Solving String Constraints with Regex-Dependent Functions through Transducers with Priorities and Variables

Taolue Chen

Birkbeck, University of London, UK

Alejandro Flores-Lamas, Matthew Hague

Royal Holloway, University of London, UK

Zhilei Han

Tsinghua University, China

Denghang Hu, Zhilin Wu

Institute of Software, CAS & UCAS, China

Shuanglong Kan, Anthony W.Lin

University of Kaiserslautern, Germany

Philipp Rümmer

Uppsala University, Sweden

POPL 2022

Table of Content

- **Background**
- Real-world Regular Expression and PSST
- The String Logic and Decision Procedure
- Implementation

Background

- The **string** type is ubiquitous in practical programs.
- Abundant operations for manipulating strings are provided
  - `replace`, `extract`, `match` . . .
  - `split`, `join`, `indexof` . . .

Background

- The **string** type is ubiquitous in practical programs.
- Sadly, strings are vulnerable to attacks[1].

Injection

```
String query = "SELECT * FROM accounts WHERE custID='"
        + request.getParameter("id") + "'";
```

Cross-Site Scripting (XSS)

```
String page += "<input name='creditcard' type='TEXT' value='"
        + request.getParameter("CC") + "'>";
```

Insecure Deserialization

LOG4J

1. https://owasp.org/www-project-top-ten/2017/Top_10

Motivation

Q1: How to analyze and verify string-manipulating programs?

Motivation

Q1: How to analyze and verify string-manipulating programs?

Constraint-based verification

```
// XSS vulnerable
function instantiate(info) {
  var template =
  "<h1>User<span onMouseOver="popupText('{{bio}}')">{{userName}}</span></h1>"
  var result = template.replace("{{bio}}", info.bio);
  result = template.replace("{{userName}}", info.username);
  return result;
}
```

$\Rightarrow x_1 = \mathrm{replaceAll}(\mathrm{temp}, \text{“\{\{bio\}\}”}, \mathrm{bio}) \wedge x_2 = \mathrm{replaceAll}(x_1, \text{“\{\{userName\}\}”}, \mathrm{user}) \wedge x_2 \in R$

Motivation

Q2: Are existing string theories/solvers sufficient for verifying practical programs?

Motivation

Q2: Are existing string theories/solvers sufficient for verifying practical programs?

`No.`

The regular expressions in real programming languages (***regex***) have more features than *classical* regular expressions.

Motivation

Q2: Are existing string theories/solvers sufficient for verifying practical programs?

`No.`

The regular expressions in real programming languages (***regex***) have more features than *classical* regular expressions.

- greedy/lazy matching: `a*` versus `a*?`

	matched by	result
“<script>foo</script>”	“<(.*)>”	“script>foo</script”
	“<(.*?)>”	“script”

Motivation

Q2: Are existing string theories/solvers sufficient for verifying practical programs?

`No.`

The regular expressions in real programming languages (***regex***) have more features than *classical* regular expressions.

- greedy/lazy matching: `a*` versus `a*?`
- capturing groups and references:

```
var t = replace(s, /((ab*?)+)/g, $2);
```

Motivation

Q2: Are existing string theories/solvers sufficient for verifying practical programs?

`No.`

The regular expressions in real programming languages (***regex***) have more features than *classical* regular expressions.

- greedy/lazy matching: `a*` versus `a*?`
- capturing groups and references:

```
var t = replace(s, /((ab*?)+)/g, $2);
```

- anchors:

```
s.match(/^a+(b*)c+$/);
```

Nonstandard Semantics

Example. (Nested Repetition)

In Javascript, some operators' behaviour depends on its context. For example, the following statement:

```
var result = "aaa".match(/(a*)*/)[1]
```

returns "aaa", while

```
var result = "aaa".match(/(a*?)*/)[1]
```

returns "a".[2]

2. The ECMAScript standard prohibits the match of e in e^* to be ε. https://262.ecma-international.org/12.0/#sec-runtime-semantics-repeatmatcher-abstract-operation

Table of Content

- Background
- **Real-world Regular Expression and PSST**
- The String Logic and Decision Procedure
- Implementation

Real-world Regular Expression

Definition 1. (Real-world Regular Expression, regex)

A real-world regular expression is defined as:

$$
\begin{aligned}
e \xlongequal{\text{def}} \; & \emptyset \mid \varepsilon \mid a \mid [e+e] \mid [e\cdot e] \mid \\
& (e) \mid && \texttt{Capturing Group} \\
& [e^{?}] \mid [e^{??}] \mid && \texttt{Optional} \\
& [e^{*}] \mid [e^{*?}] \mid && \texttt{Kleene Star} \\
& [e^{+}] \mid [e^{+?}] \mid && \texttt{Kleene Plus} \\
& [e^{\{m_1,m_2\}}] \mid [e^{\{m_1,m_2\}?}] && \texttt{Repetition}
\end{aligned}
$$

where a is a letter in alphabet Σ, $m_1, m_2 \in \mathbb{N}$ with $m_1 \leqslant m_2$.

It's hard to give a denotational semantics to regex.

Operational semantics?

Construction of PSST

We construct a *Prioritized Streaming String Transducers (PSST)* $\mathcal{T}_e$ for each regex e inductively as its operational semantics.

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

- Q *is a finite set of states,* $q_0 \in Q$ *is the initial state*
- Σ *is the input and output alphabet*
- X *is a finite set of string variables*
- $\delta \in Q \times \Sigma \to \bar{Q}$ *defines the non-*ε *transitions as well as their priorities (from highest to lowest)*
- $\tau \in Q \to \bar{Q} \times \bar{Q}$ *such that for every* $q \in Q$, *if* $\tau(q) = (P_1; P_2)$, *then* $P_1 \cap P_2 = \emptyset$,
- E *associates with each transition a string-variable assignment function, i.e.,* E *is partial function from* $Q \times \Sigma^\varepsilon \times Q$ *to* $X \to (X \cup \Sigma)^*$ *such that its domain is the set of tuples* (q, a, q') *satisfying that either* $a \in \Sigma$ *and* $q' \in \delta(q, a)$ *or* $a = \varepsilon$ *and* $q' \in \tau(q)$
- F *is the output function, which is a partial function from* Q *to* $(X \cup \Sigma)^*$

PSST

PSST extends finite state transducer with:

- Priorities: nondeterministic transitions are ordered
- Memory: a fixed number of *string variables* containing unbounded string.

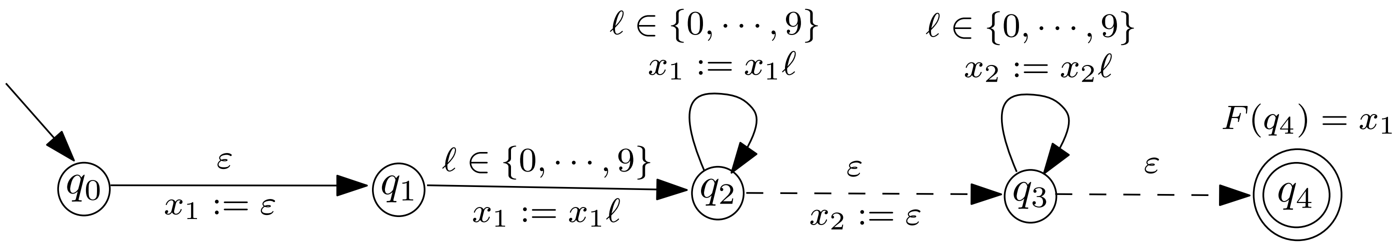


Figure 1. PSST $\mathcal{T}$ to extract the matching of the first capturing group in (\d+)(\d*)

The accepted run of $\mathcal{T}$ on the input string 2022 is:

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

...with output 2022.

Construction of PSST

We construct a PSST $\mathcal{T}_e$ for each regex e inductively as its operational semantics.

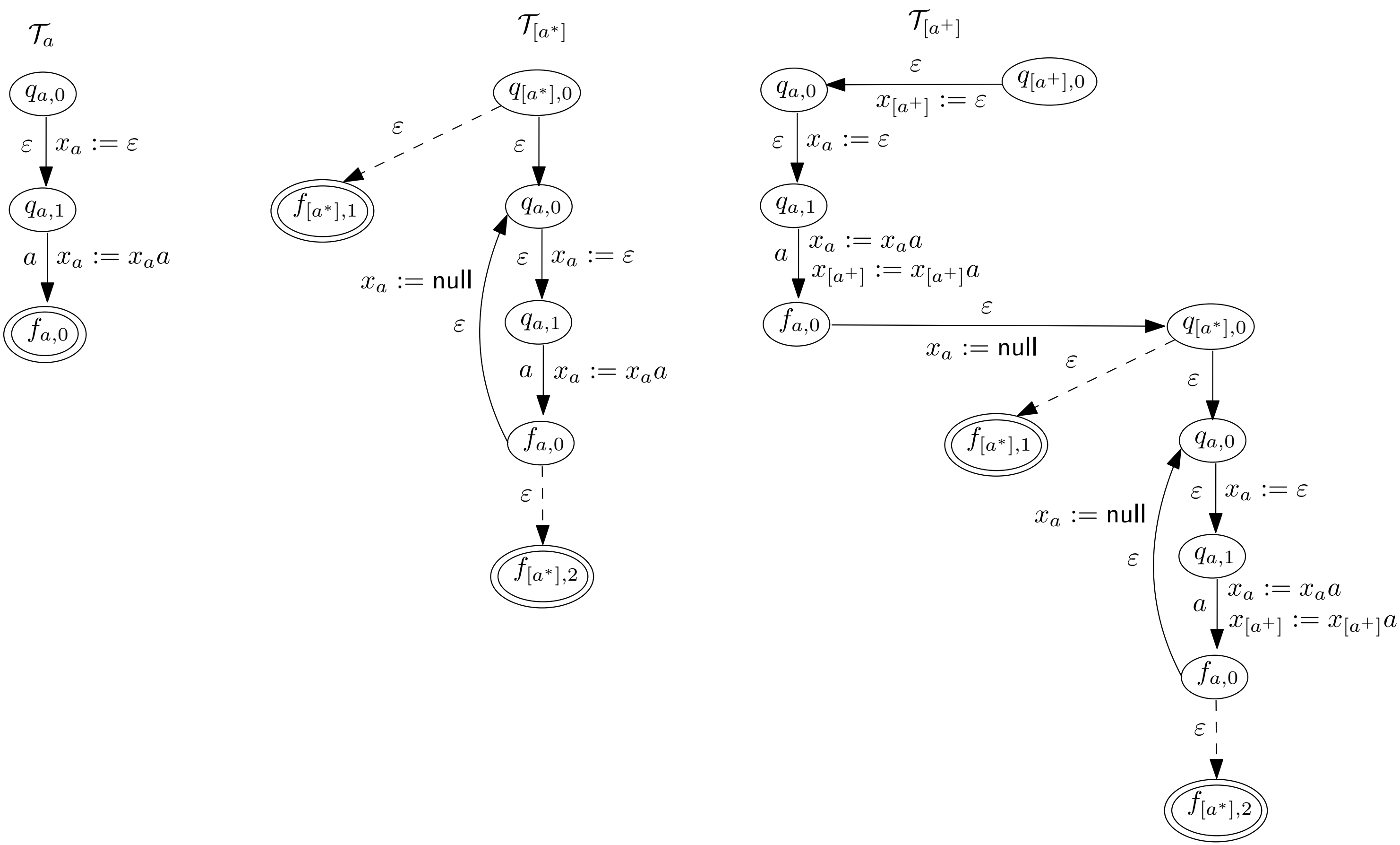


Figure 2. PSST for the regex a, a^* and a^+

Validation

We conduct experiments to validate the formal semantics against the actual JavaScript regex-string matching semantics.

For each regex e we construct $\mathcal{T}_e$ and generate an input string w and the corresponding output string w'. We execute the following code in Node.js:

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

And compare the output against w'.

1110 nontrivial regexes, including all operators, are tested. We confirm consistency of semantics on all of them.

Table of Content

- Background
- Real-world Regular Expression and PSST
- **The String Logic and Decision Procedure**
- Implementation

Our String Logic

Definition 2. (STR) *The well-formed formula of the theory STR is defined as:*

$$
\begin{aligned}
\varphi \stackrel{\text{def}}{=\!=} \;& x = y \\
& \mid z = x \cdot y && \text{concatenation} \\
& \mid x \in e && \text{regular constraint} \\
& \mid y = \mathsf{extract}_{i,e}(x) && \text{extraction} \\
& \mid y = \mathsf{replaceAll}_{\mathrm{pat},\mathrm{rep}}(x) && \text{replacement} \\
& \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi
\end{aligned}
$$

where x, y, z *are string variables,* $i \in \mathbb{N}$ *is the index of capturing groups,* e, $\mathrm{pat} \in \mathrm{regex}$ *is the match pattern, and* $\mathrm{rep} \in \Re$ *is the replacement string.* $\Re$ *is defined as the concatenation of letters in* Σ *and references* $\$i$, *for* $i \in \mathbb{N}$.

Theorem 1. *For each constraint* $y = \mathsf{extract}_{i,\mathrm{pat}}(x)$ *and* $y = \mathsf{replaceAll}_{\mathrm{pat},\mathrm{rep}}(x)$. *an equivalent PSST* $\mathcal{T}$ *can be constructed. (Lemma 4.7)*

Decidability

Theorem 2. *STR is undecidable.*

Proof. Followed directly from[3] □

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

Sequent Calculus for STR

we handle STR constraints with a sound sequent calculus.

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

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

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

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

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

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

Sequent Calculus for STR

we handle STR constraints with a sound sequent calculus.

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

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

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

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

- It is shown in previous work[4] that the preimage of concatenaction, $\cdot^{-1}(L(e))$, is computable.
- Is the preimage of a PSST computable?

4. T. Chen, Y. Chen, M. Hague, A. W. Lin, and Z. Wu, 'What is decidable about string constraints with the ReplaceAll function', *PACMPL*, vol. 2, no. POPL, p. 3:1-3:29, 2018

Regularity-Preserving Property

Theorem 3.

Given a PSST $\mathcal{T}$ and an FA A, we can compute an FA B in exponential time such that $B = \mathcal{T}^{-1}(A)$. (Lemma 5.5)

Proof. By simulation. Available in full version of the paper[5]. □

5. https://arxiv.org/pdf/2111.04298.pdf

Decidability

Theorem 4. *STR is undecidable.*

Proof. Followed directly from[6] □

But. . .

Theorem 5. *The straight-line fragment of STR is decidable.*

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

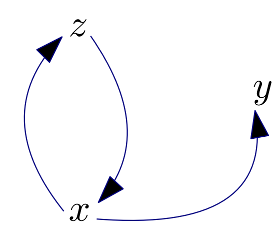
Straight-line Fragment

Definition 3. *A STR formula φ is said to be **straight-line**, if*

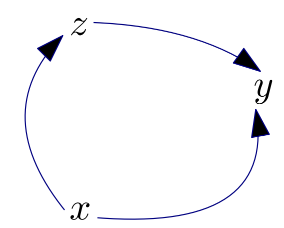
1. *it contains neither negation nor disjunction*
2. *φ can be ordered into a sequence of equations $x_1 = t_1, x_2 = t_2, \ldots, x_n = t_n$ plus regular constraints, such that $x_1, \ldots, x_n$ are mutually distinct, and for each $i \in \{1, \ldots, n\}$, x_i does not occur in $t_1, \ldots, t_{i-1}$.*

Let $\mathrm{STR}_{\mathrm{SL}}$ denote the set of straight-line STR formulas.

$z = \mathsf{replaceAll}_{(a+b)^*, \$1}(x) \wedge x = y \cdot z \wedge y \in ab^*$ $\times$



$z = \mathsf{replaceAll}_{(a+b)^*, \$1}(y) \wedge x = y \cdot z \wedge y \in ab^*$ $\checkmark$



Decision Procedure for $\mathrm{STR}_{\mathrm{SL}}$

Every $\mathrm{STR}_{\mathrm{SL}}$ formula φ can be simplified into conjunctions of formulas of the form $z = x \cdot y$, $y = \mathcal{T}(x)$ and $x \in A$, where $\mathcal{T}$ is a PSST and A is an FA.

Example. The following constraint:

$$\begin{array}{rcl} x \in A_x & \wedge & \\ y = \mathcal{T}(x) & \wedge & y \in A_y \\ z = x \cdot y & \wedge & z \in A_z \end{array}$$

is straight-line. We decide its satisfiability by iteratively computing **pre-images** of regular constraints under $\cdot$ (concatenation) and $\mathcal{T}$.

Step 1. For $z = x \cdot y$ and $z \in A_z$, by previous results, we can compute $\cdot^{-1}(A_z) = \{(x, y) | \, x \cdot y \in A_z\} = \bigcup_1^n A'_{i,x} \times A'_{i,y}$ for some $A'_{i,x}$ and $A'_{i,y}$.

Decision Procedure for $\mathrm{STR}_{\mathrm{SL}}$

Example. The following constraint:

$$\begin{aligned} x \in A_x &\quad \wedge \\ y = \mathcal{T}(x) &\quad \wedge \quad y \in A_y \\ z = x \cdot y &\quad \wedge \quad z \in A_z \end{aligned}$$

is straight-line. We decide its satisfiability by iteratively computing **pre-images** of regular constraints under $\cdot$ (concatenation) and $\mathcal{T}$.

$$\begin{aligned} x \in A_x \cap A'_{i,x} &\quad \wedge \\ y = \mathcal{T}(x) &\quad \wedge \quad y \in A_y \cap A'_{i,y} \\ z = x \cdot y &\quad \wedge \quad z \in A_z \end{aligned}$$

Decision Procedure for $\mathrm{STR}_{\mathrm{SL}}$

Example. The remaining constraint:

$$\begin{aligned} & x \in A_x \cap A'_x \ \wedge \\ & \quad y = \mathcal{T}(x) \ \wedge \ y \in A_y \cap A'_y \end{aligned}$$

Step 2. For $y = \mathcal{T}(x)$ and $y \in A_y \cap A'_y$, we compute $A''_x = \mathrm{Pre}(\mathcal{T}, A_y \cap A'_y) = \mathcal{T}^{-1}(A_y \cap A'_y) = \{x | \mathcal{T}(x) \in A_y \cap A'_y\}$.

Decision Procedure for $\mathrm{STR}_{\mathrm{SL}}$

Example. The remaining constraint:

$$x \in A_x \cap A'_x \cap A''_x$$

~~$y = T(x) \quad \wedge \quad y \in A_y \cap A'_y$~~

Step 3. We check the emptiness of $A_x \cap A'_x \cap A''_x$. If the language is empty, the constraint is unsatisfiable. Otherwise, it's satisfiable.

Our String Solver

OSTRICH: Optimistic STRIng Constraint Handler[7]

Version 1.1 now supports solving constraints with real-world regular expressions.

- The first and yet the only solver with such support
- The implementation supports more features like anchors.

7. https://github.com/uuverifiers/ostrich

Evaluation

RQ: How does OSTRICH compare to other solvers that can handle real-world regular expression?

We evaluate OSTRICH on over 195 000 string constraints.

It greatly increase precision and efficiency (18x) compared to previous approximation-based methods.

Average Time	OSTRICH	ExpoSE+Z3
match (98,117 constraints)	1.57s	28.0s
replace (98,117 constraints)	6.62s	55.0s

Takeaway

In our work, we propose:

1. The first string theory and solver supporting *regex* and regex-dependent string-manipulating functions.
2. A new automata model called Prioritized Streaming String Transducer (PSST) to precisely capture the semantics of real-world regular expressions.
3. The proof of regularity-preserving property of PSST.
4. A sound sequent calculus for solving the string theory, which is complete for straight-line fragment.