(\Sigma^*(\Sigma^+_{#}))$. The difference between the longer and shorter match, i.e., $#a\#a\$, corresponds to $(\Sigma^+_{#})$ in the pattern.

5.2.8. Step 7: Replacement Transducer $A_{10}(r, \omega)$

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

Definition 5.32. The replacement transducer $A_{10}(r, \omega)$ is a quintuple $(\Sigma_2, Q_{10}, q_{10}, F_{10}, \delta_{10})$ where $Q_{10} = \{q_{10}^0, q_{10}^1\}$ and $F_{10} = \{q_{10}^0\}$. $\delta_{10}$ contains four transitions: $(q_{10}^0, q_{10}^1, \Sigma^+$, $(q_{10}^1, q_{10}^0, \# : c)$, and $(q_{10}^1, q_{10}^0, \Sigma : \epsilon)$. Here $\delta_{10}$ is because (\Sigma^*_{#,s}) where $\Sigma^*$ denotes a longer match of $r$ with another longer match existing and starts at the same #.

5.2.9. Correctness Proof

This subsection presents the formal proof. We show that the composition of $A_3(r)||A_5(r)||A_6||A_7(r)||A_8(r)||A_9(r)$ can correctly generate the proper greedy marking for any input word. Then feeding the marked word to $A_{10}(r, \omega)$ yields the desired result.

Definition 5.33. The greedy marking transducer $M^+_r$ is defined as $A_3(r)||A_5(r)||A_6||A_7(r)||A_8(r)||A_9(r)$. The procedural greedy replacement transducer $M^+_{r-\omega}$ is $M^+_{r-\omega}$.

Definition 5.34. Let $\kappa \in R$ with $\epsilon \notin r$, for any $\kappa \in \Sigma^*$ and $\zeta \in \Sigma^*$, $\zeta$ is said to be an $r$-greedy-marking of $\kappa$ iff one of the following two conditions is satisfied:

1. $\kappa \in \Sigma^* - \Sigma^*r\Sigma^* \text{ and } \zeta = \kappa$; 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 $y\beta \notin r$. Let $\eta$ be an $r$-greedy-marking of $\mu$. Then $\zeta = \nu\#\beta\eta$.

Definition 5.34 is adapted from the formalization of $S^+_{r-\omega}$ 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 5.35. The proof can be obtained by applying induction on the length of the input word. It is also not hard to infer Lemma 5.36 from the transition relation construction of $A_{10}(r, \omega)$ and Definition 5.34.

Lemma 5.35. For any $r \in R$ and any $\kappa \in \Sigma^*$, there exists one and only one $r$-greedy-marking of $\kappa$ (letting it be $\zeta$), and $\pi_\Sigma(\zeta) = \kappa$.

Lemma 5.36. For any $r \in R$ and any $\kappa, \zeta, \omega \in \Sigma^*$ and let $\eta$ be the $r$-greedy-marking of $\kappa$, $\zeta = \kappa^+_{r-\omega}$ iff $(\eta, \zeta) \in L(A_{10}(r, \omega))$.

Lemma 5.37. For any $r \in R$, $\kappa \in \Sigma^*$ and $\zeta \in \Sigma^*$, if $(\kappa, \zeta) \in L(M^+_r)$ then $\pi_\Sigma(\zeta) = \kappa$. 
Lemma 5.38. For any \( r \in R \), any \( \kappa \in \Sigma^* \) and \( \eta \in \Sigma_3^r \), \( \eta \) is the \( r \)-greedy-marking of \( \kappa \) iff \((\kappa, \eta) \in L(\mathcal{M}_r^+) \).

Proof: We show one portion of the proof and how \( L_2(2) \) in \( \mathcal{A}_5(r) \) 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 \( \eta \) is the \( r \)-greedy-marking of \( \kappa \). It is assumed that the direction \( \Rightarrow \) has been proved (see Appendix B.6). As there exists one and only one \( r \)-greedy marking for any word, and it is the output word of \( \mathcal{M}_r^+ \) (by direction \( \Rightarrow \)). We only need to show the following proposition is true for accomplishing the proof goal:

(J1) If both \((\kappa, \eta) \) and \((\kappa, \eta') \) 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 \( \eta \) and \( \eta' \) as following, using subscript to indicate the output of a particular transducer.

\[
\begin{align*}
\kappa \text{ } \mathcal{A}_3(r) \xrightarrow{\eta_1} \mathcal{A}_5(r) \xrightarrow{\eta_2} \mathcal{A}_7(r) \xrightarrow{\eta_3} \mathcal{A}_8(r) \xrightarrow{\eta_4} \mathcal{A}_9(r) \xrightarrow{\eta_5} \\
\kappa \text{ } \mathcal{A}_3(r) \xrightarrow{\eta'_1} \mathcal{A}_5(r) \xrightarrow{\eta'_2} \mathcal{A}_7(r) \xrightarrow{\eta'_3} \mathcal{A}_8(r) \xrightarrow{\eta'_4} \mathcal{A}_9(r) \xrightarrow{\eta'_5}
\end{align*}
\]

Based on the construction algorithm of all component transducers, one can infer that \( \eta_3 = \eta'_3 \), and \( \eta_6 = \eta_7 = \eta_8 = \eta \), and \( \eta'_6 = \eta'_7 = \eta'_8 = \eta' \).

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

- **Case 1** (\( e, a \)): \( |\eta| = i \) and \( \eta'[i] \in \Sigma \); or symmetrically \( |\eta'| = i \) and \( \eta[i] \in \Sigma \).
- **Case 2** (\( e, \# \)): \( |\eta| = i \) and \( \eta'[i] = \# \); or symmetrically \( |\eta'| = i \) and \( \eta[i] = \# \).
- **Case 3** (\( e, \$ \)): \( |\eta| = i \) 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 neither of the above cases can hold via contradiction. Take Case 2 as an example.

**Case 2**: The goal is to prove by contradiction that the following cannot be true: \((K)\): \(|\eta| < i \) and \( \eta'[i] = \#, \) and \( \eta[0, i] = \eta'[0, i] \). The idea of the proof is presented in Figure 8. Assume that proposition \((K)\) is true, one can reach the following proposition:

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

\( e \in r \) follows from the following facts: (1) \( \pi_\Sigma(\eta) = \pi_\Sigma(\eta') \) and \( \pi_\Sigma(\eta) = \pi_\Sigma(\eta'[0, i]) \), which implies that \( \pi_\Sigma(\eta'[i], \eta'[i]) = c \); and (2) by Lemma 5.13, every \( \# \) precedes a match of \( r \). Then we have \( \eta'[i + 1] = \$ \).

In the following, we trace ("taint") the data processing that generates the last two symbols in \( \eta' \), and let them be \( \# i \) and \( \$ i + 1 \), respectively. Observing all the transducers in \( \mathcal{M}_r^+ \), \( \mathcal{A}_3(r) \) introduces \( \# \), \( \mathcal{A}_5(r) \)

\[\text{Fig. 8. Proof Idea of Lemma 5.38}\]

15 Here ‘(‘ and ‘)’ are the grouping control characters in regular expression. They should not be treated as elements of \( \Sigma \).
introduces $, and $A_i$ could remove # as well as $. We know that $η_3=η'_3$, and $η_6 = η$, and $η'_6 = η'$. From the above facts, one can infer that $\#_i$ appears in both $η_3$ and $η'_3$. However, after $η_3$ is fed to $A_i$, $\#_i$ is removed from the output word (i.e., $η_6$) using transition $(q^*_1, q^*_0, \# : ε)$. Notice that $q^*_0$ is not a final state. To come back to final state $q^*_0$, a transition $(q^*_0, q^*_0, $ : $)$ has to be taken. The only $|$ now left on the input word (i.e., $η_6$) is the $i$, and it is guaranteed to be the last input symbol on $η_6$ (because $A_5$ does not insert consecutive $|$ symbols). Thus $i_{i+1}$ is also the last symbol of $η_6 = η$. Notice that the last element of $η$ is located at index $i = i$. Therefore, the following is true:

(K2): $η[i-1] = η'[i-1] =$.

Combining (K1) and (K2), one can infer that $η'$ can be written as the following form, where $α ∈ Σ^*$_2:

$η' = α Σ_{i-1} # i Σ_{i+1}$

Now since the $|$ before $\#_i$ (i.e., $i_{i-1}$) has a paired $|$ in $η'$, we can infer that there is at least one element before it. However, $η'[i-2]$ cannot be $\#$, because $η'[i-2] = $ leads to the conclusion that $η'[i-2] = η[i]$, when traced back to $η_6$, are consecutive $|$ signs generated by $A_6(r)$. This contradicts with Lemma 5.13 and Definition 5.12. Similarly, $η'[i-2]$ cannot be $\#$. Thus the only choice for $η'[i-2]$ is a symbol in $Σ$ (let it be $a$). Therefore one can write $η'$ as the following, where $β ∈ Σ^*$_2:

$η' = β α S # S$

Notice that $η'$ is contained in $Σ^*$_2 | # | ($ ∩ r_#_S$). This conflicts with the filter $L_2(2)$ placed in $A_6(r)$. In conclusion, Case 2 cannot be true.

Definitions 5.36 and 5.38 immediately lead to Theorem 5.39, given that $M^+ r-ω M^+ || A_10(r, ω)$.

Theorem 5.39. Given any $r ∈ R$ and $κ, ζ, ω ∈ Σ^*$, $ζ = κ^+ r-ω$ iff $(κ, ζ) ∈ L(M^+ r-ω)$.

6. Solving Simple Linear String Equation

This section presents the solution algorithm. A SISE equation is first decomposed into a number of atomic steps. Each step is a string operation such as concatenation, substring, 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. The computation is similar to the standard symbolic image computation in symbolic model checking. 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 $μ ≡ r$ be a SISE and $v$ be a string variable in $μ$. The solution pool for $v$, denoted by $sp(v)$, is defined as $sp(v) = {ω | ω = ρ(v)$ where $ρ is a solution to $μ ≡ r}$.

Example 6.2. By Definition 3.3, a SISE solution $ρ$ is defined as a function $ρ : V^* → Σ^*$ where $V^*$ is always finite, hence $ρ$ is often written as a set of tuples. Let $x, y ∈ V$ and $α ∈ Σ$. Given $x ∘ y = α a a$, the following are all concrete solutions to the equation: $ρ_1 = {(x, ɛ), (y, a a)}$, and $ρ_2 = {(x, a), (y, a)}$, and $ρ_3 = {(x, a a), (y, ɛ)}).

This immediately leads to $sp(x) = {ɛ, a, aa}$ and $sp(y) = {ɛ, a, aa}$. 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 will 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 (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 Case

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

**Equivalence 1.** For any SISE of the form $\mu[i, j] \equiv r$ where $\mu \in E$ and $i, j \in N$ with $i \leq j$, $\rho$ is a solution to $\mu[i, j] \equiv r$ iff it is a solution to $\mu \equiv \Sigma^i(r \cap \Sigma^{j-1})\Sigma^*$. 

**Example 6.3.** Consider SISE $x[2, 4] \equiv ab^*$ where $x \in V$ and $a, b \in \Sigma$. Using Equivalence 1 we obtain $x \equiv \Sigma^2(ab^* \cap \Sigma^2)\Sigma^*$ and hence $sp(x) = \Sigma^2ab\Sigma^*$. Consider an arbitrary word in $sp(x)$, e.g., $x = ccabcc$. Let $\rho = \{(x, ccabcc)\}$. 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 $x_1r \equiv r_2$, where $x \in V$ and $r_1, r_2 \in R$. This can be solved using right quotient of regular expression [HU79]. By convention, the right quotient $r_2/r_1 = \{ x | xw \in r_2 \text{ and } w \in r_1 \}$. Similarly, the left quotient is defined as $r_2 \backslash r_1 = \{ x | wx \in r_2 \text{ and } w \in r_1 \}$. We know that if $r_1$ and $r_2$ are regular, then $r_2/r_1 \text{ and } r_2 \backslash r_1$.

Now consider the general case $\mu \nu \equiv r$ where both $\mu$ and $\nu$ are non-trivial string expressions. Let $\approx$ be the result of replacing every variable in $\nu$ with $\Sigma^*$. It follows that $\approx \nu$ is a regular expression. Given a solution $\rho$ and a string expression $\mu$, let $\rho_\mu$ represent the restriction of $\rho$'s domain to the variable set of $\mu$ (i.e., $\rho_\mu$ includes the tuples related to variables in $\mu$ only). We have Equivalence 2. It relies on Lemma 6.4.

**Lemma 6.4.** Let $\nu$ be a string expression on $\Sigma$. For any $w \in \approx \nu$ there is a solution to $\nu \equiv w$.

**Equivalence 2.** For any SISE of the form $\mu \nu \equiv r_2$, $\rho$ is a solution to $\mu \nu \equiv r_2$ iff it is a solution to $\mu \equiv r_2/\approx \nu$.

**Proof:** (Direction $\Rightarrow$) We need to prove that if $\rho$ is a solution to $\mu \nu \equiv r_2$, then $\rho_\nu$ is a solution to $\mu \equiv r_2/\approx \nu$. This can be inferred from the following facts: (L1) $\exists w \in r_2 \text{ s.t. } \rho(\mu) \cap \approx \nu = w$, (L2) $\rho(\nu) \in \approx \nu$. L1 and L2 imply that $\rho(\mu) \in r_2/\approx \nu$.

(Direction $\Leftarrow$) We need to prove that if $\rho_\nu$ is a solution to $\mu \equiv r_2/\approx \nu$, then $\rho_\mu$ can be extended into a solution to $\mu \nu \equiv r_2$. Let $w = \rho_\nu(\mu)$ and let $w_1 \in r_2/\approx w$. By Lemma 6.4, there always exists a solution to $\nu \equiv w_2$, and let it be $\rho_\nu$. Merging $\rho_\mu$ and $\rho_\nu$ leads to the desired solution to $\mu \nu \equiv r_2$ (this is possible because $\mu$ and $\nu$ have disjoint variable sets by Definition 3.4). \( \blacksquare \)

### 6.1.3. Replacement Case

The replacement case has a general form of $\mu_{r_1 \cdots \omega} \equiv r_2$ (similarly $\mu_{r_1 \cdots \omega}^- \equiv r_2$, $\mu_{r_1 \cdots \omega}^+ \equiv r_2$), where $\mu \in E$, $r_1, r_2 \in R$, and $\omega \in \Sigma^*$. In the following we discuss the solution to $\mu_{r_1 \cdots \omega} \equiv r_2$. 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 $M_{\mu_{r_1 \cdots \omega}} \Rightarrow r_2$ s.t. $(s, \eta) \in L(M_{\mu_{r_1 \cdots \omega}}) \iff \eta \in L(r_2)$ and $\eta \in s_{r_1 \cdots \omega}$. Let $M_1(r_2)$ be the FST that accepts the identity relation $\{(s, s) | s \in L(r_2)\}$. Let $M_{\mu_{r_1 \cdots \omega}}$ be the FST shown in Lemma 4.5, i.e., $(s, \eta) \in L(M_{\mu_{r_1 \cdots \omega}})$ iff $\eta \in s_{r_1 \cdots \omega}$. $M_{\mu_{r_1 \cdots \omega}}$ can then be constructed as $M_{\mu_{r_1 \cdots \omega}||M_1(r_2)}$. Similarly, for the pure reluctant semantics, $M_{\mu_{r_1 \cdots \omega}^+ \Rightarrow r_2}$ can be constructed as $M_{\mu_{r_1 \cdots \omega}^+||M_1(r_2)}$, where $M_{\mu_{r_1 \cdots \omega}^+}$ is defined in Theorem 5.20. $M_{\mu_{r_1 \cdots \omega}^- \Rightarrow r_2}$ can be defined similarly for the greedy semantics (using Theorem 5.39).

---

16 We abuse the notation here because $x_{r_1}$ 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. $x \circ w_1 = w_2$.

17 Here $\rho(\mu)$ represents the result of replacing in $\mu$ each variable $v$ with its value defined by $\rho$. 

---
1 function computeSolutionPool(μ ⊩ r)
2 switch (μ):
3 case x ∈ V: return \{(x, r)\}
4 case r₁ ∈ R: if L(r₁) \cap L(r) \neq \emptyset return \emptyset o.t. return ⊥
5 case μ[1, 2]: return computeSolutionPool(μ ⊩ (r₁ \cap r₁⁻¹)(Σ*)
6 case μ₁ →: return computeSolutionPool(μ ⊩ (r₁ \cap r₁⁻¹)(Σ*)
7 case μ₁ →: return computeSolutionPool(μ ⊩ (r₁ \cap r₁⁻¹)(Σ*)
8 case μ₁ →: return computeSolutionPool(μ ⊩ (r₁ \cap r₁⁻¹)(Σ*)
9 case μ₁ →: return computeSolutionPool(μ ⊩ (r₁ \cap r₁⁻¹)(Σ*)
10 case μν:
11 Let r₁ be approx(μ) and r₂ be approx(ν)
12 return computeSolutionPool(μ ⊩ r₁ ∪ r₂) \cup computeSolutionPool(ν ⊩ r₁\r₂)

Equivalence 3. For any SISE of the form μ₁ → ⊩ r₂ where μ ∈ E, r₁, r₂ ∈ R, and ω ∈ Σ*. ρ is a solution to μ₁ → ⊩ r₂ if and only if it is a solution to μ ⊩ r where L(r) = \{s | (s, η) ∈ L(M_{r₁ → x} ⇒ r₂)\}.

The L(r) in the above Equivalence can be computed by projecting M_{r₁ → x} ⇒ r₂ 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 M_{r₁ → y} ⇒ r₂ and M_{r₁ → y} ⇒ r₂.

6.2. Recursive Algorithm for Computing Solution Pool

Based on Equivalences 1, 2, and 3, one can develop a recursive algorithm for generating the solution pool for all variables in a SISE. The algorithm is shown in Figure 9. Function computeSolutionPool() returns a set of tuples, with each tuple representing a solution pool (note: not solution) for a variable. We use ⊥ to represent “no solution”. When applying any set operation (e.g., intersection and union) on ⊥, the result is ⊥. Note that ⊥ is not the same as empty set Ø.

Example 6.5. Consider the following SISE equation where x ∈ V and a, b, c ∈ Σ:

\[ x_{a+b}^+ \circ (y[2, 3]) : b^+ c \]

(2)

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

\[ x_{a+b}^+ \circ (\Sigma^*[2, 3]) : b^+ c \]

(3)

\[ \Sigma_{a+b}^+ \circ (y[2, 3]) : b^+ c \]

(4)

We tackle Equation 3 first. Using Equivalence 2, Equation 3 can be transformed into the following:

\[ x_{a+b}^+ \equiv b^+ c / \Sigma^*[2, 3] = b^+ c / \Sigma = b^+ \]

(5)

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

Theorem 6.6. For any SISE μ ⊩ r and any variable v in μ, the following are true:

1. Let ρ be the set of tuples returned by computeSolutionPool(μ ⊩ r). When ρ \neq ⊥, there exists one and only one tuple (v, x) for v in ρ and x = 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 E₁ and let E₂ be the equation resulted from any of the equation transformations outlined in Figure 9. A variable mapping \( g \) is a solution to E₁ iff it is a solution to E₂. (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 Figure 9, a variable is always assigned a regular solution pool. This leads to statement (2) in the theorem.

Theorem 6.7. The worst complexity of the algorithm in Figure 9 is $O(|\mu| \times 2^{8 \times 2^{2|\mu|}})$.

In computeSolutionPool the most expensive computation is the case $\mu^+_1 \Rightarrow r$, a composition of eight transducers. Among them, $A_3(r_1)$ is the largest transducer, whose size is $2^{2|r_1|}$. Since $|r_1| < |\mu|$, we can approximate the worst complexity of the $\mu^+_1 \Rightarrow r$ case as $O(2^{8 \times 2^{2|\mu|}})$. Finally, given $\mu \equiv r$, computeSolutionPool is called recursively for at most $|\mu|$ times. This leads to the complexity result in Theorem 6.7.

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 from 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 [Mo]. Listing 4 presents a sample program for solving a string equation shown below:

\[
\text{uname}=\circ \ x_{[0,5]} \circ \ \text{pwd}=\equiv \ \text{uname}=\left[\text{^1| \ '}'\right]\]

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.brics.Automaton.
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) $T$ 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. $T$ includes a set of transitions with the same source and destination states: $T = \{(q, q', a : b) \mid a \in \phi \text{ and } b \in \varphi\}$. If $|\phi| > 1$ and $|\varphi| > 1$, it is required that $\phi = \varphi$. For $\phi$ and $\varphi$, $\epsilon$ is represented using $[-1, -1]$. □

Excluding the $\epsilon$ transitions, there are three categories of SFTSes:

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

The top of Figure 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 \texttt{u0061} and \texttt{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 $(q, q', [u0000, u0060] : \epsilon)$ and $(q, q', [u007B, uFFFF] : \epsilon)$.
2. Identity relation $\text{Id}(\Sigma)$ can be represented using $(q, q', [u0000, uFFFF] : [u0000, uFFFF])$.
3. Type II SFTS $(q, q', [u0061, u007A])$ 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 $(q, q', [u0061, u007A])$ for short. □

The algorithms for supporting FST operations (such as union, Kleene star) should be customized correspondingly. In the following, we take FST composition as an example. Let $A = (\Sigma, Q, q, F, \delta)$ be the composition of $A_1 = (\Sigma, Q_1, q^0_1, F_1, \delta_1)$ and $A_2 = (\Sigma, Q_2, s^0_2, F_2, \delta_2)$. Given $t_1 = (t_1, t'_1, \phi_1 : \varphi_1)$ in $A_1$ and $t_2 = (t_2, t'_2, \phi_2 : \varphi_2)$ in $A_2$, where $\varphi_1 \cap \phi_2 \neq \emptyset$, an SFTS $\tau = (s_1, s_2, \phi : \varphi)$ is defined for $A$ s.t. $s_1 = (t_1, t_2)$, $s_2 = (t'_1, t'_2)$, and the input/output charset of $\tau$ is defined following the table in Figure 11. For all cases except for (I,II), the algorithm produces one SFTS given a pair of SFTSes from $A_1$ and $A_2$. For example, when both $t_1$ and $t_2$ 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 Figure 10 shows the intuition of the algorithm. The dashed circles represent the corresponding input/output charset.
<table>
<thead>
<tr>
<th>Type of τ₁</th>
<th>Type of τ₂</th>
<th>input of τ</th>
<th>output of τ</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>I</td>
<td>φ₁</td>
<td>φ₂</td>
</tr>
<tr>
<td>II</td>
<td>I</td>
<td>φ₁</td>
<td>φ₂</td>
</tr>
<tr>
<td>III</td>
<td>I</td>
<td>φ₁ ∩ φ₂</td>
<td>φ₂</td>
</tr>
<tr>
<td>I</td>
<td>II</td>
<td>{x} where x ∈ φ₁</td>
<td>{y} where y ∈ φ₂</td>
</tr>
<tr>
<td>II</td>
<td>II</td>
<td>φ₁</td>
<td>φ₂</td>
</tr>
<tr>
<td>III</td>
<td>II</td>
<td>φ₁ ∩ φ₂</td>
<td>φ₂</td>
</tr>
<tr>
<td>I</td>
<td>III</td>
<td>φ₁</td>
<td>φ₁ ∩ φ₂</td>
</tr>
<tr>
<td>II</td>
<td>III</td>
<td>φ₁</td>
<td>φ₁ ∩ φ₂</td>
</tr>
<tr>
<td>III</td>
<td>III</td>
<td>φ₁ ∩ φ₂</td>
<td>φ₁ ∩ φ₂</td>
</tr>
</tbody>
</table>

**Fig. 11.** FST Composition Algorithm for SFTS Encoding

<table>
<thead>
<tr>
<th>Type</th>
<th>SFTS of A₁</th>
<th>SFTS of A₂</th>
<th>Resulting SFTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I, I)</td>
<td>(q, q', [a, b] : [c, c])</td>
<td>(q, q', [a, b] : [e, e])</td>
<td>(q, q', [a, b] : [c, e])</td>
</tr>
<tr>
<td>(II, I)</td>
<td>(q, q', [a, a] : [c, d])</td>
<td>(q, q', [a, a] : [c, e])</td>
<td>(q, q', [a, a] : [c, e])</td>
</tr>
<tr>
<td>(III, I)</td>
<td>(q, q', [a, b] : [a, b])</td>
<td>(q, q', [b, b] : [c, e])</td>
<td>(q, q', [a, b] : [c, e])</td>
</tr>
<tr>
<td>(I, II)</td>
<td>(q, q', [a, b] : [c, c])</td>
<td>(q, q', [c, c] : [d, e])</td>
<td>(q, q', [c, c] : [d, e])</td>
</tr>
</tbody>
</table>

**Fig. 12.** SFTS Composition Examples

**Example 7.3.** Figure 12 lists four examples of the application of the SFTS composition algorithm. In each example, A₁ and A₂ are both single-transition finite state transducers. The resulting FST has one SFTS for all except the (I, II) case.

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 1GB RAM, running in an Oracle VirtualBox. The host system is a Lenovo W700 laptop with a 3.06GHz Intel Core2 CPU and 4GB 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 Kudzu tool website [SAM +10].

Kaluza dataset consists of four categories of constraints: 19986 small satisfiable constraints (referred as SAT small in our later discussion), 1836 big satisfiable constraints (SAT big), 11762 small unsatisfiable constraints (SAT small), and 21470 big unsatisfiable constraints (UNSAT big). Each Kaluza constraint is essentially a conjunction of atomic string constraints, though in practice the SUSHI constraint 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. 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 $\mu$ 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 Figure 9 remains the same, except that the approximation of $\mu$ is now $\text{approx}(\mu) \cap R(\mu)$. 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 [Ome94]. 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 (categorized as IntProj failure in the data analysis).
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 (categorized as SingleOcc failure).

Example 8.1. Consider a simple Kaluza constraint presented on the left of Figure 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 Figure 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 $I_0 : |T_3| = |T_5|$. 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.

$$I_1 : |T_3| = |T_2| + |T_3| \land |T_4| = |T_5| + |T_6| \land \bigwedge_{1 \leq i \leq 6} |T_i| \geq 0.$$  

Then $I_0 \land I_1$ is projected to each string variable using existential elimination. For example, for variable

Fig. 13. Conversion Algorithm from Kaluza to SISE
Success.

Example 8.2. The Kaluza example named Kaluza_1 in Figure 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

From each of the four Kaluza datasets, we randomly select 400 constraints for the experiment. Table 1 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 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 and some rare cases of Omega constraint solver crash. Most of exceptions are the time-out on Omega solvers when existential elimination operations are performed to project integer constraints for one Kaluza input. Omega calculator is forced to terminate after 300 seconds.

The conversion algorithm works very well for the SAT_small and UNSAT_small datasets, with over 96% of the small constraints successfully converted. Only about 66% 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 31%), and more than 11% of the UNSAT_big constraints are identified to be unsatisfiable at the integer constraint projection stage.

Table 2 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. 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, 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 huge because of the existential elimination, which is 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 other string operations such as replacement and substring) 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 great cost of integer constraint projection during the conversion process. A more reasonable comparison in terms of constraint solving cost is presented in §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 15 lists five SISE equations for stress-testing the SUSHI package and the running statistics. Note that each equation is parametrized by an integer $n$ (ranging from 1 to 30). For example, when $n = 30$, the size of the RHS of eq4 is 60.

The sample set covers all string operations we discussed earlier. In Figure 16, the first two diagrams present the size of the FST (the number of states and the number of transitions) used in the solution used in the solution.
<table>
<thead>
<tr>
<th>ID</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq1</td>
<td>$x \circ a{n, n} \equiv (a{2n, 2n})$</td>
</tr>
<tr>
<td>eq2</td>
<td>$x[2n, 2n] \equiv a{n, n}$</td>
</tr>
<tr>
<td>eq3</td>
<td>$x \circ a \circ y[0, n] \equiv b{n, n}ab{n, n}$</td>
</tr>
<tr>
<td>eq4</td>
<td>$x^+ \rightarrow b{n, n} \equiv b{2n, 2n}$</td>
</tr>
<tr>
<td>eq5</td>
<td>$uname^* \circ x_{r,n}[0, n] \circ pwd^* \equiv uname^<em>[^</em>[^<em>[^</em>[^<em>[^</em>]]]]]]$</td>
</tr>
</tbody>
</table>

Fig. 15. Sample SUSHI Equations

Fig. 16. Constraint Solving Cost Using SUSHI

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 Figure 16, 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 Figures 16(a) and 16(b)) indicate that to model greedy regular replacement (e.g., eq4) is expensive because the modeling process involves composition of seven transducers.

At this moment, we are not able to explain the abnormal curve of the cost of time at the coordinate $n = 23$ (Figure 16(d)). We suspect that this is caused by the garbage collection of the virtual machine.

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 Figure 15 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 $n$, $s^+_r \omega \in r_2$ 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 $A_t$ is used in both $s$ and $t$. 

with a working exploit. This could be caused by several reasons: (1) the poster would not like to provide security analysis. The application is motivated by the fact that most of the vulnerability reports do not come hard to solve). The search algorithm (e.g., adopted by Kaluz a) works better in other cases.

The solution space of $S$ (representing all possible concrete solution values for $x$ when the shortest solution is found. For $S$, re-generated SISE constraints) is often smaller than $S$ re-generated SISE equations from the Kaluza input.

As shown in Table 3, the SUSHI constraint solving cost ($C_1$) on the five sample equations are compatible with the ones shown in Figure 16. 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 we use a special algorithm for computing concrete solutions for $S_2$ without the use of 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.

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

3. We then convert the Kaluza specification again back to SUSHI and obtain its running cost. This is to compare with the performance of SUSHI on the original SISE equation.\(^{21}\)

Table 3 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 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 Table 3 denotes time out.

As shown in Table 3, the SUSHI constraint solving cost ($S_1$) on the five sample equations are compatible with the ones shown in Figure 16. 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 we use a special algorithm for computing concrete solutions for $S_2$ without the use of solution pool. Take eq4 as an example. Consider eq4: $x_{a^+} = b\{2n, 2n\}$. 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 with a big margin on eq5 (Kaluza is timed out starting from $n = 9$). This is as expected. Recall that eq5 is a simplification of the password bypassing attack in §2. The solution space of $x$ is very small ($x$ has to contain the right number of single quotes to get it chopped by the substring operation).

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

\(^{21}\) In many cases, the conversion algorithm will fail on SingleOcc. We modify the conversion algorithm to ignore the error and continue with the conversion.
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).

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 Figure 17. They are reusable patterns for characterizing a working XSS exploit.

1. The XSS pattern in Figure 17 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).

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 files against schema [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]) fall into the category of forward analysis. The work presented in this paper is backward.

---

<table>
<thead>
<tr>
<th>Rule</th>
<th>Regular Expression Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSS</td>
<td>.<em>&lt;script&gt;alert('XSS found!')&lt;/script&gt;</em></td>
</tr>
<tr>
<td>EffectiveScript</td>
<td>.<em>[a-zA-Z0-9]</em> = &quot;[^&quot;]<em>&lt;script.</em>&gt;.*</td>
</tr>
<tr>
<td>MatchTag</td>
<td>.<em>&lt;embed[^&gt;]</em>&gt;.* ∩ .<em>&lt;/embed[^&gt;]</em>&gt;.*</td>
</tr>
</tbody>
</table>

Fig. 17. Rules in RHS
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 Mohri-Nederhof algorithm [MN01] to approximate from context-free languages to regular. 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 vulnerability, 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 of a vulnerability. It suffers from false-negatives, i.e., there are cases that vulnerabilities are ignored by the analysis. It is 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 \texttt{java.regex} operations are supported. This determines that the traditional technique for solving word equations, such as 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 \( \forall \exists \) theory of concatenation and word length predicates (according to [Lot02]) are 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 HAMPJ string constraint solver [KGG+09], which also supports a backward analysis. HAMPJ solves string constraints with context-free components, which are essentially unfolded to regular language within a certain bound. HAMPJ, however, does not support string replacement nor regular replacement, which limits its ability to reason about sanitation 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 §2. The BEK solver [HLM+10] supports an imperative language for modeling string sanitation 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, BTMV09, 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, 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, Bjorner, and 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,Mol97,KpCGS96] in computational linguistics, for processing phonological and morphological rules. In [KpCGS96], 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 the 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 more efficient. In practice, SUSHI adopts a symbolic representation (range of character set) of transitions in a finite state transducer, 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 that it is possible to construct a context-free grammar to model the approximation of regular replacement. 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 is 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 the 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 engine 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 [NTGG05] 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., [HHL03]) 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 grader for web application projects used in computer science education.

Acknowledgment: We would like to thank Fang Yu, Tevfik Bultan, and Oscar Ibarra for the constructive discussion that inspired this research. We are grateful to the anonymous reviewers for the constructive comments on the experimental evaluation. Prateek Saxena has provided great help in answering our questions on Kaluza.


A. Regular Replacement with \( \epsilon \) in Search Pattern

Replacement with \( \epsilon \) in search pattern is often counter intuitive. For example, "a".replaceAll("a*?","b") in Java yields \( bab \). Here the \( a \) in the input word is not replaced! It is valuable to formally model such rare cases of regular replacement, and it helps to perform thorough program analysis on sanitation 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 \( r = a^* \), \( a_{\text{\texttt{-}ba}} \) inspects indices 0 and 1, finds two matches of \( r \) (i.e., \( \epsilon \)), 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 \texttt{java.util.regex}.

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

\[
\begin{align*}
\tilde{s}_{r-\omega} &= \{ \omega \text{ where } s = a\mu \text{ and } a \in \Sigma \} \text{ if } s = \epsilon. \\
&= \{ \omega \mu \text{ if } s = \epsilon \text{ and } \omega \not\in \Sigma \} \text{ otherwise.}
\end{align*}
\]

\[\square\]

**Example A.2.** Consider the following three cases, all of which are verified using \texttt{java.util.regex}. (i) If \( s = aa, r = a^* \), and \( \omega = c \), then \( \tilde{s}_{r-\omega} = cacac \). (ii) If \( s = ba, r = a^* \), and \( \omega = c \), then \( \tilde{s}_{r-\omega} = cb cac \). (iii) If \( s = ba, r = b|a^* \), and \( \omega = c \), then \( \tilde{s}_{r-\omega} = cb cac \).

Lemmas A.3 and A.4 follow directly from Definitions A.1 and 5.12.

**Lemma A.3.** Let \( \mu, \eta, \omega \in \Sigma^* \), \( n = |\mu| \), and \( r \in R \) with \( \epsilon \in r \). \( \eta \) is the \( r \)-begin-marked output of \( \mu \) iff \( |\eta| = 2|\mu| + 1 \), and \( \forall_i 0 \leq i < |\mu| : |\eta|[2i] = \# \land \eta[2i+1] = \mu[i] \), and \( |\eta| - 1 = \# \).

**Lemma A.4.** Let \( \mu, \eta, \omega \in \Sigma^* \), \( n = |\mu| \), and \( r \in R \) with \( \epsilon \in r \). \( \mu_r - \omega = \eta \) iff \( \eta \) can be written as \( \eta_1 \circ \cdots \circ \eta_n \circ \omega \) where \( \forall_i 1 \leq i \leq n : \eta_i = \omega \mu[i] \).

Similar to Lemma 5.19, we have the following lemma for the reluctant replacement transducer \( \mathcal{A}_\epsilon(r, \omega) \) 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 \( \zeta \in \Sigma^* \): \( (\kappa, \zeta) \in L(\mathcal{A}_\epsilon(r, \omega)) \) iff \( \kappa = \# \land \zeta = \omega \), or both of the following are true:

- (F1') \( \kappa \) can be written as \#|a\mu| such that \( a \in \Sigma \).
- (F2') \( \zeta \) can be written as \( \omega \eta \), and \( (\mu, \eta) \) is accepted by \( \mathcal{A}_\epsilon(r, \omega) \).

The definition of greedy replacement with \( \epsilon \) 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 \( \epsilon \in r \). \( s^+_{r-\omega} \) is defined using one of the following three rules:

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

\[\square\]

**Example A.7.** Consider the following three cases, all of which are verified using \texttt{java.util.regex}. (i) If \( s = aa, r = a^* \), and \( \omega = c \), then \( s^+_{r-\omega} = cc \). (ii) If \( s = ba, r = a^* \), and \( \omega = c \), then \( s^+_{r-\omega} = cbcc \). (iii) If \( s = ba, r = b|a^* \), and \( \omega = c \), then \( s^+_{r-\omega} = cccc \).

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

\[\text{Here } \bullet ? \text{ represents the reluctant semantics of the } \ast \text{ operator.}\]
Definition A.8. Let \( r \in R \) with \( \epsilon \in r \), for any \( \kappa \in \Sigma^* \) and \( \zeta \in \Sigma^* \), \( \zeta \) is said to be a \( r \)-greedy-marking of \( \kappa \) iff one of the following three conditions is satisfied:

1. If \( \kappa = \epsilon \), then \( \zeta = \#\$ \).
2. If \( \kappa \in (r - \epsilon)^* \), 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 \( \eta \) be a \( r \)-greedy-marking of \( \mu \). Then \( \zeta = \#\beta\$\eta \).
3. If \( \kappa \neq \epsilon \land \kappa \notin (r - \epsilon)^* \), let \( \kappa = a\mu \) where \( a \in \Sigma \), and \( \eta \) be a \( r \)-greedy marking of \( \mu \). Then \( \zeta = \#\$a\eta \).

\[ \square \]

B. Complete Proofs in Section 5

B.1. Proof of Lemma 5.3

Lemma 5.3 For any \( r \in R \), \( \mu \in \Sigma^* \) and \( \eta \in (\Sigma \cup \{\$\})^* \), \( (\mu, \eta) \in L(A_1(r)) \) iff all the following are satisfied:

1. \( \mu \in L(r) \); 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 \( (\mu, \eta) \in L(A_1(r)) \), condition (1) can be inferred by converting an acceptance run of \( (\mu, \eta) \) on \( A_1 \) to an acceptance run of \( \mu \) on \( DFSA(r) \). Condition (2) results from the fact that 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 \( A_1(r) \). This is because the destination state of a \( (\epsilon, \$) \) transition in \( A_1 \) (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 \( A_1(r) \)).

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 \( \Sigma \), and the partial run of \( (\pi_\Sigma(\eta[0, i]), \eta[0, i]) \) 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.

Proposition A.1: Let \( A_1 = (\Sigma \cup \{\$\}, Q_1, q_0', F_1, \delta_1) \), and \( q_0' \) be the initial state of \( DFSA(r) \). If \( (\mu, \eta) \) satisfies conditions 2 and 3, then there exists \( q_j' \in Q_1 \) s.t. \( q_0' \sim_{(\mu, \eta)} q_j' \) in \( A_1(r) \). If \( \eta[|\eta| - 1] = \$ \), then \( q_j' \in F_1 \); otherwise, \( q_j' \) is the state imported from \( DFSA(r) \) in the construction algorithm of \( A_1 \), and \( q_0' \sim_{(\mu, \eta)} q_j' \) in \( DFSA(r) \).

A1 can be proved by induction on the length of \( \eta \). 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, \( (\mu, \eta) \) is accepted by \( A_1 \). This completes the proof of Direction \( \Leftarrow \).

B.2. Proof of Lemma 5.6

Lemma 5.6 For any \( r \in R \) and \( \mu \in \Sigma^* \) there exists one and only one \( r \)-end-marked output.

Proof: We prove by induction on the length of \( \mu \). 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 \( \zeta \) is the \( r \)-end-marked output of \( \nu \). Then, we construct \( \mu \)'s \( r \)-end-marked output (let it be \( \eta \)) as follows. If \( \mu \in \Sigma^*r \), then \( \eta = \zeta a\$ \); otherwise \( \eta = \zeta a \).

Now we prove by contradiction that \( \eta \) is the only \( r \)-end-marked output for \( \mu \). Assume that there is another \( r \)-end-marked output of \( \mu \) (letting it be \( \eta' \)). It follows from Definition 5.5 that \( \pi_\Sigma(\eta) = \pi_\Sigma(\eta') = \mu \). Let \( \omega \) be the longest common prefix of \( \eta \) and \( \eta' \), i.e., there exists \( \omega_1 \) and \( \omega_2 \) s.t. \( \eta = \omega_1 \eta_1 \) and \( \eta' = \omega_2 \eta_2 \). Let \( n = |\omega| \). Now think about \( \eta[n] \), i.e., the first letter of \( \omega_1 \). There are two cases to consider: (i) If \( \eta[n] \in \Sigma \), it follows that \( \eta'[n] \) has to be the same as \( \eta[n] \) (otherwise \( \pi_\Sigma(\eta) \neq \pi_\Sigma(\eta') \)). However, when \( \eta[n] = \eta'[n] \), it

\[ 24 \text{ Note that } \eta[i - 1] \text{ is the last letter of } \eta[0, i]. \]
conflicts with the assumption that \( \omega \) is the longest common prefix. (ii) If \( \eta[n - 1] \neq \$ \), according to condition (2) of Definition 5.5, \( \pi_2(\omega) \in \Sigma^*r \). It leads to \( \eta'[n] = \$, which again conflicts with the assumption on \( \omega \). Thus, the proof is complete.

**B.3. Proof of Lemma 5.9**

**Lemma 5.9** For any \( r \in R \), let \( A_1(r) = (\Sigma \cup \{ \$ \}, Q_1, q_1^0, F_1, \delta_1) \), and \( A_2(r) = (\Sigma \cup \{ \$ \}, Q_2, q_2^0, F_2, \delta_2) \). For any \( t \in Q_2 \), any \( \mu \in \Sigma^* \), and any \( \eta \in (\sigma \cup \{ \$ \})^* \) s.t. \( q_0^0 \sim_{(\mu, \eta)}^* t \) in \( A_2(r) \), both of the following are true:

1. For any \( q_1^0 \in B(t) \), there exists \( \nu \preceq \mu \) and \( \zeta \in (\Sigma \cup \{ \$ \})^* \) s.t. \( q_0^0 \sim_{(\nu, \zeta)}^* q_1^0 \) in \( A_1(r) \).
2. For any \( \nu \preceq \mu \) s.t. \( \nu \in \text{PREFIX}(r) \), there exists \( \zeta \in (\Sigma \cup \{ \$ \})^* \) and \( q_1^0 \in B(t) \) s.t. \( q_0^0 \sim_{(\nu, \zeta)}^* q_1^0 \) in \( A_1(r) \).

**Proof:** We prove by induction on the length of \( \eta \). The base case vacuously holds for both statements, because \( B(q_0^0) = \{ q_0^0 \} \) and \( q_0^0 \sim_{(\epsilon, \epsilon)} q_0^0 \) in \( A_1(r) \).

**Proof of Statement 1:** Given \( q_1^0 \in B(t) \), the objective is to find the pair \((\nu, \zeta)\) that reaches \( q_1^0 \) in \( A_1(r) \). The proof idea is presented in Figure 18(A).

Consider the last transition in the partial run of \((\mu, \eta)\) on \( A_2(r) \). If it is a Case 1 transition, let it be \((q_y^0, t, a : a)\). We can safely write \( \mu \) and \( \eta \) as \( \mu = \alpha a \), and \( \eta = \beta a \). Now by case 1 Rule, for \( q_1^0 \) there must be a \( q_y^0 \in B(q_1^0) \) s.t. \( (q_y^0, q_1^0, a : a) \in \delta_1 \). 26 Since \( q_0^0 \sim_{(\alpha, \beta)}^* q_1^0 \) in \( A_2(r) \), the induction assumption applies, i.e., we can find a pair \((\theta, \kappa)\) s.t. \( q_0^0 \sim_{(\theta, \kappa)}^* q_y^0 \). Now \((\theta a, \kappa a)\) is what we need because \( q_0^0 \sim_{(\theta, \kappa)}^* q_y^0 \sim_{(a, a)}^* q_1^0 \), and hence, \( q_0^0 \sim_{(\theta a, \kappa a)}^* q_1^0 \). Similar argument applies to Case 2 Rule.

**Proof of Statement 2:** The key is to find the \( \zeta \) and \( q_1^0 \) in the statement.

If the last element of \( \eta \) is \$ (Case 2), we let \( \eta = \beta \$. Similarly (to the proof of statement 1), we can infer that there exists \( q_y^0 \in Q_2 \) s.t. \( q_0^0 \sim_{(\alpha, \beta)}^* q_y^0 \sim_{(\epsilon, \epsilon)}^* t \). The scheme of the proof is shown in Figure 18(B). Given \( \nu \preceq \alpha \), by applying the induction assumption we now have a \( q_y^0 \in B(q_0^0) \) and \( \kappa \in (\Sigma \cup \{ \$ \})^* \) s.t. \( q_0^0 \sim_{(\nu, \zeta)}^* q_y^0 \) in \( A_1(r) \). Now, if \( q_y^0 \notin F_0 \), according to Case 2 Rule of \( A_2(r) \), it is also contained in \( B(t) \). Then \( \kappa \) and \( q_1^0 \) already satisfy our needs (i.e., \( \kappa = \zeta \) and \( q_1^0 \) is the \( q_1^0 \) we are trying to find for statement 2). If \( q_y^0 \in F_0 \), then there must be a \( q_1^0 \) s.t. \( (q_1^0, q_1^0, \epsilon : \$) \in \delta_1 \). In this case, \( \kappa \$ \) and \( q_1^0 \) satisfy our needs for the proof, i.e., \( q_0^0 \sim_{(\nu, \zeta)}^* q_1^0 \) and \( q_1^0 \in B(t) \).

When \( \eta[|\eta| - 1] \in \Sigma \) (Case 1), the proof is more complex and it needs the fact that \( \nu \) is a prefix of a word in \( r \). Let \( \mu = \alpha a \) and \( \eta = \beta a \), similarly we have \( q_1^0 \in Q_2 \) s.t. \( q_0^0 \sim_{(\alpha, \beta)}^* q_1^0 \sim_{(\epsilon, \epsilon)}^* t \). We use the notations in Figure 18(A) in the following proof.

For any \( \nu \preceq \mu \) (which is a prefix of a word in \( r \)), We can write it as \( \nu = \theta \alpha \). By the induction assumption, there exists \( q_0^0 \in B(q_0^0) \) and \( \kappa \in (\Sigma \cup \{ \$ \})^* \) s.t. \( q_0^0 \sim_{(\theta, \kappa)}^* q_1^0 \). Now since \( \nu \) is a prefix of a word in \( r \), \( \theta \alpha \) has a partial run on DFSA\((r)\), and the partial run of \( \theta \) is a prefix of it. Thus, let \( q_0^0 \) and \( \delta_0 \) be the initial state and transition relation of DFSA\((r)\), there exists \( q_1^0 \) s.t. \( q_0^0 \sim_{(\delta, \delta)}^* q_1^0 \) in DFSA\((r)\). Now the question

25 Notice that the combination of the two conditions is not equivalent to the statement that \( B(t) \) is always equal to \( \{ q_1^0 \mid \exists \nu \preceq \mu, \zeta \in (\Sigma \cup \{ \$ \})^* s.t. q_0^0 \sim_{(\nu, \zeta)}^* q_1^0 \} \).

26 We only consider the case that \( q_1^0 \) is not the initial state \( q_0^0 \) of \( A_1(r) \). When \( q_1^0 = q_0^0 \), readers can verify that the claim vacuously holds.

![Figure 18](image-url)
is: would $q_i^0$ be the same as $q_i^1$? By the construction algorithm of $A_1(r)$, if $\theta \not\in r$, we can conclude that $q_i^1$ is the $q_k^0$ during the construction of $A_1(r)$. If $\theta \in r$, we can infer that $q_k^0 \in F_0$, and $q_k^0$ is the corresponding final state in $F_1$. ($q_k^1$ cannot be contained in $F_0$, otherwise the state $q_i^2$ would not have an out-going $(a, a)$, according to Case 2 Rule, in $A_2(r)$.) Thus, there is a transition $(a : a)$ from $q_i^0$ to $q_i^0$ in $A_1(r)$. Note that $q_k^0$, a state in DFSA(r), is imported to the state set of $A_1(r)$ according to the construction algorithm. In the following, we write it as $q_i^1$ to indicate that it is a state in $A_1(r)$. Finally, we have $q_i^0 \sim_{(\theta, c)} q_i^0 \sim_{(a, a)} q_i^1$, and $q_i^1 \in B(t)$. This completes the proof of statement 2.

\[ \square \]

### B.4. Proof of Lemma 5.10

**Lemma 5.10** For any $r \in R$, any $\mu \in \Sigma^*$, and any $\eta \in (\Sigma \cup \{\}$)*: $(\mu, \eta) \in L(A_2(r))$ iff $\eta$ is the r-end-marked output of $\mu$.

**Proof:** The proof by induction will be split into two parts (one for each direction). We assume that when $|\mu| + |\eta| \leq n$, $(\mu, \eta) \in L(A_2(r))$ iff $\eta$ is the r-end-marked output of $\mu$. The base case for $r \not\in R$, i.e., $\mu = \eta = \epsilon$ vacuously holds, because $(\epsilon, \epsilon) \in L(A_2(r))$. The base case for $r \in R$, i.e., $\mu = \eta = $, also holds as the word pair is accepted by $A_2(r)$.

(Direction $\Rightarrow$:) We show that if $\eta$ is the r-end-marked output of $\mu$, then $(\mu, \eta) \in L(A_2(r))$.

For the inductive step, similar to the proof of Lemma 5.9, we need to consider two cases, based on the last element of $\eta$. 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 $\beta a$ is the r-end-marked output of $\mu$, by Definition 5.5, $\pi_\Sigma(\beta a) = \mu$. Thus the last element of $\mu$ is $a$, and let $\mu = \alpha a$. We can further infer that $\beta$ is the r-end-marked output of $\alpha$. This leads to $(\alpha, \beta) \in L(A_2(r))$, using the inductive assumption. Therefore, there exists $q_i^2 \in F_2$ s.t. $q_i^0 \sim_{(a, a)} q_i^2$.

We now prove that $B(q_i^2)$ does not contain any element in $F_0$ by contradiction. Assume $B(q_i^2) \cap F_0 \neq \emptyset$, according to Case 2 Rules, there is an ($\epsilon, $) transition that leads to a final state of $A_2(r)$. This implies that $(\alpha, \beta) \in L(A_2(r))$. Now by the induction (on the combined length of $\alpha$ and $\beta$ and using the $\Rightarrow$ direction), $\beta$ is the r-end-marked output of $\alpha$. This contradicts with the fact that $\beta a$ is the r-end-marked output of $\alpha$, as on the index $[|\beta|]$, $\eta$, 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 $B(q_i^2) \cap F_0 = \emptyset$.

Now since $B(q_i^2) \cap F_0 = \emptyset$, Case 1 Rule applies. Therefore, there must exist a $t \in Q_2$ s.t. $((t, a : a) \in \delta_2$. Note that we still need to prove that $t \in F_2$ (i.e., $B(t) \cap F_0 = \emptyset$). This could be achieved using contradiction (showing that $\eta$ is not r-end-marked output of $\mu$ because another $=$ is needed at the end of $\eta$, using Lemma 5.9). In summary, we have constructed the following acceptance run for $(\mu, \eta) = (\alpha a, \beta a)$:

$q_i^1 \sim_{(a, a)} q_i^2 \sim_{(a, a)} t$

Thus, $(\mu, \eta) \in L(A_2(r))$. The case (when $|\eta|[|\eta| - 1] = $) can be proved similarly.

(Direction $\Rightarrow$:) We show that if $(\mu, \eta) \in L(A_2(r))$ then $\eta$ is the r-end-marked output of $\mu$.

According to Definition 5.5, we need to prove that $\eta$ 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 $\pi_\Sigma(\eta) = \mu$, then $\eta[|\eta| - 1] = $.

(c1) is immediately available based on the fact that there are only two types of transitions in $A_2(r)$: (b, b) for $b \in \Sigma$, and (e, $\epsilon$).

(c2) is established as follows. Consider $\pi_\Sigma(\eta[0, i]), \eta[0, i]$), which is a prefix of $(\mu, \eta)$. Since $A_2(r)$ 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 $(\mu, \eta)$. Thus, there is a $q_i^2 \in Q_2$ s.t. $q_i^0 \sim_{(\pi_\Sigma(\eta[0, i]),,\eta[0, i])} q_i^2$ in $A_2(r)$.

Now using Lemma 5.9(2), we can infer that there exists $q_j^1 \in B(q_i^2), \nu \in r$ (because $\pi_\Sigma(\eta[0, i]) \in \Sigma^r$), and $\zeta \in (\Sigma \cup \{\})^+$ s.t. $q_i^0 \sim_{(\pi_\Sigma(\eta[0, i]),,\eta[0, i])} q_i^2$ in $A_1(r)$. By the construction algorithm of $A_1(r)$, 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 $(\nu, \zeta)$ on $A_1(r)$ would be $(\epsilon, $). This conflicts with the fact that $\eta[|\eta| - 1] \neq $.

Now since $q_j^1 \in F_0 \cap B(q_i^2)$, the only transition out of $q_j^1$ is $(\epsilon, $). This concludes the proof for $\eta[|\eta| - 1] = $.
(c3) is proved similarly as (c2). We first show that there exists a run \( q_i^0 \leadsto^*_{\pi_\Sigma([\eta([0,i]),\eta([0,i])]} q_i^0 \leadsto^*_{(\epsilon,\$)} q_i^0 \). Then using Lemma 5.9(1), we show that \( \pi_\Sigma([\eta([0,i]),\eta([0,i])]} \in \Sigma^r \). The rest of the proof is to show that there will never be two consecutive \( (\epsilon,\$) \) transitions in \( A_2(r) \) (which leads to \( \eta[i-1] \neq \$ \)). This can be inferred from the fact that for any \( t \in F_2 \cap B(t) \cap F_1 = \emptyset \). Thus, the destination state of an \( (\epsilon,\$) \) transition cannot be the source state of another \( (\epsilon,\$) \) transition in \( A_2(r) \).

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

B.5. Proof of Theorem 5.20

**Theorem 5.20** Given any \( r \in R \) and \( \omega \in \Sigma^* \), and let \( M_{r-\omega} \) be \( A_3(r)||A_4(r,\omega) \), then for any \( \kappa,\zeta \in \Sigma^* : \) \( (\kappa,\zeta) \in L(M_{r-\omega}) \) iff \( \zeta = \kappa_{r-\omega} \).

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 \( (\kappa,\zeta) \in L(M_{r-\omega}) \) then \( \zeta = \kappa_{r-\omega} \). The base case where \( \kappa \in \Sigma^* - \Sigma^r \Sigma^* \) is obvious. We now study the case where \( \kappa \) has at least one match of \( r \). We prove by induction on the number of matches of \( r \) in \( \kappa \).

According to Definition 4.2, there exists \( \theta \) s.t. \( (\kappa,\theta) \in L(A_3(r)) \) and \( (\theta,\zeta) \in L(A_4(r,\omega)) \). By Lemma 5.13, \( \theta \) must be the \( r \)-begin-marked output of \( \kappa \), thus \( \pi_\Sigma(\theta) = \kappa \). Then by applying Lemma 5.19, we have \( \theta \) and \( \zeta \) satisfying conditions F1 and F2, i.e., they can be written as:

\[ \theta = \nu\#\beta\mu, \quad \zeta = \nu\#\omega\eta \]

Since \( \pi(\theta) = \kappa \) and \( \nu \in \Sigma^* - \Sigma^r \Sigma^* \), \( \kappa \) could also be written as

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

Now we try to prove \( \kappa_{r-\omega} = \zeta \) by showing that \( \kappa \) and \( \zeta \) 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 \( \beta \in \text{re}luc(r_{\#}) \), by Lemma 5.19.
3. for every \( \pi,\gamma,\nu,u,t,m,n \) 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 Equation 6, there exists \( x',y',u',t',m',n' \) s.t. \( x = x', y = y', u = \pi_\Sigma(u'), t = \pi_\Sigma(t'), m = \pi_\Sigma(m'), n = \pi_\Sigma(n') \), and \( \nu = x'y\beta = u't' \). By F1 of Lemma 5.19, we have if \( t' \neq \epsilon \), then \( u' \notin r_{\#} \). Actually it can be strengthened to the following statement (letting it be H1) “if \( \pi_\Sigma(t') \neq \epsilon \), then \( u' \notin r_{\#} \).” To see why, consider the case \( t' \neq \epsilon \). \( t' \) cannot start with \( \# \) because there is a preceding \( \# \) before \( \beta \) and \( \theta \) is \( r \)-begin-marked. Thus \( t' \) must start with a letter in \( \Sigma \) and \( \pi_\Sigma(t') \neq \epsilon \). H1 leads to (ii) because \( t = \pi_\Sigma(t') \), \( u = \pi_\Sigma(u') \), and \( r = \pi_\Sigma(r_{\#}) \). (i) is proved similarly.

4. \( \eta = \pi_\Sigma(\mu)_{r-\omega} \). The proof relies on the induction assumption. Using Lemma 5.19 we have \( (\mu,\eta) \in L(A_4(r,\omega)) \). Then, we just need to show \( (\pi_\Sigma(\mu),\mu) \in L(A_3(r)) \) so that \( (\pi_\Sigma(\mu),\eta) \in L(M_{r-\omega}) \) which leads to \( \eta = \pi_\Sigma(\mu)_{r-\omega} \) using the induction assumption.

We prove that \( \mu \) is the \( r \)-begin-marked output of \( \pi_\Sigma(\mu) \) (i.e., \( (\pi_\Sigma(\mu),\mu) \in L(A_3(r)) \). We need to show that both E1 and E2 of Definition 5.12 hold. E1 (every \( \# \) is followed by a match of \( r \) and no consecutive \( \# \) signs) is implied by the fact that \( \theta \) is \( r \)-begin-marked (where \( \mu \) is its suffix). For E2, we need to show that \( \mu[0] \neq \# \) if \( \mu \in r \Sigma^* \). This can be proved by contradiction. Assume that \( \mu[0] \neq \# \) but \( \mu \in r \Sigma^* \). Since \( \theta \) is \( r \)-begin-marked, it follows that the \( \# \) before the match of \( r \) is the last element of \( \beta \). Now this conflicts with \( \beta \in \text{re}luc(r_{\#}) \), 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 \( \kappa_{r-\omega} = \zeta \) then \( (\kappa,\zeta) \in L(M_{r-\omega}) \).

Similarly, we concentrate on the case where there is at least one match of \( r \) in \( \kappa \). Notice that there exists one and only one \( r \)-begin-marked output of \( \kappa \), and let it be \( \theta \). According to Lemma 5.13, \( (\kappa,\theta) \in L(A_3(r)) \). Thus, to show \( (\kappa,\zeta) \in L(M_{r-\omega}) \) we only need to prove \( (\theta,\zeta) \in L(A_4(r,\omega)) \). This is accomplished via induction on the number of replacements performed on \( \kappa \).
We now try to construct a run for \((\theta, \zeta)\) on \(A_4(r, \omega)\). In the following, we study the relations between \(\kappa, \theta, \) and \(\zeta\). According to Definition 3.2, \(\kappa, \theta, \) and \(\zeta\) can be written as \(\kappa = \nu\beta\mu\) and \(\zeta = \nu\omega\eta\) that satisfy the conditions which enforce \(\beta\) to be the earliest and relevant 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 \(\eta = \mu\gamma \). We could write \(\theta\) as the following form:

\[
\theta = \nu\#\beta\mu'
\]

Such a form is possible – we need to show that the first \# of \(\theta\) is right after \(\nu\). This is based on the following facts: (1) \(\theta\) is the \(r\)-begin-marked output of \(\kappa\), (2) \(\nu \in \Sigma^* - \Sigma^* r \Sigma^*\), and (3) \(\beta\) is the earliest match of \(r\) in \(\kappa\). Now we require that \(\beta'\) is the shortest substring of \(\theta\) starting at \# s.t. \(\beta = \pi_2(\beta')\). Due to this requirement, \(\beta' \in \text{reluc}(r)\).

We now construct the first portion of the acceptance run. Given \(\nu \in \Sigma^* - \Sigma^* r \Sigma^*\), \(\beta \in \text{reluc}(r)\), we have the following partial run for \((\nu\#\beta', \nu\omega)\) in \(A_4(r, \omega)\):

\[
\gamma_1 : f_1^4 \sim_{(\nu, \nu)} f_1^4 \sim_{(\#, r)} s_1^4 \sim_{(\beta', \omega)} s_2^4 \sim_{(r, r)} f_1^4
\]

Then consider \(\mu'\) and \(\eta\). Using an argument similar to (4) of Direction \(\Rightarrow\), we can show that \(\mu'\) is the \(r\)-begin-marked output of \(\mu\), thus \((\mu', \eta) \in L(A_3(r))\). Now by Definition 3.2, \(\eta = \mu\gamma \), and this leads to \((\mu, \eta) \in \mathcal{M}_{\kappa}^\rightarrow\) by the induction assumption. Therefore, we have \((\mu', \eta) \in A_4(r, \omega)\) (using Definition 4.2). This leads to the following run in \(A_4(r, \omega)\):

\[
\gamma_2 : f_1^4 \sim_{(\mu', \eta)} f_1^4
\]

It follows that \(\gamma_1 \circ \gamma_2\) is the acceptance run for \((\theta, \zeta)\) on \(A_4(r, \omega)\). This concludes the proof for \((\kappa, \zeta) \in L(\mathcal{M}_{\kappa}^\rightarrow)\).

\[\Box\]

### B.6. Proof of Lemma 5.38

**Lemma 5.38** For any \(r \in R\), any \(\kappa \in \Sigma^*\) and \(\eta \in \Sigma_2\), \(\eta\) is the \(r\)-greedy-marking of \(\kappa\) iff \((\kappa, \eta) \in L(\mathcal{M}_{\kappa}^+\)\).

Proof: (Direction \(\Rightarrow\)) we first show that if \((\kappa, \eta) \in L(\mathcal{M}_{\kappa}^+)\), this results from the following facts: (1) \((\kappa, \eta) \in L(A_3(r))\) because \(A_3(r)\) optionally inserts a $ after each match of \(r\) (so \(\eta\) is one of the output words). (2) \((\eta, \eta)\) is accepted by all of the \(A_6, A_7(r), A_8(r), A_9(r)\) filters. This is because \(\eta\) is already properly marked by the assumption.

(Direction \(\Leftarrow\)) The proof goal is if \((\kappa, \eta) \in L(\mathcal{M}_{\kappa}^+)\), then \(\eta\) is the \(r\)-greedy-marking of \(\kappa\). As there exists one and only one \(r\)-greedy marking for any word, and it is the output word of \(\mathcal{M}_{\kappa}^+\) (see direction \(\Rightarrow\)). We only need to prove the following proposition:

1. **(J1)** If both \((\kappa, \eta)\) and \((\kappa, \eta')\) are accepted by \(\mathcal{M}_{\kappa}^+\), then \(\eta = \eta'\).

Since \(\mathcal{M}_{\kappa}^+\) is a composition of a sequence of transducers, and each transducer can be regarded as an input/output device. We write the process of reaching \(\eta\) and \(\eta'\) as following, using subscript to indicate the output of a particular transducer.

\[
\begin{align*}
\kappa &\rightarrow A_3(r) \rightarrow A_5(r) \rightarrow A_6 \rightarrow A_7(r) \rightarrow A_8(r) \rightarrow A_9(r) \\
\kappa &\rightarrow A_3(r) \rightarrow A_5(r) \rightarrow A_6 \rightarrow A_7(r) \rightarrow A_8(r) \rightarrow A_9(r)
\end{align*}
\]

Based on the construction algorithm of all component transducers, one can infer that \(\eta_3 = \eta'_3\), and \(\eta_5 = \eta_7 = \eta_8 = \eta\). Therefore, we prove J1 by contradiction. Assume that \(\eta \neq \eta'\). Let index \(i\) be the first index that \(\eta\) differs from \(\eta'\), i.e., \(\eta[i] \neq \eta'[i]\) and \(\forall i, 0 \leq x < i : \eta[x] = \eta'[x]\). Then we have the following seven cases to discuss (note that the length of \(\eta\) may be different from that of \(\eta'\)):

- **Case 1** \((e, a)\): \(|\eta| = i\) and \(\eta'[i] \in \Sigma\); or symmetrically \(|\eta'| = i\) and \(\eta[i] \in \Sigma\).
- **Case 2** \((e, \#)\): \(|\eta| = i\) and \(\eta'[i] = \#\); or symmetrically \(|\eta'| = i\) and \(\eta[i] = \#\).
- **Case 3** \((e, \$)\): \(|\eta| = i\) 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 can hold via contradiction.

Cases 1 and 4: We can first discharge these two cases using Lemma 5.37. Take Case 4 $(a, b)$ as an example: we have $\forall x \leq i: \eta[x] = \eta'[x]$, 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 5.37, which requires that both of $\pi_\Sigma(\eta)$ and $\pi_\Sigma(\eta')$ should be equal to $\kappa$.

Cases 3 and 7: As an example, we prove that Case 3 $(\#, \#)$ is impossible. The assumption is that $|\eta| = i$ and $\eta'[i] = \#$. According to Lemma 5.24, $A_6$ ensures that $\#$ and $\#$ markers are paired in $\eta$. This leads to the contradiction that the $\#$ at index $i$ of $\eta'$ does not have a corresponding $\#$ in $\eta'$. Case 7 can be proved using the same technique.

Cases 6: We show that Case 6 $(a, \#)$ is impossible. The idea of the proof is presented in Figure 19.

The assumption is that $\eta[i] = a$ and $\eta'[i] = \#$, and $\forall x \leq i: \eta[x] = \eta'[x]$. By Lemma 5.24, in $\eta'$ 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 $\eta$ we also have $\eta[j] = \#$ and $\eta[j + 1, i] \in \Sigma^*$. Now study $\eta$, 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 $\eta'$: by Lemma 5.35, there must exist $k' > j$ 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 $\eta'$ is improperly marked (for a shorter match), and this conflicts with the fact that $\eta'$ passes longest match filter $A_9(r)$. Thus case 6 could not hold.

Case 2: The assumption is that $|\eta| = i$ and $\eta'[i] = \#$, and $\eta[0, i] = \eta'[0, i]$. We show that case 2 cannot hold. The idea of the proof is presented in Figure 19.

We first prove the following proposition given the assumption.

(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) In $\eta$, every $\#$ precedes a match of $r$ (ensured by $A_3(r)$). Then we have $\eta'[i + 1] = \#$. In the following, we trace ("taint") the data processing that generates the last two symbols in $\eta'$, and let them be $\#_i$ and $\$_{i+1}$, respectively (see Figure 19). Observe all the transducers in $M^*: A_3(r)$ introduces $\#$, $A_5(r)$ introduces $\$, and $A_6$ could remove $\#$ as well as $\$. We know that $\eta_3 = \eta_4$, and $\eta_6 = \eta$, and $\eta_6' = \eta'$. From the above facts, we can infer that $\#_i$ appears in both $\eta_3$ and $\eta_4$. However, after $\eta_5$ is fed to $A_6$, $\#_i$ is removed from the output word (i.e., $\eta_6$) using transition $(q_1^6, q_1^6, \#, \epsilon)$. Notice that $q_1^6$ is not a final state. To come back to final state $q_0^6$, a transition $(q_1^6, q_0^6, \$, \$) has to be taken. The only $\$ now left on the input word (i.e., $\eta_6$) is the $\$_{i+1}$, and it is guaranteed to be the last input symbol on $\eta_6$ (because $A_5$ does not insert consecutive $\$ signs). Thus we can infer that $\$_{i+1}$ is also the last symbol of $\eta_6 = \eta$. Notice that the last element of $\eta$ is located at index $i - 1$. Therefore, we have the following:

(K2) $\eta[i - 1] = \eta'[i - 1] = \$.

Combining (K1) and (K2), one can infer that $\eta'$ can be written as the following form, where $\alpha \in \Sigma^*_2$:

$\eta' = \alpha\$_{i-1}\#_{i}\$_{i+1}$

Now since the $\$ before $\#$ (i.e., $\$_{i-1}$) has a paired $\#$ in $\eta'$, we can infer that there is at least one element
before it. However, \( \eta'[i-2] \) cannot be \#, because \( \eta'[i-2] = \# \) leads to the conclusion that \( \eta'[i-2] \) and \( \eta'[i] \), when traced back to \( \eta' \), are consecutive \# signs generated by \( A_3(r) \). This contradicts with Definition 5.12. Similarly, \( \eta'[i-2] \) cannot be $$. Now the only choice for \( \eta'[i-2] \) is a symbol in \( \Sigma \) (let it be \( a \)). Up to now we can write \( \eta' \) as the following, where \( \beta \in \Sigma_2^* \):

\[
\eta' = \beta a \# b \#
\]

Therefore \( \eta' \) is contained in \( \Sigma_2^*[\#](\$ \cap r_{\#, b}) \). This conflicts with the filter \( L_2(2) \) defined in \( A_6(r) \). In conclusion, Case 2 cannot be true.

**Case 5:** We prove that Case 5 (\( b, \# \)) cannot 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 \( A_3(r) \) does not generate consecutive \# signs.

- **(ii):** Now we reject case (ii) by contradiction. The proof idea is shown in Figure 19.

  Let \( \eta[i-1] = a \) where \( a \in \Sigma \). Based on the previous discussion, \( \eta \) and \( \eta' \) can be written as the following, where \( \alpha, \beta, \beta' \in \Sigma_2^* \) (see Figure 19):

  \[
  \eta = \alpha a b \beta, \eta' = \alpha a \# b'
  \]

  Then we trace the source of the characters \( a, b, \# \) in the above formula. Note that there might be multiple \( a \)'s, \( b \)'s, and \( \# \)'s in \( \eta \) and \( \eta' \). In our following discussion, we use \( a, b, \# \) to specifically refer to \( \eta'[i-1] \) (i.e., \( \alpha \)), \( \eta[i] \) (i.e., \( \beta \)), and \( \eta'[i] \) (i.e., \( \# \)). We use indices \( a_i, b_i, \#_i \) to indicate the index of the source of \( a, b, \# \) in an intermediate output \( \eta_i \). For example, \( \eta_3[a_3] \) is the \( a \) in \( \eta_3 \), and \( \#_3 \) indicates the index of the \# character in \( \eta_3 \).

  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 \( a_5' < \#_5' < b_5' \). Note that there does not exist \( \#_6 \), because it is consumed by \( A_6 \).

  Now consider \( \eta_5[a_5+1, b_5] \) (the substring of \( \eta_5 \) 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 \( \Sigma \) (otherwise, \( a \) and \( b \) will not be neighbors in \( \eta \)). 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 \( A_3(r) \) 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: \#,\$\#,\$\$\$.

  Recall that \( \eta_5[\#_5] \) disappears from \( \eta_6 \) by traversing the \( (q_0^5, q_0^6, \#, : e) \) in \( A_6 \). \$ cannot immediately precede \#, because in \( A_4 \) none of the transitions with \$ on the input tape reaches state \( q_0^6 \). Similarly, we refute the case of \$ right after \# in \( \eta_5[a_5+1, b_5] \), because the \$ will travel on \( (q_0^6, q_0^6, \$: \$) \) and appear between \( a \) and \( b \) in \( \eta \). 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 \( \eta_6 \).

  Let us observe the run of \( \eta_6 \) on \( A_6 \). All of the \( a, \#, b \) in \( \eta_5[a_5, b_5+1] \) are taking self-loop transitions on \( q_0^5 \). There must be one \# before \( a \) for entering \( q_0^6 \), and one \$ after \( b \) for entering final state \( q_0^6 \).

  Let the \# and \$ be the element \( x \) and \( y \) in \( \eta \) (see Figure 19). It follows that \( x < i - 1 \) and \( y > i \), and \( \eta[x+1, y] \in \Sigma^* \), which implies \( \eta'[x, i] \in \#^* \). Now for \( \eta' \), there are two \# signs (at \( x \) and \( i \)), between which there is no \$ sign. This conflicts with Lemma 5.24 that markers should be paired.

  - **(iii):** We finally refute the case that \( \eta[i-1] = \$ \). In this case let \( \eta = \alpha \$ b \beta \) and \( \eta' = \alpha \$ \# b' \). Consider \( \eta[i-2] \) again, it cannot be \#, otherwise \( \eta'[i-2] = \# \) and \( \eta'[i-1] = \# \) are consecutive \# signs and conflict with properties of \( A_3(r) \). We can also infer \( \pi_{\Sigma}(b \beta) = \pi_{\Sigma}(\beta') \in r \Sigma^* \), based on \( \eta'[i] = \# \). Thus we have \( \eta \cap L_2(1) = \emptyset \), given that \( L_2(1) \) is defined as \( \Sigma_2^*[\#](\$ \Sigma \Sigma_2^* \cap r_{\#, b} \Sigma_2^*) \). Therefore, \( \eta \) cannot be an output word of \( A_6(r) \).

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