Statically Typed String Sanitation Inside a Python

Nathan Fulton Cyrus Omar Jonathan Aldrich
Carnegie Mellon University, Pittsburgh, PA
{nathanfu, comar, aldrich}@cs.cmu.edu

ABSTRACT

Web applications must ultimately command systems like web browsers and database engines using strings. Strings derived from improperly sanitized user input can thus be a vector for command injection attacks. In this paper, we introduce *regular string types*, which classify strings known statically to be in a specified regular language. These types come equipped with common operations like concatenation, substitution and coercion, so they can be used to implement, in a conventional manner, the portions of a web application or application framework that must directly construct command strings. Simple type annotations at key interfaces can be used to statically verify that sanitization has been performed correctly without introducing redundant run-time checks. We specify this type system in a minimal typed lambda calculus, λ_{RS}.

To be practical, adopting a specialized type system like this should not require the adoption of a new programming language. Instead, we advocate for extensible type systems: new type system fragments like this should be implemented as libraries atop a mechanism that guarantees that they can be safely composed. We support this with two contributions. First, we specify a translation from λ_{RS} to a language fragment containing only standard strings and regular expressions. Second, taking Python as a language with these constructs, we implement the type system together with the translation as a library using `atlang`, an extensible static type system for Python being developed by the authors.

1. INTRODUCTION

Command injection vulnerabilities are among the most common and severe security vulnerabilities in modern web applications [11]. They arise because web applications, at their boundaries, control external systems using commands represented as strings. For example, web browsers are controlled using HTML and Javascript sent from a server as a string, and database engines execute SQL queries also sent as strings. When these commands include data derived from user input, care must be taken to ensure that the user cannot subvert the intended command by carefully crafting the data they send. For example, a SQL query constructed using string concatenation exposes a SQL injection vulnerability:

```
'SELECT * FROM users WHERE name="' + name + '"'
```

If a malicious user enters the name `'"; DROP TABLE users --'`, the entire database could be erased.

To avoid this problem, the program must *sanitize* user input. For example, in this case, the developer (or, more often, a framework) might define a function `sanitize` that escapes double quotes and existing backslashes with a backslash, which SQL treats safely. Alternatively, it might HTML-encode special characters, which would head off both SQL injection attacks and cross-site scripting attacks. Guaranteeing that user input has already been sanitized before it is used to construct a command is challenging, especially because sanitization and command construction may occur in separately developed and maintained components.

We observe that many such sanitization techniques can be understood using *regular languages* [8]. For example, `name` must be a string in the language described by the regular expression `([^"\]|(\")|(\\))*` – a sequence of characters other than quotation marks and backslashes; these can only appear escaped. This syntax for regular expression patterns can be understood to desugar, in a standard way, to the syntax for regular expressions shown in Figure 1, where $r \cdot r$ is sequencing and $r + r$ is disjunction. We will work with this "core" for simplicity.

In this paper, we present a static type system that tracks the regular language a string belongs to. For example, the output of `sanitize` will be a string in the regular language described by the regular expression above (we identify regular languages by the notation $\mathcal{L}\{r\}$). By leveraging closure and decidability properties of regular languages, the type system tracks the language of a string through uses of a number of operations, including *replacement* of substrings matching a given pattern. This makes it possible to implement sanitation functions – like the one just described – in a conventional manner. The result is a system where the fact that a string has been correctly sanitized is manifest in its type. Missing calls to sanitization functions are detected statically, and, importantly, so are *incorrectly implemented sanitization functions* (i.e. these functions need not be trusted). These guarantees require run-time checks only when going from less precise to more precise types (e.g. along the interfaces with external systems or components that are not under the control of this type system).

We will begin in Sec. 2 by specifying this type system minimally, as a conservative extension of the simply typed

lambda calculus called λ_{RS}. This allows us to specify the guarantees that the type system provides precisely. We also formally specify a translation from this calculus to a typed calculus with only standard strings and regular expressions, intending it as a guide to language implementors interested in building this feature into their own languages. This also demonstrates that no additional space overhead is required.

Waiting for a language designer to build this feature in is unsatisfying in practice. Moreover, we also face a "chicken-and-egg problem": justifying its inclusion into a commonly used language benefits from empirical case studies, but these are difficult to conduct if developers have no way to use the abstraction in real-world code. We take the position that a better path forward for the community is to work within a programming language where such type system fragments can be introduced modularly and orthogonally, as libraries.

In Sec. 3, we show how to implement the type system fragment we have specified using `atlang`, an extensible static type system implemented as a library inside Python. `atlang` leverages local type inference to control the semantics of literal forms, so regular string types can be introduced using string literals without any run-time overhead. Coercions that are known to be safe due to a sublanguage relationship are performed implicitly, also without run-time overhead. This results in a *usably secure* system: working with regular strings differs little from working with standard strings in a language that web developers have already widely adopted.

We conclude after discussing related work in Sec. 4.

2. REGULAR STRING TYPES, MINIMALLY

This section is organized as follows:

- Section 2.1 describes λ_{RS}. We also give proof outlines of type safety and correctness theorems and relevant propositions about regular languages.
- Section 2.2 describes a simple target language, λ_P, with a minimal regular expression library. In Section 3, we will take Python to be such a language.
- Section 2.3 describes the translation from λ_{RS} to λ_P and ensures the correctness result for 2.1 is preserved under this translation.

The accompanying technical report [6] contains more detailed proofs, further discussion of string substitution and language replacement, and a single sheet containing Figures 2-8 for the reader's convenience.

2.1 The Language of Regular Strings

In this section, we define a minimal typed lambda calculus with regular string types called λ_{RS}. The syntax of λ_{RS} is specified in Figure 2, its static semantics in Figure 3 and its evaluation semantics in Figure 4.

There are two type constructors in λ_{RS}, $\rightarrow$ and $\mathsf{stringin}$. Arrow types classify functions, which are introduced via lambda abstraction, $\lambda x.e$, and can be applied, written $e(e)$, in the usual way [7]. Regular string types are of the form $\mathsf{stringin}[r]$, where r is a regular expression. Values of such regular string types take the form $\mathsf{rstr}[s]$, where s is a string (i.e. $s \in \Sigma^\star$, defined in the usual way). The rule S-T-StringIn-I requires that $s \in \mathcal{L}\{r\}$.

λ_{RS} provides several familiar operations on strings. The type system relates these operations over strings to corresponding operations over the regular languages they belong to. Since these operations over regular languages are known to be closed and decidable, we can use these operations as a basis for static analysis of sanitation protocols (we will see a complete example in Section 3).

$$r ::= \epsilon \mid . \mid a \mid r \cdot r \mid r + r \mid r* \qquad a \in \Sigma$$

Figure 1: Regular expressions over the alphabet Σ.

$$\begin{array}{lll} \sigma & ::= & \sigma \rightarrow \sigma \mid \mathsf{stringin}[r] & \text{source types} \\ e & ::= & x \mid \lambda x.e \mid e(e) & \text{source terms} \\ & \mid & \mathsf{rstr}[s] \mid \mathsf{rconcat}(e;e) \mid \mathsf{rstrcase}(e;e;x,y.e) & s \in \Sigma^* \\ & \mid & \mathsf{rcoerce}[r](e) \mid \mathsf{rcheck}[r](e;x.e;e) \mid \mathsf{rreplace}[r](e;e) & \\ v & ::= & \lambda x.e \mid \mathsf{rstr}[s] & \text{source values} \end{array}$$

Figure 2: Syntax of λ_{RS}.

2.1.1 Concatenation and String Decomposition

The S-T-Concat rule is the simplest example of our approach. The rule is sound because the result of concatenating two strings, the first in $\mathcal{L}\{r_1\}$ and the second in $\mathcal{L}\{r_2\}$, will always be in the language $\mathcal{L}\{r_1 \cdot r_2\}$. The rule therefore relates string concatenation to sequential composition of regular expressions.

Whereas concatenation allows the construction of large strings from smaller strings, $\mathsf{rstrcase}(e; e_0; x, y.e_1)$ allows the decomposition, or elimination, of large strings into smaller strings. Intuitively, this operation branches based on whether a string is empty or not, exactly analagous to case analysis on lists in a functional language. The branch for a non-empty string "peels off" the first character, binding it and the remainder of the string to specified variables. The evaluation rules S-E-Case-ϵ and S-E-Case-Concat express this semantics. This construct can be used to implement a standard string indexing operation as a function, given a suitable definition of natural numbers (omitted).

The regular expressions describing the first character and remaining characters of a matching string in the language $\mathcal{L}\{r\}$ must be determined to correctly specify the static semantics. For instance, the first character of a string in the language $\mathcal{L}\{(A \cdot B \cdot C) + (D \cdot E \cdot F)\}$ is in the language $\mathcal{L}\{A + D\}$ and the remaining characters are in the language $\mathcal{L}\{(B \cdot C) + (E \cdot F)\}$.

The regular expression recognizing any one-character prefix of the strings in $\mathcal{L}\{r\}$ is easily defined.

Definition 1 (Definition of $\mathsf{lhead}(r)$)**.** The relation $\mathsf{lhead}(r) = r'$ is defined in terms of the structure of r:

- $\mathsf{lhead}(\epsilon) = \epsilon$
- $\mathsf{lhead}(.) = a_1 + a_2 + ... + a_n$ for all $a_i \in \Sigma$ where $|\Sigma| = n$.
- $\mathsf{lhead}(a) = a$
- $\mathsf{lhead}(r_1 \cdot r_2) = \mathsf{lhead}(r_1)$
- $\mathsf{lhead}(r_1 + r_2) = \mathsf{lhead}(r_1) + \mathsf{lhead}(r_2)$
- $\mathsf{lhead}(r*) = \epsilon + \mathsf{lhead}(r)$

Given this definition of $\mathsf{lhead}(r)$, regular expression derivatives [2] provide a useful tool for defining $\mathsf{ltail}(r)$.

Definition 2 (Brzozowski's Derivative)**.** The *derivative of r with respect to s* is denoted by $\delta_s(r)$ and is $\delta_s(r) = \{t | st \in \mathcal{L}\{r\}\}$.

Definition 2 is equivalent to the definition given in [2], although we refer the unfamiliar reader to [12]. Definition 2

$\boxed{\Psi \vdash e : \sigma}$ $\quad \Psi ::= \emptyset \mid \Psi, x : \sigma$

$$\text{S-T-Var}\ \frac{x : \sigma \in \Psi}{\Psi \vdash x : \sigma} \qquad \text{S-T-Abs}\ \frac{\Psi, x : \sigma_1 \vdash e : \sigma_2}{\Psi \vdash \lambda x.e : \sigma_1 \to \sigma_2}$$

$$\text{S-T-App}\ \frac{\Psi \vdash e_1 : \sigma_2 \to \sigma \qquad \Psi \vdash e_2 : \sigma_2}{\Psi \vdash e_1(e_2) : \sigma} \qquad \text{S-T-Stringin-I}\ \frac{s \in \mathcal{L}\{r\}}{\Psi \vdash \mathsf{rstr}[s] : \mathsf{stringin}[r]}$$

$$\text{S-T-Concat}\ \frac{\Psi \vdash e_1 : \mathsf{stringin}[r_1] \qquad \Psi \vdash e_2 : \mathsf{stringin}[r_2]}{\Psi \vdash \mathsf{rconcat}(e_1; e_2) : \mathsf{stringin}[r_1 \cdot r_2]}$$

$$\text{S-T-Case}\ \frac{\Psi \vdash e_1 : \mathsf{stringin}[r] \qquad \Psi \vdash e_2 : \sigma \qquad \Psi, x : \mathsf{stringin}[\mathsf{lhead}(r)], y : \mathsf{stringin}[\mathsf{ltail}(r)] \vdash e_3 : \sigma}{\Psi \vdash \mathsf{rstrcase}(e_1; e_2; x, y.e_3) : \sigma}$$

$$\text{S-T-SafeCoerce}\ \frac{\Psi \vdash e : \mathsf{stringin}[r'] \qquad \mathcal{L}\{r'\} \subseteq \mathcal{L}\{r\}}{\Psi \vdash \mathsf{rcoerce}[r](e) : \mathsf{stringin}[r]}$$

$$\text{S-T-Check}\ \frac{\Psi \vdash e_0 : \mathsf{stringin}[r_0] \qquad \Psi, x : \mathsf{stringin}[r] \vdash e_1 : \sigma \qquad \Psi \vdash e_2 : \sigma}{\Psi \vdash \mathsf{rcheck}[r](e_0; x.e_1; e_2) : \sigma}$$

$$\text{S-T-Replace}\ \frac{\Psi \vdash e_1 : \mathsf{stringin}[r_1] \qquad \Psi \vdash e_2 : \mathsf{stringin}[r_2] \qquad \mathsf{lreplace}(r; r_1; r_2) = r'}{\Psi \vdash \mathsf{rreplace}[r](e_1; e_2) : \mathsf{stringin}[r']}$$

Figure 3: Typing rules for λ_{RS}. The typing context Ψ is standard.

is equivalent to Definition 3.1 in both papers. We now define $\mathsf{ltail}(r)$ using derivatives with respect to $\mathsf{lhead}(r)$.

Definition 3 (Definition of ltail(r))**.** The relation $\mathsf{ltail}(r) = r'$ is defined in terms of $\mathsf{lhead}(r)$. Note that $\mathsf{lhead}(r) = a_1 + a_2 + ... + a_i$. We define $\mathsf{ltail}(r) = \delta_{a_1}(r) + \delta_{a_2}(r) + ... + \delta_{a_i}(r)$.

The S-T-Case rule, which is defined in terms of these operations, thus relates the result of "peeling off" the first character of a string to regular expression derivatives.

2.1.2 Coercion

The λ_{RS} language supports two forms of coercion. Safe coercions, written $\mathsf{rcoerce}[r](e)$, allow passage to a "smaller" regular language. Such coercions will always succeed. Conversely, checked coercions, written $\mathsf{rcheck}[r](e_0; x.e_1; e_2)$, allow for passage to regular languages that are not necessarily smaller. Checked coercions branch based on whether the coercion succeeded or not.

The rule S-T-SafeCoerce checks for language inclusion, which we write $\mathcal{L}\{r_1\} \subseteq \mathcal{L}\{r_2\}$. Language inclusion is a decidable relation. As a result, the rule S-E-SafeCoerce does not need to perform any checks. The rule S-T-Check, conversely, does not perform any static checks on the two languages involved, only checking that the two branches have the same type. The checks are performed dynamically, by rules S-E-Check-Ok and S-E-Check-NotOk.

In our calculus, both forms of coercion are explicit. For safe coercions, it is often useful for the coercion to be performed implicitly. This can be seen as a form of subtyping for regular string types [5]. In practice, subtyping is crucial

$\boxed{e \Downarrow v}$

$$\text{S-E-Abs}\ \frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \text{S-E-App}\ \frac{e_1 \Downarrow \lambda x.e_3 \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_3 \Downarrow v}{e_1(e_2) \Downarrow v}$$

$$\text{S-E-RStr}\ \frac{}{\mathsf{rstr}[s] \Downarrow \mathsf{rstr}[s]} \qquad \text{S-E-Concat}\ \frac{e_1 \Downarrow \mathsf{rstr}[s_1] \qquad e_2 \Downarrow \mathsf{rstr}[s_2]}{\mathsf{rconcat}(e_1; e_2) \Downarrow \mathsf{rstr}[s_1 s_2]}$$

$$\text{S-E-Case-}\epsilon\ \frac{e_1 \Downarrow \mathsf{rstr}[\epsilon] \qquad e_2 \Downarrow v_2}{\mathsf{rstrcase}(e_1; e_2; x, y.e_3) \Downarrow v_2}$$

$$\text{S-E-Case-Concat}\ \frac{e_1 \Downarrow \mathsf{rstr}[as] \qquad [\mathsf{rstr}[a], \mathsf{rstr}[s]/x, y]e_3 \Downarrow v_3}{\mathsf{rstrcase}(e_1; e_2; x, y.e_3) \Downarrow v_3}$$

$$\text{S-E-SafeCoerce}\ \frac{e \Downarrow \mathsf{rstr}[s]}{\mathsf{rcoerce}[r](e) \Downarrow \mathsf{rstr}[s]}$$

$$\text{S-E-Check-Ok}\ \frac{e \Downarrow \mathsf{rstr}[s] \qquad s \in \mathcal{L}\{r\} \qquad [\mathsf{rstr}[s]/x]e_1 \Downarrow v}{\mathsf{rcheck}[r](e; x.e_1; e_2) \Downarrow v}$$

$$\text{S-E-Check-NotOk}\ \frac{e \Downarrow \mathsf{rstr}[s] \qquad s \notin \mathcal{L}\{r\} \qquad e_2 \Downarrow v}{\mathsf{rcheck}[r](e; x.e_1; e_2) \Downarrow v}$$

$$\text{S-E-Replace}\ \frac{e_1 \Downarrow \mathsf{rstr}[s_1] \qquad e_2 \Downarrow \mathsf{rstr}[s_2] \qquad \mathsf{subst}(r; s_1; s_2) = s}{\mathsf{rreplace}[r](e_1; e_2) \Downarrow \mathsf{rstr}[s]}$$

Figure 4: Big step semantics for λ_{RS}

.

to the usability of our system due to the nature of the replacement operator. For simplicity, we do not include this relation; however, subtyping for regular strings is present in previous treatments of our work [4, 5]. The implementation discussed in Section 3 also provides subtyping between regular string types.

2.1.3 Replacement

The premier operation for working with regular strings in λ_{RS} is replacement, written $\mathsf{rreplace}[r](e_1; e_2)$. It behaves analogously to `str_replace` in PHP and `String.replace` in Java, differing in that the replacement pattern r must be statically given. The evaluation rule S-E-replace is defined via the relation $\mathsf{subst}(r; s_1; s_2) = s$. For simplicity, we do not inductively define this standard relation.

Definition 4 (`subst`)**.** The relation $\mathsf{subst}(r; s_1; s_2) = s$ produces a string s in which all substrings of s_1 in $\mathcal{L}\{r\}$ are replaced with s_2.

The typing rule S-T-Replace requires computing the regular language of the string resulting from this operation given the languages of s_1 and s_2, written as the relation $\mathsf{lreplace}(r; r_1; r_2) = r'$. Given an automata-oriented interpretation of regular languages, it may be helpful to think in terms of replacing sub-automata.

A complete definition of $\mathsf{lreplace}$ would also give an rewrite system with correctness and termination proofs, which is beyond the scope of this paper. Instead, we provide an abstract definition of the relation and state necessary properties.

Definition 5 (`lreplace`)**.** The relation $\mathsf{lreplace}(r; r_1; r_2) = r'$ relates r, r_1, and r_2 to a language r' containing all strings of r_1 except that any substring $s_{pre}ss_{post} \in \mathcal{L}\{r_1\}$ where $s \in \mathcal{L}\{r\}$ is replaced by the set of strings $s_{pre}s_2s_{post}$ for all $s_2 \in \mathcal{L}\{r_2\}$ (the prefix and postfix positions may be empty). This procedure saturates the string.

Proposition 6 (Closure)**.** *If $\mathcal{L}\{r\}, \mathcal{L}\{r_1\}$ and $\mathcal{L}\{r_2\}$ are regular languages, then $\mathcal{L}\{\mathsf{lreplace}(r; r_1; r_2)\}$ is also a regular language.*

Proof Sketch. Algorithms for the inclusion problem may be adopted to identify any sublanguage $x \subseteq r$ of r_1. The language x_{pre} can be computed by taking total derivatives until the remaining language equals x; x_{post} can be computed in a similar way after reversal. Then r' is $x_{pre}r_2x_{post}$. □

Proposition 7 (Substitution Correspondence)**.** *If $s_1 \in \mathcal{L}\{r_1\}$ and $s_2 \in \mathcal{L}\{r_2\}$ then $\mathsf{subst}(r; s_1; s_2) \in \mathcal{L}\{\mathsf{lreplace}(r; r_1; r_2)\}$.*

Proof Sketch. The proposition follows from the definitions of subst and lreplace; note that language substitutions over-approximate string substitutions. □

2.1.4 Safety

In this section, we establish type soundness for λ_{RS}. The theorem relies upon the definitions and propositions given above and some basic properties of regular languages.

Lemma 8 (Properties of Regular Languages.)**.**

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.*

If any of these properties are unfamiliar, the reader may refer to a standard text on the subject [8].

We also need to establish a well-formedness lemma, which simply expresses the totality of the operations over regular expressions used as premises in the rules in Figure 3.

Lemma 9. *If $\Psi \vdash e : \mathsf{stringin}[r]$ then r is a well-formed regular expression.*

Proof Sketch. The only non-trivial cases are S-T-Replace and S-T-Concat. The first follows from Proposition 6, and the other follows from the fact that lhead and ltail always produce well-formed expressions. □

Safety for the string fragment of λ_{RS} requires validating that the type system's definition is justified by our theorems about regular languages.

We avoid the most significant issues inherent to proofs related to big-step semantics by avoiding non-termination in our specification. The simply typed lambda calculus terminates, and our conservative extension clearly terminates because new binding structures are not introduced and termination (i.e. decidability) of subst and lreplace is known. Even in a non-terminating calculus, type safety would hold, but a more careful treatment or treatment for a language with non-termination would need to proceed by either a coinductive argument [1] or with a small-step semantics.

Theorem 10 (Type Safety.)**.** *If $\emptyset \vdash e : \sigma$ then $e \Downarrow v$ and $\emptyset \vdash v : \sigma$.*

$$
\begin{array}{lcll}
\tau & ::= & \tau \to \tau \mid \mathsf{string} \mid \mathsf{regex} & \text{target types} \\
\iota & ::= & x \mid \lambda x.\iota \mid \iota(\iota) & \text{target terms} \\
 & \mid & \mathsf{str}[s] \mid \mathsf{concat}(\iota;\iota) \mid \mathsf{strcase}(\iota;\iota;x,y.\iota) & \\
 & \mid & \mathsf{rx}[r] \mid \mathsf{check}(\iota;\iota;\iota;\iota) \mid \mathsf{replace}(\iota;\iota;\iota) & \\
\dot{v} & ::= & \lambda x.\iota \mid \mathsf{str}[s] \mid \mathsf{rx}[r] & \text{target values}
\end{array}
$$

Figure 5: Syntax for the target language, λ_P, containing strings and statically constructed regular expressions.

Proof Sketch. By induction on the typing relation. The S-T-Concat case requires Lemma 8.1, the S-T-Check case requires Lemma 8.2 and the S-T-Replace case appeals to Proposition 7. □

We can also define a canonical forms lemma for regular strings.

Lemma 11 (Canonical Forms for String Fragment of λ_{RS})**.** *If $\emptyset \vdash e : \mathsf{stringin}[r]$ and $e \Downarrow v$ then $v = \mathsf{rstr}[s]$.*

Proof Sketch. The proof is again by induction on the typing relation, following straightforwardly. □

2.1.5 The Security Theorem

The chief benefit of λ_{RS} is its security theorem, which states that any value of a regular string type is recognized by the regular language corresponding to the regular expression the type is indexed by. This is beneficial because this ensure that membership in a regular language known to be secure becomes manifest in the type of the string, rather than being a property that must be established extralinguistically.

Theorem 12 (Correctness of Input Sanitation for λ_{RS})**.** *If $\emptyset \vdash e : \mathsf{stringin}[r]$ and $e \Downarrow \mathsf{rstr}[s]$ then $s \in \mathcal{L}\{r\}$.*

Proof Sketch. The theorem follows directly from type safety, canonical forms for λ_{RS}, and inversion of S-T-Stringin-I, the typing rule for terms of the form $\mathsf{rstr}[s]$. □

2.2 Target Language

Our next major technical result, stated in Sec. 2.3, establishes that the security property is preserved under translation into λ_P, which we now define.

The system λ_P is another straight-forward extension of a simply typed lambda calculus with a string type and a regex type, as well as some operations – such as concatenation and replacement – found in the standard libraries of many programming languages. The operations of λ_P correspond to "run-time" versions of the operations performed statically in λ_{RS} in a way made precise by the translation rules described in the next section. The language λ_P is so-called because it is reminscent of popular web programming languages, such as **P**ython or **P**HP, albeit statically typed. We will discuss an implementation within the former in the next section.

The grammar of λ_P is defined in Figure 5. The typing rules P-T- are defined in Figure 6 and a big-step semantics is defined by the rules P-E- in Figure 7. The semantics are straightforward given the discussion of the corresponding operations in the previous section.

$\boxed{\Theta \vdash \iota : \tau}$ $\quad \Theta ::= \emptyset \mid \Theta, x : \tau$

$$\text{P-T-Var}\ \frac{x : \tau \in \Theta}{\Theta \vdash x : \tau} \qquad \text{P-T-Abs}\ \frac{\Theta, x : \tau_1 \vdash \iota_2 : \tau_2}{\Theta \vdash \lambda x.\iota_2 : \tau_1 \to \tau_2}$$

$$\text{P-T-App}\ \frac{\Theta \vdash \iota_1 : \tau_2 \to \tau \qquad \Theta \vdash \iota_2 : \tau_2}{\Theta \vdash \iota_1(\iota_2) : \iota} \qquad \text{P-T-String}\ \frac{}{\Theta \vdash \mathsf{str}[s] : \mathsf{string}}$$

$$\text{P-T-Regex}\ \frac{}{\Theta \vdash \mathsf{rx}[r] : \mathsf{regex}} \qquad \text{P-T-Concat}\ \frac{\Theta \vdash \iota_1 : \mathsf{string} \qquad \Theta \vdash \iota_2 : \mathsf{string}}{\Theta \vdash \mathsf{concat}(\iota_1; \iota_2) : \mathsf{string}}$$

$$\text{P-T-Case}\ \frac{\Theta \vdash \iota_1 : \mathsf{string} \qquad \Theta \vdash \iota_2 : \tau \qquad \Theta, x : \mathsf{string}, y : \mathsf{string} \vdash \iota_3 : \tau}{\Theta \vdash \mathsf{strcase}(\iota_1; \iota_2; x, y.\iota_3) : \tau}$$

$$\text{P-T-Replace}\ \frac{\Theta \vdash \iota_1 : \mathsf{regex} \qquad \Theta \vdash \iota_2 : \mathsf{string} \qquad \Theta \vdash \iota_3 : \mathsf{string}}{\Theta \vdash \mathsf{replace}(\iota_1; \iota_2; \iota_3) : \mathsf{string}}$$

$$\text{P-T-Check}\ \frac{\Theta \vdash \iota_r : \mathsf{regex} \qquad \Theta \vdash \iota_1 : \mathsf{string} \qquad \Theta \vdash \iota_2 : \sigma \qquad \Theta \vdash \iota_3 : \sigma}{\Theta \vdash \mathsf{check}(\iota_r; \iota_1; \iota_2; \iota_3) : \sigma}$$

Figure 6: Typing rules for λ_P. The typing context Θ is standard.

2.2.1 Safety

Type safety for λ_P is straight-forward, but is necessary in order to establish the correctness of our translation. Again, we elide a more careful treatment by treating a special case where our language terminates per the discussion above.

Theorem 13 (Safety for λ_P)**.** *If $\emptyset \vdash \iota : \tau$ then $\iota \Downarrow \dot{v}$ and $\emptyset \vdash \dot{v} : \tau$.*

We can also define canonical forms for regular expressions and strings in the usual way:

Lemma 14 (Canonical Forms for Target Language)**.**

1. *If $\emptyset \vdash \iota : \mathsf{regex}$ then $\iota \Downarrow \mathsf{rx}[r]$ such that r is a well-formed regular expression.*
2. *If $\emptyset \vdash \iota : \mathsf{string}$ then $\iota \Downarrow \mathsf{str}[s]$.*

2.3 Translation from λ_{RS} to λ_P

The translation from λ_{RS} to λ_P is defined in Figure 8. The coercion cases are most interesting. If the safety of coercion in manifest in the types of the expressions, then no runtime check is inserted. If the safety of coercion is not manifest in the types, then a check is inserted. Note that regular strings translate to strings directly; there is no space overhead.

The translation correctness theorem guarantees that the translation is type preserving and that semantics of the original code and translation coincide, following the treatment of compilation pioneered by the TIL compiler for SML [15]. Note that we apply the translation inline in judgements for concision when convenient.

Theorem 15 (Translation Correctness)**.** *If $\Theta \vdash e : \sigma$ then there exists an ι such that $[\![e]\!] = \iota$ and $[\![\Theta]\!] \vdash \iota : [\![\sigma]\!]$. Furthermore, $e \Downarrow v$ and $\iota \Downarrow \dot{v}$ such that $[\![v]\!] = \dot{v}$.*

$\boxed{\iota \Downarrow \dot{v}}$

$$\text{P-E-Abs}\ \frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \text{P-E-App}\ \frac{\iota_1 \Downarrow \lambda x.\iota_3 \qquad \iota_2 \Downarrow \dot{v}_2 \qquad [\dot{v}_2/x]\iota_3 \Downarrow \dot{v}_3}{\iota_1(\iota_2) \Downarrow \dot{v}_3}$$

$$\text{P-E-Str}\ \frac{}{\mathsf{str}[s] \Downarrow \mathsf{str}[s]} \qquad \text{P-E-Rx}\ \frac{}{\mathsf{rx}[r] \Downarrow \mathsf{rx}[r]} \qquad \text{P-E-Concat}\ \frac{\iota_1 \Downarrow \mathsf{str}[s_1] \qquad \iota_2 \Downarrow \mathsf{str}[s_2]}{\mathsf{concat}(\iota_1; \iota_2) \Downarrow \mathsf{str}[s_1 s_2]}$$

$$\text{P-E-Case-}\epsilon\ \frac{\iota_1 \Downarrow \mathsf{str}[\epsilon] \qquad \iota_2 \Downarrow \dot{v}_2}{\mathsf{strcase}(\iota_1; \iota_2; x, y.\iota_3) \Downarrow \dot{v}_2}$$

$$\text{P-E-Case-Concat}\ \frac{\iota_1 \Downarrow \mathsf{str}[as] \qquad [\mathsf{str}[a], \mathsf{str}[s]/x, y]\iota_3 \Downarrow \dot{v}}{\mathsf{strcase}(\iota_1; \iota_2; x, y.\iota_3) \Downarrow \dot{v}}$$

$$\text{P-E-Replace}\ \frac{\iota_1 \Downarrow \mathsf{rx}[r] \qquad \iota_2 \Downarrow \mathsf{str}[s_2] \qquad \iota_3 \Downarrow \mathsf{str}[s_3] \qquad \mathsf{subst}(r; s_2; s_3) = s}{\mathsf{replace}(\iota_1; \iota_2; \iota_3) \Downarrow \mathsf{str}[s]}$$

$$\text{P-E-Check-OK}\ \frac{\iota_r \Downarrow \mathsf{rx}[r] \qquad \iota \Downarrow \mathsf{str}[s] \qquad s \in \mathcal{L}\{r\} \qquad \iota_1 \Downarrow \dot{v}_1}{\mathsf{check}(\iota_r; \iota; \iota_1; \iota_2) \Downarrow \dot{v}_1}$$

$$\text{P-E-Check-NotOK}\ \frac{\iota_r \Downarrow \mathsf{rx}[r] \qquad \iota \Downarrow \mathsf{str}[s] \qquad s \notin \mathcal{L}\{r\} \qquad \iota_2 \Downarrow \dot{v}_2}{\mathsf{check}(\iota_r; \iota; \iota_1; \iota_2) \Downarrow \dot{v}_2}$$

Figure 7: Big step semantics for λ_P

.

Proof Sketch. The proof proceeds by induction on the typing relation for e. We choose an ι based on the syntactic form in λ_P corresponding to the form under consideration (e.g. we choose replace when considering sreplace). The proof proceeds by our type safety theorems and an appeal to the induction hypothesis. □

2.3.1 Preservation of Security

Finally, our main result establishes that correctness of λ_{RS} is preserved under the translation into λ_P.

Theorem 16 (Correctness of Input Sanitation for Translated Terms)**.** *If $[\![e]\!] = \iota$ and $\emptyset \vdash e : \mathsf{stringin}[r]$ then $\iota \Downarrow \mathsf{str}[s]$ for $s \in \mathcal{L}\{r\}$.*

Proof. By Theorem 15 and the rules given, $\iota \Downarrow \mathsf{str}[s]$ implies that $e \Downarrow \mathsf{rstr}[s]$. Theorem 12 together with the assumption that e is well-typed implies that $s \in \mathcal{L}\{r\}$. □

3. IMPLEMENTATION IN ATLANG

In the previous section, we specified a type system and a translation semantics to a language containing only strings and regular expressions. In this section, we take Python to be such a target language. Python does not have a static type system, however, so to implement these semantics, we will use `atlang`, an extensible type system for Python (being developed by the authors). By using `atlang`, which leverages Python's quotations and reflection facilities, we can implement these semantics as a library, rather than as a new dialect of the language.

$\boxed{[\![\sigma]\!] = \tau}$

$$\textsc{Tr-T-String}\ \frac{}{[\![\mathsf{stringin}[r]]\!] = \mathsf{string}} \qquad \textsc{Tr-T-Arrow}\ \frac{[\![\sigma_1]\!] = \tau_1 \qquad [\![\sigma_2]\!] = \tau_2}{[\![\sigma_1 \to \sigma_2]\!] = \tau_1 \to \tau_2}$$

$\boxed{[\![\Psi]\!] = \Theta}$

$$\textsc{Tr-T-Context-Emp}\ \frac{}{[\![\emptyset]\!] = \emptyset} \qquad \textsc{Tr-T-Context-Ext}\ \frac{[\![\Psi]\!] = \Theta \qquad [\![\sigma]\!] = \tau}{[\![\Psi, x : \sigma]\!] = \Theta, x : \tau}$$

$\boxed{[\![e]\!] = \iota}$

$$\textsc{Tr-Var}\ \frac{}{[\![x]\!] = x} \qquad \textsc{Tr-Abs}\ \frac{[\![e]\!] = \iota}{[\![\lambda x.e]\!] = \lambda x.\iota} \qquad \textsc{Tr-App}\ \frac{[\![e_1]\!] = \iota_1 \qquad [\![e_2]\!] = \iota_2}{[\![e_1(e_2)]\!] = \iota_1(\iota_2)}$$

$$\textsc{Tr-string}\ \frac{}{[\![\mathsf{rstr}[s]]\!] = \mathsf{str}[s]} \qquad \textsc{Tr-Concat}\ \frac{[\![e_1]\!] = \iota_1 \qquad [\![e_2]\!] = \iota_2}{[\![\mathsf{rconcat}(e_1; e_2)]\!] = \mathsf{concat}(\iota_1; \iota_2)}$$

$$\textsc{Tr-Case}\ \frac{[\![e_1]\!] = \iota_1 \qquad [\![e_2]\!] = \iota_2 \qquad [\![e_3]\!] = \iota_3}{[\![\mathsf{rstrcase}(e_1; e_2; x, y.e_3)]\!] = \mathsf{strcase}(\iota_1; \iota_2; x, y.\iota_3)}$$

$$\textsc{Tr-Subst}\ \frac{[\![e_1]\!] = \iota_1 \qquad [\![e_2]\!] = \iota_2}{[\![\mathsf{rreplace}[r](e_1; e_2)]\!] = \mathsf{replace}(\mathsf{rx}[r]; \iota_1; \iota_2)} \qquad \textsc{Tr-SafeCoerce}\ \frac{[\![e]\!] = \iota}{[\![\mathsf{rcoerce}[r']\!](e)]\!] = \iota}$$

$$\textsc{Tr-Check}\ \frac{[\![e]\!] = \iota \qquad [\![e_1]\!] = \iota_1 \qquad [\![e_2]\!] = \iota_2}{[\![\mathsf{rcheck}[r](e; x.e_1; e_2)]\!] = \mathsf{check}(\mathsf{rx}[r]; \iota; (\lambda x.\iota_1)(\iota); \iota_2)}$$

Figure 8: Translation from source to target.

3.1 Example Usage

Figure 9 demonstrates the use of two type constructors, `fn` and `stringin`, corresponding to the two type constructors of λ_{RS}. We show these as being imported from `atlib`, the standard library for `atlang` (it benefits from no special support from the language itself).

The `fn` type constructor can be used to annotate functions that should be statically checked by `atlang`.[1] The function `sanitize` on lines 3-7, for example, specifies one argument, `s`, of type `stringin[r'.*']`. Here, the index is a regular expression, written using Python's *raw string notation* as is conventional (in this particular instance, the `r` is not strictly necessary). The sanitize function takes an arbitrary string and returns a string without double quotes or left and right brackets. Note that the return type need not be specified: `atlang` uses a form of *local type inference* [13].

In this example, we use an HTML encoding so that the same sanitization function can be used to generate both SQL commands and HTML securely. The sanitized string is generated by invoking the `replace` operator, which has the same semantics as rreplace in λ_{RS}. Unlike in the core calculus, it is invoked like a method on `s`. The regular expression determining the substrings to be replaced is given as the first argument (as in λ_{RS}, the only restriction here is that the regular expression must be specified statically.)

[1]Here, we use argument annotation syntax only available in versions 3.0+ of Python. Alternative syntax supporting Python 2.7+ can also be used, but we omit it here.

```
1  from atlib import fn, stringin
2
3  @fn
4  def sanitize(s : stringin[r'.*']):
5    return (s.replace(r'"', '&quot;')
6            .replace(r'<', '&lt;')
7            .replace(r'>', '&gt;'))
8
9  @fn
10 def results_query(s : stringin[r'[^"]*']):
11   return 'SELECT * FROM users WHERE name="' + s + '"'
12
13 @fn
14 def results_div(s : stringin[r'[^<>]*']):
15   return '<div>Results for ' + s + '</div>'
16
17 @fn
18 def main():
19   input = sanitize(user_input())
20   results = db_execute(results_query(input))
21   return results_div(input) + format(results)
```

Figure 9: Regular string types in atlang, a library that enables static type checking for Python.

```
1  import re
2
3  def sanitize(s):
4     return re.sub(r'"', re.sub(r'<', re.sub(r'>',
5       s, '&gt;'), '&lt;'), '&quot;')
6
7  def results_query(s):
8    return 'SELECT * FROM users WHERE name="' + s + '"'
9
10 def results_div(s):
11   return '<div>Results for ' + s + '</div>'
12
13 def main():
14   input = sanitize(user_input())
15   results = db_execute(results_query(input))
16   return results_div(input) + format(results)
```

Figure 10: The output of compilation of Figure 9 (at the terminal, `atc figure9.py`, or just-in-time).

The functions `results_query` and `results_div` construct a SQL query and an HTML snippet, respectively, by regular string concatenation. The argument type annotations serve as a check that sanitation was properly performed. In the case of results_query, this specification ensures that user input cannot prematurely be terminated. In the case of results_div, this specification ensures that user input does not contain any HTML tags, which is a conservative but effective policy for preventing XSS attacks. Note that the type of the surrounding string literals are determined by the type constructor of the function they appear in, `fn` in both cases, which we assume simply chooses `stringin[r'.*']` (an alternative strategy would be to use the most specific type for the literal, rather than the most general, but this choice is immaterial for this example). The addition operator here corresponds to the rconcat operator in λ_{RS}.

The `main` function invokes the functions just described. It begins by passing user input to `sanitize`, then executing a database query and returning HTML based on this sanitized input. The helper functions `user_input` and `db_execute` are not shown but can be assumed standard. Importantly, had we mistakenly forgotten to call `sanitize`, the function would not type check (in this case, it is obvious that we did, but lines 19 and 20 would in practice be separated more drastically in the code). Moreover, had `sanitize` itself not been implemented correctly (e.g. we forgot to strip out quotation marks), then `main` would also not typecheck either.

```
class stringin(atlang.Type):
  def __init__(self, rx):
    atlang.Type.__init__(idx=rx)

  def ana_Str(self, ctx, node):
    if not in_lang(node.s, self.idx):
      raise atlang.TypeError("...", node)

  def trans_Str(self, ctx, node):
    return astx.copy(node)

  def syn_BinOp_Add(self, ctx, node):
    left_t = ctx.syn(node.left)
    right_t = ctx.syn(node.right)
    if isinstance(left_t, stringin):
      left_rx = left_t.idx
      if isinstance(right_t, stringin):
        right_rx = right_t.idx
        return stringin[lconcat(left_rx, right_rx)]
    raise atlang.TypeError("...", node)

  def trans_BinOp_Add(self, ctx, node):
    return astx.copy(node)

  def syn_Method_replace(self, ctx, node):
    [rx, exp] = node.args
    if not isinstance(rx, ast.Str):
      raise atlang.TypeError("...", node)
    rx = rx.s
    exp_t = ctx.syn(exp)
    if not isinstance(exp_t, stringin):
      raise atlang.TypeError("...", node)
    exp_rx = exp_t.idx
    return stringin[lreplace(self.idx, rx, exp_rx)]

  def trans_Method_replace(self, ctx, node):
    return astx.quote(
      """__import__(re); re.sub(%0, %1, %2)""",
      astx.Str(s=node.args[0]),
      astx.copy(node.func.value),
      astx.copy(node.args[1]))

  # check and strcase omitted

  def check_Coerce(self, ctx, node, other_t):
    # coercions can only be defined between
    # types with the same type constructor,
    if rx_sublang(other_t.idx, self.idx):
      return other_t
    else: raise atlang.TypeError("...", node)
```

Figure 11: Implementation of the `stringin` type constructor in atlang.

One somewhat subtle issue here is that the return type of `sanitize` is equivalent to `stringin[r'[^"<>]*']`, which is a distinct type from the argument types to `results_query` and `results_div`. More specifically, however, it is a "smaller" type, in that it could be coerced to these argument types using an operator like rcoerce in λ_{RS}. In our implementation, safe coercions are performed implicitly rather than explicitly. Because all regular strings are implemented as strings, this coercion induces no run-time change in representation.

Figure 10 shows the output of typechecking and translating this code (this can occur either in batch mode at the terminal, generating a new file, or "just-in-time" at the first callsite of each function in the dynamically typed portion of the program, not shown).

3.2 Implementation

The primary unit of extension in `atlang` is the *active type constructor*, rather than the abstract syntax as in other work on language extensibility. This allows us to implement the entire system as a library in Python and avoid needing to develop new tooling, and also makes it more difficult to create ambiguities between extensions. Active type constructors are subclasses of `atlang.Type`, and types are instances of these classes. The methods of active type constructors control how typechecking and translation proceed for associated operations. In Figure 11, we show key portions of the implementation of the `stringin` type used in the example above. Although a detailed description of the extension mechanism is beyond the scope of this work, we describe the intuitions behind the various methods below.

The constructor, `__init__` in Python, is called when a type is constructed. It simply stores the provided regular expression as the type index by calling the superclass.

When a string literal is being checked against a regular string type, the method `ana_Str` is called. It checks that the string is in the language of the regular expression provided as the type index, corresponding to rule S-T-StringIn-I in Section 2. The method `trans_Str` is called after typechecking to produce a translation. Here, we just copy the original string literal – regular strings are implemented as strings.

The method `syn_BinOp_Add` synthesizes a type for the string concatenation operation if both arguments are regular strings, consistent with rule S-T-Concat. The corresponding method `trans_BinOp_Add` again simply translates the operation directly to string concatenation, consistent with our translation semantics.

The method `syn_Method_replace` synthesizes a type for the "method-like" operator `replace`, seen used in our example. It ensures that the first argument is a statically known string, using Python's built-in `ast` module, and leverages an implementation of `lreplace`, which computes the appropriate regular expression for the string following replacement, again consistent with our description in Section 2. Translation proceeds by using the `re` library built into Python, as can be seen in Figure 10.

Code for checked conversions and string decomposition is omitted, but is again consistent with our specification in the previous section. Safe coercions are controlled by the `check_Coerce` function, which checks for a sublanguage relationship. Here, as in the other methods, failure is indicated by raising an `atlang.TypeError` with an appropriate error message and location.

Taken together, we see that the mechanics of extending `atlang` with a new type constructor are fairly straightforward: we determine which syntactic forms the operators we specified in our core calculus should correspond to, then directly implement a decision procedure for type synthesis (or, in the case of literal forms, type analysis) in Python. The `atlang` compiler invokes this logic when working with regular strings. The result is a library-based embedding of our semantics into Python.

4. RELATED WORK

The input sanitation problem is well-studied. There exist a large number of techniques and technologies, proposed by both practitioners and researchers, for preventing injection-style attacks. In this section, we explain how our approach to the input sanitation problem differs from each of these approaches. Equally important, however, is our more general assertion that language extensibility is the right approach for consideration of language-oriented security mechanisms like the one we described here.

Unlike frameworks and libraries provided by languages such as Haskell and Ruby, our type system provides a *static*

guarantee that input is always properly sanitized before use. Doing so requires reasoning about the operations on regular languages corresponding to standard operations on strings; we are unaware of any production system which contains this form of reasoning. Therefore, even where frameworks and libraries provide a viable interface or wrapper around input sanitation (e.g. prepared SQL statements), our approach is complementary because it can be used to ensure the correctness of that framework or library itself. Furthermore, our approach is more general than database abstraction layers because our mechanism is applicable to all forms of command injection (e.g. shell injection or remote file inclusion).

A number of research languages provide static guarantees that a program is free of input sanitation vulnerabilities by never using a string representation at all, but rather desugaring SQL syntax to a safe representation immediately [3]. The Wyvern programming language introduced a general framework for writing syntax extensions like this [10]. Unlike this work, our solution to the input sanitation problem retains a string representation and thus has a very low barrier to adoption. Our implementation conservatively extends Python – a popular language among web developers – rather than requiring the adoption of a new language entirely. We also believe our semantic extensibility approach (as opposed to only syntactic extensibility in Wyvern) is better-positioned for security, where continuously evolving threats might require frequent addition of new semantic analyses. The `atlang` system is particularly well-suited to type system based analyses.

Incorporating regular expressions into the type system is not novel. The XDuce system [9] checks XML documents against schema using regular expressions. Similarly, XHaskell [14] focuses on XML documents. We differ from this and related work in at least three ways:

- Although our static replacement operation is definable in some languages with regular expression types, we are the first to expose this operation and connect the semantics of regular language replacement with the semantics of string substitution via a type safety and compilation correctness argument.
- The underlying representation of regular strings is guaranteed to be a string, rather than a heavier implementation based on an encoding using functional datatypes.
- We demonstrate that regular expression types are applicable to the web security domain, whereas previous work on regular expression types focused on XML.
- We work within an extensible type system.

5. FUTURE WORK

We believe that this type system extension serves as a useful basis for web-oriented static analysis; frameworks and regular expression libraries could be annotated, along with use-sites. We hope to empirically evaluate the feasability of this approach in the future using `atlang`. We also believe that extensible programming languages are a promising approach toward incorporating other security analyses into programming languages. Construing such analyses as type systems, specifying them rigorously and implementing them within an extensible type system appears to be a promising general technique that the community may wish to emulate.

6. ACKNOWLEDGEMENTS

This work was supported by supported by the National Security Agency lablet contract #H98230-14-C-0140.

7. REFERENCES

[1] D. Ancona. How to prove type soundness of Java-like languages without forgoing big-step semantics. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs*, FTfJP'14, pages 1:1–1:6, New York, NY, USA, 2014. ACM.

[2] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, Oct. 1964.

[3] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI'10*, Oct. 2010.

[4] N. Fulton. Security through extensible type systems. SPLASH '12, pages 107–108, New York, NY, USA, 2012. ACM.

[5] N. Fulton. A typed lambda calculus for input sanitation. Undergraduate thesis in mathematics, Carthage College, 2013.

[6] N. Fulton, C. Omar, and J. Aldrich. Statically typed string sanitation inside a python. Technical Report CMU-ISR-14-112, Carnegie Mellon University.

[7] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[8] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[9] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. In *ICFP '00*, 2000.

[10] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP 2014*, volume 8586 of *Lecture Notes in Computer Science*, pages 105–130. Springer Berlin Heidelberg, 2014.

[11] OWASP. Open web application security project top 10. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[12] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, Mar. 2009.

[13] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.

[14] M. Sulzmann and K. Lu. XHaskell – adding regular expression types to Haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 75–92. Springer Berlin Heidelberg, 2008.

[15] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *PLDI '96*, Philadelphia, PA, May 1996.