Statically Typed String Sanitation Inside a Python
(Technical Report)

Nathan Fulton       Cyrus Omar       Jonathan Aldrich
December 2014
CMU-ISR-14-112

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract
This report contains supporting evidence for claims put forth and explained in the paper “Statically Typed String Sanitation Inside a Python” [1], including proofs of lemmas and theorems asserted in the paper, examples, additional discussion of the paper’s technical content, and errata.

This work was supported by the National Security Agency lablet contract #H98230-14-C-0140.
Keywords: type systems; regular languages; input sanitation; string sanitation
Contents

1 Terminology and Notation 2

2 Regular Expressions 2

3 $\lambda_{RS}$ 2
   3.1 Static Semantics 2
      3.1.1 Case Analysis 2
      3.1.2 Replacement 3
   3.2 Dynamic Semantics 3
      3.2.1 Canonical Forms 3
      3.2.2 Type Safety 3
      3.2.3 The Security Theorem 6

4 $\lambda_P$ 7
   4.1 Static Semantics 7
   4.2 Dynamic Semantics 7
      4.2.1 Canonical Forms 7
      4.2.2 Type Safety 7

5 Translation from $\lambda_{RS}$ to $\lambda_P$ 10

List of Figures

1 Syntax of Regular Expressions 17
2 Syntax of $\lambda_{RS}$ 17
3 Syntax of $\lambda_P$ 17
4 Static Semantics of $\lambda_{RS}$ 17
5 Dynamic Semantics of $\lambda_{RS}$ 18
6 Static Semantics of $\lambda_P$ 19
7 Dynamic Semantics of $\lambda_P$ 20
8 Translation from $\lambda_{RS}$ to $\lambda_P$ 21
1 Terminology and Notation

Theorems and lemmas appearing in [1] are numbered correspondingly, while supporting facts appearing only in the Technical Report are lettered. Throughout this technical report, we use a small step semantics corresponding to the big step semantics given in [1].

2 Regular Expressions

The syntax of regular expressions over some alphabet $\Sigma$ is shown in Figure 1.

**Assumption A** (Regular Expression Congruences). We assume regular expressions are implicitly identified up to the following congruences:

- $\epsilon \cdot r \equiv r$
- $r \cdot \epsilon \equiv r$
- $(r_1 \cdot r_2) \cdot r_3 \equiv r_1 \cdot (r_2 \cdot r_3)$
- $r_1 + r_2 \equiv r_2 + r_1$
- $(r_1 + r_2) + r_3 \equiv r_1 + (r_2 + r_3)$
- $\epsilon^* \equiv \epsilon$

**Assumption B** (Properties of Regular Languages). We assume the following properties:

1. If $s_1 \in \mathcal{L}\{r_1\}$ and $s_2 \in \mathcal{L}\{r_2\}$ then $s_1s_2 \in \mathcal{L}\{r_1 \cdot r_2\}$.
2. For all strings $s$ and regular expressions $r$, either $s \in \mathcal{L}\{r\}$ or $s \notin \mathcal{L}\{r\}$.
3. Regular languages are closed under reversal.

3 $\lambda_{RS}$

The syntax of $\lambda_{RS}$ is specified in Figure 2.

3.1 Static Semantics

The static semantics of $\lambda_{RS}$ is specified in Figure 3. The typing context obeys the standard structural properties of weakening, exchange and contraction.

3.1.1 Case Analysis

The following correctness conditions must hold for any definition of $lhead(r)$ and $ltail(r)$.

**Condition C** (Correctness of Head). If $c_1s' \in \mathcal{L}\{r\}$, then $c_1 \in \mathcal{L}\{lhead(r)\}$.

**Condition D** (Correctness of Tail). If $c_1s' \in \mathcal{L}\{r\}$ then $s' \in \mathcal{L}\{ltail(r)\}$.

For example, we conjecture (but do not here prove) that the definitions below satisfy these conditions. Note that these are slightly amended relative to the published paper.
Definition 1 (Definition of lhead(r)). We first define an auxiliary relation that determines the set of characters that the head might be, tracking the remainder of any sequences that appear:

\[ \text{lhead}(\epsilon, \epsilon) = \emptyset \]
\[ \text{lhead}(\epsilon, r') = \text{lhead}(r', \epsilon) \]
\[ \text{lhead}(a, r') = \{a\} \]
\[ \text{lhead}(r_1 \cdot r_2, r') = \text{lhead}(r_1, r_2 \cdot r') \]
\[ \text{lhead}(r_1 + r_2, r') = \text{lhead}(r_1, r') \cup \text{lhead}(r_2, r') \]
\[ \text{lhead}(r^*, r') = \text{lhead}(r, \epsilon) \cup \text{lhead}(r', \epsilon) \]

We define \( \text{lhead}(r) = a_1 + a_2 + \ldots + a_i \) iff \( \text{lhead}(r, \epsilon) = \{a_1, a_2, \ldots, a_i\} \).

Definition 2 (Brzozowski’s Derivative). The derivative of \( r \) with respect to \( s \) is denoted by \( \delta_s(r) \) and is:

\[ \delta_s(r) = \{t \mid st \in L\{r\}\} \]

Definition 3 (Definition of ltail(r)). If \( \text{lhead}(r, \epsilon) = \{a_1, a_2, \ldots, a_i\} \), then we define \( \text{ltail}(r) = \delta_{a_1}(r) + \delta_{a_2}(r) + \ldots + \delta_{a_i}(r) \).

3.1.2 Replacement
The following correctness condition must hold for any definition of lreplace(r, r_1, r_2).

Condition E (Replacement Correctness). If \( s_1 \in L\{r_1\} \) and \( s_2 \in L\{r_2\} \) then:

\[ \text{replace}(r; s_1; s_2) \in L\{\text{lreplace}(r, r_1, r_2)\} \]

We do not give a particular definition for lreplace(r, r_1, r_2) here.

3.2 Dynamic Semantics
Figure 5 specifies a small-step operational semantics for \( \lambda_{RS} \).

3.2.1 Canonical Forms

Lemma F (Canonical Forms). If \( \emptyset \vdash v : \sigma \) then:

1. If \( \sigma = \text{stringin}[r] \) then \( v = \text{rstr}[s] \) and \( s \in L\{r\} \).
2. If \( \sigma = \sigma_1 \rightarrow \sigma_2 \) then \( v = \lambda x.e' \).

Proof. By inspection of the static and dynamic semantics.

3.2.2 Type Safety

Lemma G (Progress). If \( \emptyset \vdash e : \sigma \) either \( e = v \) or \( e \mapsto e' \).

Proof. The proof proceeds by rule induction on the derivation of \( \emptyset \vdash e : \sigma \).

\( \lambda \) fragment. Cases SS-T-Var, SS-T-Abs, and SS-T-App are exactly as in a proof of progress for the simply typed lambda calculus.
Lemma I. Suppose $\emptyset \vdash rstr[s] : \text{stringin}[s]$. Then $e = rstr[s]$.

S-T-Stringin-I. Suppose $\emptyset \vdash rconcat(e_1; e_2) : \text{stringin}[r_1 \cdot r_2]$ and $\emptyset \vdash e_1 : \text{stringin}[r_1]$ and $\emptyset \vdash e_2 : \text{stringin}[r_2]$. By induction, $e_1 \mapsto e_1'$ or $e_1 = v_1$ and similarly, $e_2 \mapsto e_2'$ or $e_2 = v_2$. If $e_1$ steps, then SS-E-Concat-Left applies and so rconcat($e_1; e_2) \mapsto rconcat(e_1'; e_2)$. Similarly, if $e_2$ steps then $e$ steps by SS-E-Concat-Right.

In the remaining case, $e_1 = v_1$ and $e_2 = v_2$. But then it follows by Canonical Forms that $e_1 = rstr[s_1]$ and $e_2 = rstr[s_2]$. Finally, by SS-E-Concat, rconcat($rstr[s_1]; rstr[s_2]) \mapsto rstr[s_1 s_2]$. 

S-T-Case. Suppose $e = rstrcase(e_1; e_2; x; y.e_3) \vdash \emptyset \vdash : \text{stringin}[r]$. By induction and Canonical Forms it follows that $e_1 \mapsto e_1'$ or $e_1 = rstr[s]$. In the former case, $e$ steps by S-E-Case-Left. In the latter case, note that $s = \epsilon$ or $s = at$ for some string $t$. If $s = \epsilon$ then $e$ steps by S-E-Case-$\epsilon$-Val, and if $s = at$ the $e$ steps by S-E-Case-Concat.

S-T-Replace. Suppose $e = replace[r](e_1; e_2), \emptyset \vdash : \text{stringin}[replace(r, r_1, r_2)]$ and:

(1) $\emptyset \vdash e_1 : \text{stringin}[r_1]$

(2) $\emptyset \vdash e_2 : \text{stringin}[r_2]$

By induction on (1), $e_1 \mapsto e_1'$ or $e_1 = v_1$ for some $e_1'$. If $e_1 \mapsto e_1'$ then $e$ steps by SS-E-Replace-Left. Similarly, if $e_2$ steps then $e$ steps by SS-E-Replace-Right. The only remaining case is where $e_1 = v_1$ and also $e_2 = v_2$. By Canonical Forms, $e_1 = rstr[s_1]$ and $e_2 = rstr[s_2]$. Therefore, $e \mapsto rstr[replace(r; s_1; s_2)]$ by SS-E-Replace.

S-T-SafeCoerce. Suppose that $\emptyset \vdash coercer[r](e_1) : \text{stringin}[r]$ and $\emptyset \vdash e_1 : \text{stringin}[r']$ for $L\{r'\} \subseteq L\{r\}$. By induction, $e_1 = v_1$ or $e_1 \mapsto e_1'$ for some $e_1'$. If $e_1 \mapsto e_1'$ then $e$ steps by SS-E-SafeCoerce-Step. Otherwise, $e_1 = v$ and by Canonical Forms $e_1 = rstr[s]$. In this case, $e = coercer[r](rstr[s]) \mapsto rstr[s]$ by SS-E-SafeCoerce.

S-T-Check Suppose that $\emptyset \vdash rcheck[r](e_0; x.e; e_2) : \text{stringin}[r]$ and:

(3) $\emptyset \vdash e_0 : \text{stringin}[r_0]$

(4) $\emptyset, x : \text{stringin}[r] \vdash e_1 : \sigma$

(5) $\emptyset \vdash e_2 : \sigma$

By induction, $e_0 \mapsto e_0'$ or $e_0 = v$. In the former case $e$ steps by SS-E-Check-StepLeft. Otherwise, $e_0 = rstr[s]$ by Canonical Forms. By Lemma 3 part 2, either $s \in L\{r_0\}$ or $s \notin L\{r_0\}$. In the former case $e$ takes a step by SS-E-Check-Ok. In the latter case $e$ takes a step by SS-E-Check-NotOk.

Assumption H (Substitution). If $\Psi, x : \sigma' \vdash e : \sigma$ and $\Psi \vdash e' : \sigma'$, then $\Psi \vdash [e'/x]e : \sigma$.

Lemma I (Preservation for Small Step Semantics). If $\emptyset \vdash e : \sigma$ and $e \mapsto e'$ then $\emptyset \vdash e' : \sigma$.

Proof. By induction on the derivation of $e \mapsto e'$ and $\emptyset \vdash e : \sigma$.

$\lambda$ fragment. Cases SS-E-AppLeft, SS-E-AppRight, and SS-E-AppAbs are exactly as in a proof of type safety for the simply typed lambda calculus.
**S-E-Concat-Left.** Suppose \( e = \text{rconcat}(e_1; e_2) \mapsto \text{rconcat}(e'_1; e_2) \) and \( e_1 \mapsto e'_1 \). The only rule that applies is S-T-Concat, so \( \emptyset \vdash e_1 : \text{stringin}[r_1] \) and \( \emptyset \vdash e_2 : \text{stringin}[r_2] \). By induction, \( \emptyset \vdash e'_1 : \text{stringin}[r_1] \). Therefore, by S-T-Concat, \( \emptyset \vdash \text{rconcat}(e'_1; e_2) : \text{stringin}[r_1 r_2] \).

**S-E-Concat-Right.** Suppose \( e = \text{rconcat}(e_1; e_2) \mapsto \text{rconcat}(e_1; e'_2) \) and \( e_2 \mapsto e'_2 \). The only rule that applies is S-T-Concat, so \( \emptyset \vdash e_1 : \text{stringin}[r_1] \) and \( \emptyset \vdash e_2 : \text{stringin}[r_2] \). By induction, \( \emptyset \vdash e'_2 : \text{stringin}[r_2] \). Therefore, by S-T-Concat, \( \emptyset \vdash \text{rconcat}(e_1; e'_2) : \text{stringin}[r_1 r_2] \).

**S-E-Concat.** Suppose \( \text{rconcat}(rstr[s_1]; rstr[s_2]) \mapsto rstr[s_1 s_2] \). The only applicable rule is S-T-Concat, so \( \emptyset \vdash rstr[s_1] : \text{stringin}[r_1] \) and \( \emptyset \vdash rstr[s_2] : \text{stringin}[r_2] \) and \( \emptyset \vdash \text{rconcat}(rstr[s_1]; rstr[s_2]) : \text{stringin}[r_1 \cdot r_2] \). By Canonical Forms, \( s_1 \in \mathcal{L}\{r_1\} \) and \( s_2 \in \mathcal{L}\{r_2\} \) from which it follows by Lemma [B] that \( s_1 s_2 \in \mathcal{L}\{r_1 \cdot r_2\} \). Therefore, \( \emptyset \vdash rstr[s_1 s_2] : \text{stringin}[r_1 \cdot r_2] \) by S-T-Restr.

**S-E-Case-Left.** Suppose \( e \mapsto \text{rstrcase}(e'_1; e_2; x, y, e_3) \) and \( \emptyset \vdash e : \sigma \) and \( e_1 \mapsto e'_1 \). The only rule that applies is S-T-Case, so:

\[
\begin{align*}
(6) & \quad \emptyset \vdash e_1 : \text{stringin}[r] \\
(7) & \quad \emptyset \vdash e_2 : \sigma \\
(8) & \quad \emptyset, x : \text{stringin}[\text{lhead}(r)], y : \text{stringin}[\text{ltail}(r)] \vdash e_3 : \sigma
\end{align*}
\]

By (6) and the assumption that \( e_1 \mapsto e'_1 \), it follows by induction that \( \emptyset \vdash e'_1 : \text{stringin}[r] \). This fact together with (7) and (8) implies by S-T-Case that \( \emptyset \vdash \text{rstrcase}(e'_1; e_2; x, y, e_3) : \sigma \).

**S-E-Case-ε-Val.** Suppose \( \text{rstrcase}(e_0; e_2; x, y, e_3) \mapsto e_2 \). The only rule that applies is S-T-Case, so \( \emptyset \vdash e_2 : \sigma \).

**S-E-Case-Concat.** Suppose that \( e = \text{rstrcase}(\text{rstr}[as]; e_2; x, y, e_3) \mapsto \text{rstr}[a], \text{rstr}[s]/x, y)e_3 \) and that \( \emptyset \vdash e : \sigma \). The only rule that applies is S-T-Case so:

\[
\begin{align*}
(9) & \quad \emptyset \vdash \text{rstr}[as] : \text{stringin}[r] \\
(10) & \quad \emptyset \vdash e_2 : \sigma \\
(11) & \quad \emptyset, x : \text{stringin}[\text{lhead}(r)], y : \text{stringin}[\text{ltail}(r)] \vdash e_3 : \sigma
\end{align*}
\]

We know that \( as \in \mathcal{L}\{r\} \) by Canonical Forms on (9). Therefore, \( a \in \mathcal{L}\{\text{lhead}(r)\} \) by Condition [C] and \( s \in \mathcal{L}\{\text{ltail}(r)\} \) by Condition [D].

From these facts about \( a \) and \( s \) we know by S-T-Restr that \( \emptyset \vdash \text{rstr}[a] : \text{stringin}[\text{lhead}(r)] \) and \( \emptyset \vdash \text{rstr}[s] : \text{stringin}[\text{ltail}(r)] \). It follows by Assumption [F] that \( \emptyset \vdash [\text{rstr}[a], \text{rstr}[s]/x, y)e_3 : \sigma \).

**Case S-E-Replace-Left.** Suppose that \( e = \text{rreplace}[r](e_1; e_2) \mapsto \text{rreplace}[r](e'_1; e_2) \) when \( e_1 \mapsto e'_1 \). The only rule that applies is S-T-Replace, so \( \emptyset \vdash e : \text{stringin}[\text{rreplace}(r, r_1, r_2)] \) where:

\[
\begin{align*}
\emptyset & \vdash e_1 : \text{stringin}[r_1] \\
\emptyset & \vdash e_2 : \text{stringin}[r_2]
\end{align*}
\]

By induction, \( \emptyset \vdash e'_1 : \text{stringin}[r_1] \). Therefore, \( \emptyset \vdash \text{rreplace}[r](e'_1; e_2) : \text{stringin}[\text{rreplace}(r, r_1, r_2)] \) by S-T-Replace.
Case S-E-Replace-Right. Suppose that \( e = \text{replace}[^r](e_1; e_2) \mapsto \text{replace}[^r](e'_1; e_2) \) when \( e_1 \mapsto e'_1 \).
The only rule that applies is S-T-Replace, so \( \emptyset \vdash e : \text{stringin}[\text{replace}(r, r_1, r_2)] \) where:

\[
\begin{align*}
\emptyset & \vdash e_1 : \text{stringin}[r_1] \\
\emptyset & \vdash e_2 : \text{stringin}[r_2]
\end{align*}
\]

By induction, \( \emptyset \vdash e'_1 : \text{stringin}[r_1] \). Therefore, \( \emptyset \vdash \text{replace}[^r](r_1'; r_2) : \text{stringin}[\text{replace}(r, r_1, r_2)] \) by S-T-Replace.

Case S-E-Replace.
Suppose \( e = \text{replace}[^r](rstr[s_1]; rstr[s_2]) \mapsto rstr[\text{replace}(r; s_1; s_2)] \). The only applicable rule is S-T-Replace, so

\[
\begin{align*}
\emptyset & \vdash rstr[s_1] : \text{stringin}[r_1] \\
\emptyset & \vdash rstr[s_2] : \text{stringin}[r_2]
\end{align*}
\]

By canonical forms, \( s_1 \in L\{r_1\} \) and \( s_2 \in L\{r_2\} \). Therefore,

\[
\text{replace}(r; s_1; s_2) \in L\{\text{replace}(r, r_1, r_2)\}
\]

by Condition [x]. It is finally derivable by S-T-Rstr that:
\[
\emptyset \vdash rstr[\text{replace}(r; s_1; s_2)] : \text{stringin}[\text{replace}(r, r_1, r_2)].
\]

Case S-E-SafeCoerce. Suppose that \( \text{coerce}[^r](rstr[s_1]) \mapsto rstr[s_1] \). The only applicable rule is S-T-SafeCoerce, so \( \emptyset \vdash \text{coerce}[^r](s_1) : \text{stringin}[r] \) and \( \emptyset \vdash rstr[s_1] : \text{stringin}[r'] \) and \( L\{r'\} \subset L\{r\} \).
By Canonical Forms, \( s' \in L\{r'\} \). By the definition of subset, \( s' \in L\{r\} \). Therefore, by S-T-Rstr, we have that \( \emptyset \vdash rstr[s'] : \text{stringin}[r] \).

Case S-E-Check-Ok.
Suppose \( \text{check}[r](rstr[s]; x.e_1; e_2) \mapsto [rstr[s]/x]e_1 \) and \( s \in L\{r\} \), and \( \emptyset \vdash \text{check}[r](rstr[s]; x.e_1; e_2) : \sigma \). The only rule that applies is S-T-Check, so \( \emptyset, x : \text{stringin}[r] \vdash e_1 : \sigma \).
By S-T-Rstr, we have that \( \emptyset \vdash rstr[s] : \text{stringin}[r] \). By Substitution, we have that \( \emptyset \vdash [rstr[s]/x]e_1 : \sigma \).

Case S-E-Check-NotOk.
Suppose \( \text{check}[r](rstr[s]; x.e_1; e_2) \mapsto e_2 \) and \( s \notin L\{r\} \) and \( \emptyset \vdash \text{check}[r](rstr[s]; x.e_1; e_2) : \sigma \). The only applicable rule is S-T-Check, so \( \emptyset \vdash e_2 : \sigma \).

\[\square\]

**Theorem J** (Type Safety for small step semantics.). If \( \emptyset \vdash e : \sigma \) then either \( e \ \text{val} \) or \( e \mapsto^* e' \) and \( \emptyset \vdash e' : \sigma \).

**Proof.** Follows from applying progress and preservation transitively over the multistep judgement. \[\square\]

### 3.2.3 The Security Theorem

**Theorem 4** (Correctness of Input Sanitation for \( \lambda_{RS} \)). If \( \emptyset \vdash e : \text{stringin}[r] \) and \( e \mapsto^* rstr[s] \) then \( s \in L\{r\} \).

**Proof.** By type safety, \( \emptyset \vdash rstr[s] : \text{stringin}[r] \). By canonical forms, \( s \in L\{r\} \). \[\square\]

6
We will define a translation to a language with only standard strings and regular expressions. The syntax of $\lambda_P$ is shown in Figure 3.

### 4.1 Static Semantics

The static semantics of $\lambda_P$ is shown in Figure 6. The typing context of $\lambda_P$ obeys the standard structural properties of weakening, exchange and contraction.

### 4.2 Dynamic Semantics

The dynamic semantics of $\lambda_P$ is shown in Figure 7.

#### 4.2.1 Canonical Forms

**Lemma 5 (Canonical Forms).** If $\emptyset \vdash \dot{v} : \tau$ then:

- If $\tau = \tau_1 \rightarrow \tau_2$ then $\dot{v} = \lambda x : \tau.\.v$.
- If $\tau = \text{regex}$ then $\dot{v} = r\dot{x}[r]$.
- If $\tau = \text{string}$ then $\dot{v} = \text{str}[s]$.

*Proof.* By inspection of the static and dynamic semantics. \(\square\)

#### 4.2.2 Type Safety

**Theorem 6 (Progress).** If $\emptyset \vdash \iota : \tau$ either $\iota = \dot{v}$ or $\iota \mapsto \iota'$.

*Proof.* The proof proceeds by induction on the typing assumption.

- **$\lambda$ fragment.** Cases P-T-Var, P-T-Abs, and P-T-App are exactly as in a proof of progress for the simply typed lambda calculus.

- **P-T-String.** In this case, $\iota = \text{str}[s]$, which is a value.

- **P-T-Regex.** In this case, $\iota = r\dot{x}[r]$, which is a value.

- **P-T-Concat.** In this case, we have that $\emptyset \vdash \text{pconcat}(\iota_1; \iota_2) : \text{string}$ and $\emptyset \vdash \iota_1 : \text{string}$ and $\emptyset \vdash \iota_2 : \text{string}$. By the IH, we have that either $\iota_1 \mapsto \iota'_1$ or $\iota_1 = \dot{\iota}_1$, and similarly $\iota_2 \mapsto \iota'_2$ or $\iota_2 = \dot{\iota}_2$. If $\iota_1$ steps, then we can make progress via PS-E-ConcatLeft. If $\iota_2$ steps, then we can make progress via PS-E-ConcatRight. If both are values, then by canonical forms $\iota_1 = \text{str}[s_1]$ and $\iota_2 = \text{str}[s_2]$ so we can make progress by PS-E-Concat.

- **P-T-Case.** Suppose $\emptyset \vdash \text{pstrcase}(\iota_1; \iota_2; x; y.\iota_3) : \tau$ and $\emptyset \vdash \iota_1 : \text{string}$. By induction and canonical forms, either $\iota_1 \mapsto \iota'_1$ or $\iota_1 = \text{str}[s_1]$. If $\iota_1$ steps then we can make progress by PS-E-CaseLeft. If it is a value, then by the definition of strings, either $s_1 = \epsilon$ or $s_1 = as$ for some string $s$. If $s_1$ is empty, then we can make progress by PS-E-Case-Epsilon. Otherwise, we can make progress by PS-E-Case-Cons.
The proof proceeds by rule induction on

**Preservation**

**Theorem 7** (Substitution). If \( \Theta, x : \tau' \vdash \iota : \tau \) and \( \Theta \vdash \iota' : \tau' \) then \( \Theta \vdash [\iota'/x]\iota : \tau \).

**Proof.** The proof proceeds by rule induction on \( \iota \mapsto \iota' \) and \( \emptyset \vdash \iota : \tau \).

**Case PS-E-ConcatLeft.** Suppose \( \emptyset \vdash \text{pconcat}(\iota_1; \iota_2) \) and \( \emptyset \vdash \iota_1 : \text{string} \) and \( \emptyset \vdash \iota_2 : \text{string} \). By induction, and canonical forms, either \( \iota_1 \mapsto \iota_1' \) or \( \iota_1 = \text{rx}[r] \). Similarly, \( \iota_2 \mapsto \iota_2' \) or \( \iota_2 = \text{str}[s] \). If \( \iota_1 \) steps, then we can make progress by PS-E-ReplaceLeft. If \( \iota_2 \) steps then we can make progress by PS-E-ReplaceMid. If \( \iota_3 \) steps, then we can make progress by PS-E-ReplaceRight. If all three are values, we can make progress by PS-E-Replace.

**Case PS-E-Check.** Suppose \( \emptyset \vdash \text{pcheck}(\iota_1; \iota_2; \iota_3; \iota_4) \) and \( \emptyset \vdash \iota_1 : \text{regex} \) and \( \emptyset \vdash \iota_2 : \text{string} \). By induction and canonical forms, either \( \iota_1 \mapsto \iota_1' \) or \( \iota_1 = \text{rx}[r] \). Similarly, \( \iota_2 \mapsto \iota_2' \) or \( \iota_2 = \text{str}[s] \). If \( \iota_1 \) steps, then we can make progress by PS-E-CheckLeft. If \( \iota_2 \) steps, then we can make progress by PS-E-CheckRight. If both are values, then by Assumption \( \Theta \), either \( s \in \mathcal{L}\{r\} \) or \( s \notin \mathcal{L}\{r\} \). In the former case, we can make progress by PS-E-Check-OK. In the latter case, we can make progress by PS-E-Check-NotOK.

**Assumption K (Substitution).** If \( \Theta, x : \tau' \vdash \iota : \tau \) and \( \Theta \vdash \iota' : \tau' \) then \( \Theta \vdash [\iota'/x]\iota : \tau \).
The only rule that applies is P-T-Replace, so: 
\[ \emptyset \vdash \tau \]
\[ \emptyset, x : \text{string}, y : \text{string} \vdash \tau \]
By P-T-String, we have that \( \emptyset \vdash \text{str}[a] : \text{string} \) and \( \emptyset \vdash \text{str}[s] : \text{string} \). By weakening and Substitution applied twice, we have that \( \emptyset \vdash [\text{str}[a], \text{str}[s]/x, y]_{\tau_3} : \tau \).

**Case PS-E-ReplaceLeft.** Suppose preplace(\( \tau_1; \tau_2; \tau_3 \)) \( \rightarrow \) preplace(\( \tau'_1; \tau_2; \tau_3 \)) and \( \tau_1 \mapsto \tau'_1 \). The only rule that applies is P-T-Replace, so \( \tau = \text{string} \) and:
\[ \emptyset \vdash \tau_1 : \text{regex} \]
\[ \emptyset \vdash \tau_2 : \text{string} \]
\[ \emptyset \vdash \tau_3 : \text{string} \]
By induction, \( \emptyset \vdash \tau'_1 : \text{regex} \). Therefore, by P-T-Replace \( \emptyset \vdash \text{preplace}(\tau'_1; \tau_2; \tau_3) \).

**Case PS-E-ReplaceMid.** Suppose preplace(\( \text{rx}[r]; \tau_2; \tau_3 \)) \( \rightarrow \) preplace(\( \text{rx}[r]; \tau'_2; \tau_3 \)) and \( \tau_2 \mapsto \tau'_2 \). The only rule that applies is P-T-Replace, so \( \tau = \text{string} \) and:
\[ \emptyset \vdash \text{rx}[r] : \text{regex} \]
\[ \emptyset \vdash \tau_2 : \text{string} \]
\[ \emptyset \vdash \tau_3 : \text{string} \]
By induction, \( \emptyset \vdash \tau'_2 : \text{string} \). Therefore, by P-T-Replace \( \emptyset \vdash \text{preplace}(\text{rx}[r]; \tau'_2; \tau_3) \).

**Case PS-E-ReplaceRight.** Suppose preplace(\( \text{rx}[r]; \text{str}[s]; \tau_3 \)) \( \rightarrow \) preplace(\( \text{rx}[r]; \text{str}[s]; \tau'_3 \)) and \( \tau_3 \mapsto \tau'_3 \). The only rule that applies is P-T-Replace, so \( \tau = \text{string} \) and:
\[ \emptyset \vdash \text{rx}[r] : \text{regex} \]
\[ \emptyset \vdash \text{str}[s] : \text{string} \]
\[ \emptyset \vdash \tau_3 : \text{string} \]
By induction, \( \emptyset \vdash \tau'_3 : \text{string} \). Therefore, by P-T-Replace \( \emptyset \vdash \text{preplace}(\text{rx}[r]; \text{str}[s]; \tau'_3) \).

**Case PS-E-Replace.** Suppose preplace(\( \text{rx}[r]; \text{str}[s_2]; \text{str}[s_3] \)) \( \rightarrow \) str[replace(\( \text{r}; \text{s}_2; \text{s}_3 \)). The only applicable rule is P-T-Replace, so \( \tau = \text{string} \). By P-T-String, \( \emptyset \vdash \text{str}[\text{replace}(\text{r}; \text{s}_2; \text{s}_3)] : \text{string} \).

**Case PS-E-CheckLeft.** Suppose pcheck(\( \tau_1; \tau_2; \tau_3; \tau_4 \)) \( \rightarrow \) pcheck(\( \tau'_1; \tau_2; \tau_3; \tau_4 \)) and \( \tau_1 \mapsto \tau'_1 \). The only applicable typing rule is P-T-Check, so:
\[ \emptyset \vdash \tau_1 : \text{regex} \]
\[ \emptyset \vdash \tau_2 : \text{string} \]
\[ \emptyset \vdash \tau_3 : \tau \]
\[ \emptyset \vdash \tau_4 : \tau \]
By induction, \( \emptyset \vdash \tau'_1 : \text{regex} \). Therefore, by P-T-Check \( \emptyset \vdash \text{pcheck}(\tau'_1; \tau_2; \tau_3; \tau_4) : \tau \).
Case PS-E-CheckRight. Suppose \( \text{pcheck}(rx[r]; \iota_2; \iota_3; \iota_4) \mapsto \text{pcheck}(rx[r]; \iota'_2; \iota_3; \iota_4) \) and \( \iota_2 \mapsto \iota'_2 \). The only applicable typing rule is P-T-Check, so:

\[
\begin{align*}
\emptyset & \vdash rx[r] : \text{regex} \\
\emptyset & \vdash \iota_2 : \text{string} \\
\emptyset & \vdash \iota_3 : \tau \\
\emptyset & \vdash \iota_4 : \tau
\end{align*}
\]

By induction, \( \emptyset \vdash \iota'_2 : \text{string} \). Therefore, by P-T-Check \( \emptyset \vdash \text{pcheck}(rx[r]; \iota'_2; \iota_3; \iota_4) : \tau \).

Case PS-E-Check-Ok. Suppose \( \text{pcheck}(rx[r]; \text{str}[s]; \iota_3; \iota_4) \mapsto \iota_3 \). The only applicable typing rule is P-T-Check, so \( \emptyset \vdash \iota_3 : \tau \).

Case PS-E-Check-Ok. Suppose \( \text{pcheck}(rx[r]; \text{str}[s]; \iota_3; \iota_4) \mapsto \iota_4 \). The only applicable typing rule is P-T-Check, so \( \emptyset \vdash \iota_4 : \tau \).

5 Translation from \( \lambda_{RS} \) to \( \lambda_P \)

The translation from \( \lambda_{RS} \) to \( \lambda_P \) is specified in Figure 8.

Theorem 8 (Type-Preserving Translation). If \( \Psi \vdash e : \sigma \) then \( [\Psi] \vdash [e] : [\sigma] \)

Proof. By induction on the typing relation.

Case S-T-Var. Suppose \( \Psi \vdash x : \sigma \) and \( x : \sigma \in \Psi \). We have by definition that \( x : [\sigma] \in [\Psi] \) and \( [x] = x \). By P-T-Var, we have that \( [\Psi] \vdash x : [\sigma] \).

Case S-T-Abs. Suppose \( \Psi \vdash \lambda x : \sigma_1. e' : \sigma_1 \rightarrow \sigma_2 \) and \( \Psi, x : \sigma_1 \vdash e' : \sigma_2 \). We have by definition:

\[
\begin{align*}
[\lambda x : \sigma_1. e'] &= \lambda x : [\sigma_1].[e'] \\
[\sigma_1 \rightarrow \sigma_2] &= [\sigma_1] \rightarrow [\sigma_2] \\
[\Psi, x : \sigma_1] &= [\Psi], x : [\sigma_1]
\end{align*}
\]

By induction, we have that \( [\Psi], x : [\sigma_1] \vdash [e'] : [\sigma_2] \).

By P-T-Abs, we have that \( [\Psi] \vdash \lambda x : [\sigma_1].[e'] : [\sigma_1] \rightarrow [\sigma_2] \).

Case S-T-App. Suppose \( \Psi \vdash e_1(e_2) : \sigma \) and \( \Psi \vdash e_1 : \sigma_2 \rightarrow \sigma \) and \( \Psi \vdash e_2 : \sigma_2 \). We have by definition:

\[
\begin{align*}
[e_1(e_2)] &= [e_1][e_2] \\
[\sigma_2 \rightarrow \sigma] &= [\sigma_2] \rightarrow [\sigma]
\end{align*}
\]

By induction, \( [\Psi] \vdash [e_1] : [\sigma_2] \rightarrow [\sigma] \) and \( [\Psi] \vdash [e_2] : [\sigma_2] \). Therefore, \( [\Psi] \vdash [e_1][e_2] : [\sigma] \) by P-T-App.
Case S-T-StringIn-I. Suppose $\Psi \vdash \text{str}[s] : \text{stringin}[r]$. By definition, $[[\text{str}[s]]] = \text{str}[s]$ and $[[\text{stringin}[r]]] = \text{string}$. By P-T-String, $\Theta \vdash \text{str}[s] : \text{string}$.

Case S-T-Concat. Suppose $\Psi \vdash \text{pconcat}(e_1; e_2) : \text{stringin}[r_1 \cdot r_2]$ and $\Psi \vdash e_1 : \text{stringin}[r_1]$ and $\Psi \vdash e_2 : \text{stringin}[r_2]$. We have by definition:

$$[[\text{pconcat}(e_1; e_2)]] = \text{pconcat}([[e_1]]; [[e_2]])$$

$$[[\text{stringin}[r_1]]] = \text{string}$$

$$[[\text{stringin}[r_2]]] = \text{string}$$

$$[[\text{stringin}[r_1 \cdot r_2]]] = \text{string}$$

By induction, $[[\Psi]] \vdash [e_1] : \text{string}$ and $[[\Psi]] \vdash [e_2] : \text{string}$. Thus, $[[\Psi]] \vdash \text{pconcat}([e_1]; [e_2]) : \text{string}$ by P-T-Concat.

Case S-T-Case. Suppose $\Psi \vdash \text{pstrcase}(e_1; e_2; x, y.e_3) : \sigma$ and $\Psi \vdash e_1 : \text{stringin}[r]$ and $\Psi \vdash e_2 : \sigma$ and $\Psi, x : \text{stringin}[^\text{head}(r)], y : \text{stringin}[^\text{tail}(r)] \vdash e_3 : \sigma$. We have by definition:

$$[[\text{pstrcase}(e_1; e_2; x, y.e_3)]] = \text{pstrcase}([[e_1]]; [[e_2]]; x, y, [[e_3]])$$

$$[[\text{stringin}[r]]] = \text{string}$$

$$[[\text{stringin}[\text{head}(r)]]] = \text{string}$$

$$[[\text{stringin}[\text{tail}(r)]]] = \text{string}$$

$$[[\Psi, x : \text{stringin}[\text{head}(r)], y : \text{stringin}[\text{tail}(r)]]] = [[\Psi]], x : \text{string}, y : \text{string}$$

By induction, $[[\Psi]] \vdash [e_1] : \text{string}$ and $[[\Psi]] \vdash [e_2] : [\sigma]$, and $[[\Psi]], x : \text{string}, y : \text{string} \vdash [e_3] : [\sigma]$. By P-T-Case, we have that $[[\Psi]] \vdash \text{pstrcase}([[e_1]]; [[e_2]]; x, y, [[e_3]]) : [[\Psi]]$.

Case S-T-Replace. Suppose $\Psi \vdash \text{replace}[r](e_1; e_2) : \text{stringin}[\text{replace}(r, r_1, r_2)]$ and $\Psi \vdash e_1 : \text{stringin}[r_1]$ and $\Psi \vdash e_2 : \text{stringin}[r_2]$. We have by definition:

$$[[\text{replace}[r](e_1; e_2)]] = \text{replace}(\lambda x : \text{string}.[[e_1]]; [[e_2]])$$

$$[[\text{stringin}[r_1]]] = \text{string}$$

$$[[\text{stringin}[r_2]]] = \text{string}$$

$$[[\text{stringin}[\text{replace}(r, r_1, r_2)]]] = \text{string}$$

By induction, we have that $[[\Psi]] \vdash [e_1] : \text{string}$ and $[[\Psi]] \vdash [e_2] : \text{string}$. By P-T-Regex, we have that $[[\Psi]] \vdash \text{regex}[r] : \text{regex}$. By P-T-Replace, we have that $[[\Psi]] \vdash \text{replace}(\lambda x : \text{string}.[[e_1]]; [[e_2]]) : [[\Psi]]$.

Case S-T-SafeCoerce. Suppose $\Psi \vdash \text{coerce}[r](e) : \text{stringin}[r]$ and $\Psi \vdash e : \text{stringin}[r']$. By definition, $[[\text{coerce}[r](e)]] = [e]$. By induction, $[[\Psi]] \vdash [e] : [[\text{stringin}[r']]]$.

Case S-T-Check. Suppose $\Psi \vdash \text{rcheck}[r](e_0; x.e_1; e_2) : \sigma$ where $\Psi \vdash e_0 : \text{stringin}[r']$ and $\Psi, x : \text{stringin}[r] \vdash e_1 : \sigma$ and $\Psi \vdash e_2 : \sigma$. We have by definition:

$$[[\text{rcheck}[r](e_0; x.e_1; e_2)]] = \text{rcheck}(\lambda x : \text{string}.[[e_1]]; [[e_0]]; [[e_2]])$$

$$[[\text{stringin}[r']]] = \text{string}$$

$$[[\text{stringin}[r]]] = \text{string}$$

$$[[\Psi, x : \text{stringin}[r]]] = [[\Psi]], x : \text{string}$$

11
By induction, we have that $\llbracket \Psi \rrbracket \vdash [e_0] : \text{string}$ and $\llbracket \Psi \rrbracket, x : \text{string} \vdash [e_1] : \sigma$ and $\llbracket \Psi \rrbracket \vdash [e_2] : \sigma$.

By P-T-Regex, we have that $\llbracket \Psi \rrbracket \vdash \text{rx} \cdot r : \text{regex}$.

By P-T-Abs and P-T-App, we have that $\llbracket \Psi \rrbracket \vdash (\lambda x : \text{string}.[e_1])([e_0]) : \sigma$.

By P-T-Check, we have that $\llbracket \Psi \rrbracket \vdash \text{pcheck}(\text{rx} \cdot r; [e_0]; (\lambda x : \text{string}.[e_1])([e_0]); [e_2]) : \sigma$.

Assumption L (Substitution Translation). $\llbracket [v/x]e \rrbracket = \llbracket [v]x \rrbracket[\llbracket e \rrbracket]$.

Definition 9 (Multistep). We write $\iota \mapsto^* \iota'$ for the reflexive, transitive closure of the stepping judgement.

Assumption M (Multistep Closure). The following closure properties hold:

1. If $\iota_1 \mapsto^* \iota'_1$ then $\bar{\iota}_1(\iota_2) \mapsto^* \bar{\iota}_1(\iota'_2)$.
2. If $\iota_2 \mapsto^* \iota'_2$ then $\bar{\iota}_1(\iota_2) \mapsto^* \bar{\iota}_1(\iota'_2)$.
3. If $\iota_1 \mapsto^* \iota'_1$ then $\text{pconcat}(\iota_1; \iota_2) \mapsto^* \text{pconcat}(\iota'_1; \iota_2)$.
4. If $\iota_2 \mapsto^* \iota'_2$ then $\text{pconcat}(\text{str} \cdot s_1; \iota_2) \mapsto^* \text{pconcat}(\text{str} \cdot s_1; \iota'_2)$.
5. If $\iota_1 \mapsto^* \iota'_1$ then $\text{pstrcase}(\iota_1; \iota_2; x, y, \iota_3) \mapsto^* \text{pstrcase}(\iota'_1; \iota_2; x, y, \iota_3)$.
6. If $\iota_1 \mapsto^* \iota'_1$ then $\text{preplace}(\iota_1; \iota_2; \iota_3) \mapsto^* \text{preplace}(\iota'_1; \iota_2; \iota_3)$.
7. If $\iota_2 \mapsto^* \iota'_2$ then $\text{preplace}(\text{rx} \cdot r; \iota_2; \iota_3) \mapsto^* \text{preplace}(\text{rx} \cdot r; \iota'_2; \iota_3)$.
8. If $\iota_3 \mapsto^* \iota'_3$ then $\text{preplace}(\text{rx} \cdot r; \text{str} \cdot s; \iota_3) \mapsto^* \text{preplace}(\text{rx} \cdot r; \text{str} \cdot s; \iota'_3)$.
9. If $\iota_1 \mapsto^* \iota'_1$ then $\text{pcheck}(\iota_1; \iota_2; \iota_3; \iota_4) \mapsto^* \text{pcheck}(\iota'_1; \iota_2; \iota_3; \iota_4)$.
10. If $\iota_2 \mapsto^* \iota'_2$ then $\text{pcheck}(\text{rx} \cdot r; \iota_2; \iota_3; \iota_4) \mapsto^* \text{pcheck}(\text{rx} \cdot r; \iota'_2; \iota_3; \iota_4)$.

Theorem 10 (Translation Correctness). If $\emptyset \vdash e : \sigma$ and $e \mapsto e'$ then $\llbracket e \rrbracket \mapsto^* \llbracket e' \rrbracket$.

Proof. By induction on evaluation and typing.

Case SS-E-AppLeft. Suppose $e_1(e_2) \mapsto e'_1(e_2)$ and $e_1 \mapsto e'_1$. We have by definition that

$$\llbracket e_1(e_2) \rrbracket = \llbracket e_1 \rrbracket(\llbracket e_2 \rrbracket)$$
$$\llbracket e'_1(e_2) \rrbracket = \llbracket e'_1 \rrbracket(\llbracket e_2 \rrbracket)$$

The only typing rule that applies is S-T--App, so $\emptyset \vdash e_1 : \sigma_2 \rightarrow \sigma$.

Inductively, we have that $\llbracket e_1 \rrbracket \mapsto^* \llbracket e'_1 \rrbracket$.

By Assumption L, we have that $\llbracket e_1 \rrbracket(\llbracket e_2 \rrbracket) \mapsto^* \llbracket e'_1 \rrbracket(\llbracket e_2 \rrbracket)$. 

\[ \square \]
**Case SS-E-AppRight.** Suppose \( v_1(e_2) \mapsto v_1(e'_2) \) and \( e_2 \mapsto e'_2 \). We have by definition that

\[
\begin{align*}
[v_1(e_2)] &= [v_1](\llbracket e_2 \rrbracket) \\
[v_1(e'_2)] &= [v_1](\llbracket e'_2 \rrbracket)
\end{align*}
\]

The only typing rule that applies is S-T-App, so \( \emptyset \vdash e_2 : \sigma_2 \).

Inductively, we have that \( [e_2] \mapsto^* [e'_2] \).

By Assumption [M2], we have that \( [v_1](\llbracket e_2 \rrbracket) \mapsto^* [v_1](\llbracket e'_2 \rrbracket) \).

**Case SS-E-AppAbs.** Suppose \( (\lambda x : \sigma_2.e')(v_2) \mapsto [v_2/x]e' \). We have by definition and Assumption [L] that

\[
\begin{align*}
((\lambda x : \sigma_2.e')(v_2)) &= (\lambda x : [\sigma_2],[e'])(v_2) \\
[\llbracket v_2/x \rrbracket e'] &= [\llbracket v_2 \rrbracket/x][\llbracket e' \rrbracket]
\end{align*}
\]

By PS-E-AppAbs, we have that \( (\lambda x : [\sigma],[e'])(v_2) \mapsto \llbracket [v_2] \rrbracket/x][\llbracket e' \rrbracket] \).

**Case SS-E-Concat-Left.** Suppose \( rconcat(e_1; e_2) \mapsto rconcat(e'_1; e_2) \) and \( e_1 \mapsto e'_1 \). We have by definition that

\[
\begin{align*}
\llbracket rconcat(e_1; e_2) \rrbracket &= pconcat(\llbracket e_1 \rrbracket; \llbracket e_2 \rrbracket) \\
\llbracket rconcat(e'_1; e_2) \rrbracket &= pconcat(\llbracket e'_1 \rrbracket; \llbracket e_2 \rrbracket)
\end{align*}
\]

The only typing rule that applies is S-T-Concat, so \( \emptyset \vdash e_1 : \text{stringin}[r_1] \).

Inductively, we have that \( [e_1] \mapsto^* [e'_1] \).

By Assumption [M3], we have that \( pconcat([e_1]; [e_2]) \mapsto^* pconcat([e'_1]; [e_2]) \).

**Case SS-E-Concat-Right.** Suppose \( rconcat(rstr[s]; e_2) \mapsto rconcat(rstr[s]; e'_2) \) and \( e_2 \mapsto e'_2 \). We have by definition that

\[
\begin{align*}
\llbracket rconcat(rstr[s]; e_2) \rrbracket &= pconcat(\text{str}[s]; \llbracket e_2 \rrbracket) \\
\llbracket rconcat(rstr[s]; e'_2) \rrbracket &= pconcat(\text{str}[s]; \llbracket e'_2 \rrbracket)
\end{align*}
\]

The only typing rule that applies is S-T-Concat, so \( \emptyset \vdash e_2 : \text{stringin}[r_2] \).

Inductively, we have that \( [e_2] \mapsto^* [e'_2] \).

By Assumption [M4], we have that \( pconcat(\text{str}[s]; [e_2]) \mapsto^* pconcat(\text{str}[s]; [e'_2]) \).

**Case SS-E-Concat.** Suppose \( rconcat(rstr[s_1]; rstr[s_2]) \mapsto rstr[s_1s_2] \). We have by definition that

\[
\begin{align*}
\llbracket rconcat(rstr[s_1]; rstr[s_2]) \rrbracket &= pconcat(\text{str}[s_1]; \llbracket rstr[s_2] \rrbracket) \\
\llbracket rstr[s_1s_2] \rrbracket &= \text{str}[s_1s_2]
\end{align*}
\]

By PS-E-Concat, we have \( pconcat(\text{str}[s_1]; \text{str}[s_2]) \mapsto \text{str}[s_1s_2] \).
Case SS-E-Case-Left. Suppose \( \text{rstrcase}(e_1; e_2; x, y, e_3) \rightarrow \text{rstrcase}(e'_1; e_2; x, y, e_3) \) and \( e_1 \rightarrow e'_1 \). We have by definition that:

\[
\begin{align*}
\text{rstrcase}(e_1; e_2; x, y, e_3) &= \text{pstrcase}([e_1]; [e_2]; x, y, [e_3]) \\
\text{rstrcase}(e'_1; e_2; x, y, e_3) &= \text{pstrcase}([e'_1]; [e_2]; x, y, [e_3])
\end{align*}
\]

The only typing rule that applies is S-T-Case, so \( \emptyset \vdash e_1 : \text{stringin}[r] \).

Inductively, \( [e_1] \mapsto^* [e'_1] \).

By Assumption M.8, we have that \( \text{pstrcase}([e_1]; [e_2]; x, y, [e_3]) \mapsto^* \text{pstrcase}([e'_1]; [e_2]; x, y, [e_3]) \).

Case SS-E-Case-Epsilon. Suppose \( \text{rstrcase}(\text{rstr}[e]; e_2; x, y, e_3) \rightarrow e_2 \). We have by definition that:

\[
\begin{align*}
\text{rstrcase}(\text{rstr}[e]; e_2; x, y, e_3) &= \text{pstrcase}(\text{str}[e]; [e_2]; x, y, [e_3])
\end{align*}
\]

By PS-E-Case-Epsilon, we have that \( \text{pstrcase}(\text{str}[e]; [e_2]; x, y, [e_3]) \rightarrow [e_2] \).

Case SS-E-Case-Cons. Suppose \( \text{rstrcase}(\text{rstr}[as]; e_2; x, y, e_3) \rightarrow [\text{rstr}[a], \text{rstr}[s]/x, y]e_3 \). We have by Assumption L and definition that

\[
\begin{align*}
\text{rstrcase}(\text{rstr}[as]; e_2; x, y, e_3) &= \text{pstrcase}(\text{str}[as]; [e_2]; x, y, [e_3]) \\
[\text{rstr}[a], \text{rstr}[s]/x, y]e_3 &= [\text{str}[a], \text{str}[s]/x, y][e_3]
\end{align*}
\]

By PS-E-Case-Cons, we have that \( \text{pstrcase}(\text{str}[as]; [e_2]; x, y, [e_3]) \mapsto^* [\text{str}[a], \text{str}[s]/x, y][e_3] \).

Case SS-E-Replace-Left. Suppose \( \text{replace}[r](e_1; e_2) \rightarrow \text{replace}[r](e'_1; e_2) \) and \( e_1 \rightarrow e'_1 \). We have by definition that

\[
\begin{align*}
\text{replace}[r](e_1; e_2) &= \text{preplace}(\text{rx}[r]; [e_1]; [e_2]) \\
\text{replace}[r](e'_1; e_2) &= \text{preplace}(\text{rx}[r]; [e'_1]; [e_2])
\end{align*}
\]

The only typing rule that applies is S-T-Replace, so \( \emptyset \vdash e_1 : \text{stringin}[r_1] \).

Inductively, we have that \( [e_1] \mapsto^* [e'_1] \).

By Assumption M.7, we have that \( \text{preplace}(\text{rx}[r]; [e_1]; [e_2]) \mapsto^* \text{preplace}(\text{rx}[r]; [e'_1]; [e_2]) \).

Case SS-E-Replace-Right. Suppose \( \text{replace}[r](e_1; e_2) \rightarrow \text{replace}[r](e_1; e'_2) \) and \( e_2 \rightarrow e'_2 \). By definition,

\[
\begin{align*}
\text{replace}[r](e_1; e_2) &= \text{preplace}(\text{rx}[r]; [e_1]; [e_2]) \\
\text{replace}[r](e_1; e'_2) &= \text{preplace}(\text{rx}[r]; [e_1]; [e'_2])
\end{align*}
\]

The only typing rule that applies is S-T-Replace, so \( \emptyset \vdash e_2 : \text{stringin}[r_2] \).

Inductively, we have that \( [e_2] \mapsto^* [e'_2] \).

By Assumption M.8, we have that \( \text{preplace}(\text{rx}[r]; [e_1]; [e_2]) \mapsto^* \text{preplace}(\text{rx}[r]; [e_1]; [e'_2]) \).
Case SS-E-Replace. Suppose \( rreplace[r](rstr[s_1]; rstr[s_2]) \mapsto rstr[lreplace(r, s_1, s_2)] \). By definition,

\[
[rreplace[r](rstr[s_1]; rstr[s_2])] = preplace(rx[r]; str[s_1]; str[s_2])
\]

\[
[rstr[lreplace(r, s_1, s_2)]] = str[lreplace(r, s_1, s_2)]
\]

By PS-E-Replace, we have that \( preplace(rx[r]; str[s_1]; str[s_2]) \mapsto str[lreplace(r, s_1, s_2)] \).

Case SS-E-SafeCoerce-Step. Suppose \( rcoerce[r](e) \mapsto rcoerce[r](e') \) and \( e \mapsto^* e' \). By definition,

\[
[rcoerce[r](e)] = [e]
\]

\[
[rcoerce[r](e')] = [e']
\]

The only typing rule that applies is S-T-SafeCoerce, so \( \emptyset \vdash e' : stringin[r'] \).

Inductively, \([e] \mapsto^* [e']\).

Case SS-E-SafeCoerce. Suppose \( rcoerce[r](rstr[s]) \mapsto rstr[s] \). By definition,

\[
[rcoerce[r](rstr[s])] = str[s]
\]

\[
rstr[s] = str[s]
\]

We have that \( str[s] \mapsto^* str[s] \) because the multistep judgement is reflexive.

Case SS-E-Check-StepLeft. Suppose \( rcheck[r](e_0; x.e_1; e_2) \mapsto rcheck[r](e'_0; x.e_1; e_2) \) and \( e_0 \mapsto e'_0 \).

We have by definition that

\[
[rcheck[r](e_0; x.e_1; e_2)] = pcheck(rx[r]; [e_0]; (\lambda x : string.[e_1])([e_0]); [e_2])
\]

\[
[rcheck[r](e'_0; x.e_1; e_2)] = pcheck(rx[r]; [e'_0]; (\lambda x : string.[e_1])([e'_0]); [e_2])
\]

Inductively, \( e_0 \mapsto^* e'_0 \).

By Assumption [M] 10, we have that

\[
pcheck(rx[r]; [e_0]; (\lambda x : string.[e_1])([e_0]); [e_2]) \mapsto^* pcheck(rx[r]; [e'_0]; (\lambda x : string.[e_1])([e'_0]); [e_2])
\]

Case SS-E-Check-Ok. Suppose \( rcheck[r](rstr[s]; x.e_1; e_2) \mapsto [rstr[s]/x]e_1 \) and \( s \in L\{r\} \). We have by definition that

\[
[rcheck[r](rstr[s]; x.e_1; e_2)] = pcheck(rx[r]; str[s]; (\lambda x : string.[e_1])(str[s]); [e_2])
\]

\[
[rstr[s]/x]e_1 = [str[s]/x][e_1]
\]

By PS-E-Check-OK, we have that

\[
pcheck(rx[r]; str[s]; (\lambda x : string.[e_1])(str[s]); [e_2]) \mapsto (\lambda x : string.[e_1])(str[s])
\]

By PS-E-AppAbs, we have that

\[
(\lambda x : string.[e_1])(str[s]) \mapsto [str[s]/x][e_1]
\]
Case SS-E-Check-NotOk Suppose \( \text{rcheck}[r](r\text{str}[s]; x.e_1; e_2) \mapsto e_2 \) and \( s \notin L\{r\} \). By definition,

\[
[r\text{check}[r](r\text{str}[s]; x.e_1; e_2)] = \text{pcheck}(r_x[r]; \text{str}[s]; (\lambda x : \text{string}.[e_1])(\text{str}[s]); [e_2])
\]

By PS-E-Check-NotOK, we have that

\[
\text{pcheck}(r_x[r]; \text{str}[s]; (\lambda x : \text{string}.[e_1])(\text{str}[s]); [e_2]) \mapsto [e_2]
\]

References

\[ r ::= \epsilon \mid a \mid r \cdot r \mid r + r \mid r^* \quad \text{if } a \in \Sigma \]

**Figure 1:** Syntax of regular expressions over the alphabet \( \Sigma \).

\[ \sigma ::= \sigma \rightarrow \sigma \mid \text{stringin}[r] \quad \text{source types} \]

\[ e ::= x \mid v \mid e(e) \mid \text{concat}(e; e) \mid \text{case}(e; e; x, y, e) \mid \text{replace}[r](e; e) \mid \text{coerce}[r](e) \mid \text{check}[r](e; x, e; e) \quad \text{source terms} \]

\[ v ::= \lambda x. e \mid \text{string} \quad \text{source values} \]

**Figure 2:** Syntax of \( \lambda \text{RS} \)

\[ \tau ::= \tau \rightarrow \tau \mid \text{string} \mid \text{regex} \quad \text{target types} \]

\[ t ::= x \mid v \mid \lambda (\lambda) \mid \text{concat}(\tau; \tau) \mid \text{case}(\tau; \tau; x, y, \tau) \mid \text{replace}(\tau; \tau; \tau) \mid \text{check}(\tau; \tau; \tau) \quad \text{target terms} \]

\[ \dot{v} ::= \lambda x. t \mid \text{string} \mid \text{regex} \quad \text{target values} \]

**Figure 3:** Syntax of \( \lambda \text{P} \)

\[
\Psi \vdash e : \sigma \\
\Psi ::= \emptyset \mid \Psi, x : \sigma
\]

**Figure 4:** Typing rules for \( \lambda \text{RS} \). The typing context \( \Psi \) is standard.
$$e \mapsto e$$

<table>
<thead>
<tr>
<th>Rule</th>
<th>SS-E-AppLeft</th>
<th>SS-E-AppRight</th>
<th>SS-E-AppAbs</th>
<th>SS-E-Concat-Left</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$e_1 \mapsto e'_1$</td>
<td>$e_2 \mapsto e'_2$</td>
<td>$(\lambda x : \sigma.e) v_2 \mapsto [v_2/x] e$</td>
<td>$\text{rconcat}(e_1; e_2) \mapsto \text{rconcat}(e'_1; e'_2)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>SS-E-Concat-Right</th>
<th>SS-E-Concat</th>
<th>SS-E-Case-Left</th>
<th>SS-E-Case-Epsilon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$e_2 \mapsto e'_2$</td>
<td>$\text{rconcat}(v_1; e_2) \mapsto \text{rconcat}(v_1; e'_2)$</td>
<td>$e_1 \mapsto e'_1$</td>
<td>$\text{rstrcase}(e_1; e_2; x, y.e_3) \mapsto e_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>SS-E-Replace-Left</th>
<th>SS-E-Replace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$e_1 \mapsto e'_1$</td>
<td>$\text{replace}[r][v_1; e_2] \mapsto \text{replace}[r][v'_1; e_2]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>SS-E-SafeCoerce-Step</th>
<th>SS-E-SafeCoerce</th>
<th>SS-E-Check-StepLeft</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$e \mapsto e'$</td>
<td>$\text{rcoerce}<a href="e">r</a> \mapsto \text{rcoerce}<a href="e'">r</a>$</td>
<td>$\text{rcheck}[r][e; x.e_1; e_2] \mapsto \text{rcheck}[r][e'; x.e_1; e_2]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>SS-E-Check-StepRight</th>
<th>SS-E-Check-NotOk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$s \in \mathcal{L}(r)$</td>
<td>$s \notin \mathcal{L}(r)$</td>
</tr>
</tbody>
</table>

$$\text{rcheck}[r][\text{rstr}[s]; x.e_1; e_2] \mapsto \text{rstr}[s/x] e_1$$

$$\text{rcheck}[r][\text{rstr}[s]; x.e_1; e_2] \mapsto e_2$$

**Figure 5:** Small step semantics for $\lambda_{RS}$. 

18
\[ \Theta \vdash \iota : \tau \quad \Theta ::= \emptyset \mid \Theta, x : \tau \]

<table>
<thead>
<tr>
<th>P-T-Var</th>
<th>P-T-ABS</th>
<th>P-T-APP</th>
<th>P-T-STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x : \tau \in \Theta )</td>
<td>( \Theta, x : \tau_1 \vdash \iota_2 : \tau_2 )</td>
<td>( \Theta \vdash \lambda x.\iota_2 : \tau_1 \to \tau_2 )</td>
<td>( \Theta \vdash \text{str}[s] : \text{string} )</td>
</tr>
<tr>
<td>( \Theta \vdash x : \tau )</td>
<td>( \Theta \vdash \lambda x.\iota_2 : \tau_1 \to \tau_2 )</td>
<td>( \Theta \vdash \iota_1 (\iota_2) : \iota )</td>
<td>( \Theta \vdash \text{str}[s] : \text{string} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P-T-REGEX</th>
<th>P-T-CONCAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta \vdash \text{rx}[r] : \text{regex} )</td>
<td>( \Theta \vdash \text{pconcat}(\iota_1; \iota_2) : \text{string} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P-T-CASE</th>
<th>P-T-REPLACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta \vdash \iota_1 : \text{string} ) ( \Theta \vdash \iota_2 : \tau ) ( \Theta, x : \text{string}, y : \text{string} \vdash \iota_3 : \tau )</td>
<td>( \Theta \vdash \text{pstrcase}(\iota_1; \iota_2; x; y; \iota_3) : \tau )</td>
</tr>
</tbody>
</table>

| P-T-CHECK   | |
|-------------||
| \( \Theta \vdash \iota_1 : \text{regex} \) \( \Theta \vdash \iota_2 : \text{string} \) \( \Theta \vdash \iota_3 : \tau \) \( \Theta \vdash \iota_4 : \tau \) | \( \Theta \vdash \text{pcheck}(\iota_1; \iota_2; \iota_3; \iota_4) : \tau \) |

**Figure 6:** Typing rules for \( \lambda_P \). The typing context \( \Theta \) is standard.
\[ \lambda x : \tau. \ell \vdash \emptyset_\ell \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS-E-APPLEFT</td>
<td>( \ell_1 \rightarrow \ell'_1 )</td>
</tr>
<tr>
<td>PS-E-APPRIGHT</td>
<td>( \ell_2 \rightarrow \ell'_2 )</td>
</tr>
<tr>
<td>PS-E-APPABS</td>
<td>( (\lambda x : \tau. \ell) \ell_2 \rightarrow \ell_2[x/\ell] )</td>
</tr>
<tr>
<td>PS-E-CONCATLEFT</td>
<td>( \ell_1 \rightarrow \ell'_1 )</td>
</tr>
<tr>
<td>PS-E-CONCATRIGHT</td>
<td>( \ell_2 \rightarrow \ell'_2 )</td>
</tr>
<tr>
<td>PS-E-CONCAT</td>
<td>( \text{concat}(\ell_1; \ell_2) \rightarrow \text{concat}(\ell'_1; \ell'_2) )</td>
</tr>
<tr>
<td>PS-E-CASELEFT</td>
<td>( \ell_1 \rightarrow \ell'_1 )</td>
</tr>
<tr>
<td>PS-E-Case-Cons</td>
<td>( \text{pstrcase}(\ell_1; \ell_2; x, y, \ell_3) \rightarrow \text{pstrcase}(\ell'_1; \ell_2; x, y, \ell_3) )</td>
</tr>
<tr>
<td>PS-E-REPLACE</td>
<td>( \text{replace}(\ell_1; \ell_2; \ell_3) \rightarrow \text{replace}(\ell'_1; \ell_2; \ell_3) )</td>
</tr>
<tr>
<td>PS-E-CheckLeft</td>
<td>( \ell_1 \rightarrow \ell'_1 )</td>
</tr>
<tr>
<td>PS-E-CheckRight</td>
<td>( \ell_2 \rightarrow \ell'_2 )</td>
</tr>
<tr>
<td>PS-E-Check</td>
<td>( \text{pcheck}(\ell_1; \ell_2; \ell_3; \ell_4) \rightarrow \text{pcheck}(\ell'_1; \ell_2; \ell_3; \ell_4) )</td>
</tr>
<tr>
<td>PS-E-Check-OK</td>
<td>( \ell_3 \rightarrow \ell_3 )</td>
</tr>
<tr>
<td>PS-E-Check-NOTOK</td>
<td>( \ell_3 \rightarrow \ell_4 )</td>
</tr>
</tbody>
</table>

**Figure 7:** Small step semantics for \( \lambda_P \)
\[ [\sigma] = \tau \]

\[ [[\text{stringin}[\tau]]] = \text{string} \]
\[ [[\sigma_1 \rightarrow \sigma_2]] = [[\sigma_1]] \rightarrow [[\sigma_2]] \]

\[ [[\emptyset]] = \emptyset \]
\[ [[\Psi, x : \sigma]] = [[\Psi]], x : [[\sigma]] \]

\[ [[e]] = \iota \]

\[ [[x]] = x \]
\[ [[\lambda x : \sigma.e]] = \lambda x : [[\sigma]].[[e]] \]
\[ [[e_1(e_2)]] = [[e_1]]([[e_2]]) \]
\[ [[\text{rstr}[s]]] = \text{str}[s] \]
\[ [[\text{rstrcase}(e_1; e_2; x; y; e_3)]] = \text{pstrcase}([[e_1]]; [[e_2]]; x; y; [[e_3]]) \]
\[ [[\text{rconcat}(e_1; e_2)]] = \text{pconcat}([[e_1]]; [[e_2]]) \]
\[ [[\text{replace}[r](e_1; e_2)]] = \text{preplace}(rx[r]; [[e_1]]; [[e_2]]) \]
\[ [[\text{rcoerce}[r](e)]] = [[e]] \]
\[ [[\text{rcheck}[r](e; e_1; e_2)]] = \text{pcheck}(rx[r]; [[e]]; (\lambda x : \text{string}.[[e_1]])([[e]]); [[e_2]]) \]

\textbf{Figure 8:} Translation from \( \lambda_{RS} \) to \( \lambda_P \)