A Type System for String Sanitation Implemented Inside a Python

Nathan Fulton Cyrus Omar Jonathan Aldrich

School of Computer Science
Carnegie Mellon University
{nfulton, comar, aldrich}@cs.cmu.edu

Abstract

ABSTRACT HERE

1. Introduction

In web applications and other settings, incorrect input sanitation often causes security vulnerabilities. For this reason, frameworks often provide methods for sanitizing user input. When these methods are insufficient or unavailable, developers often create custom input sanitation algorithms. In both cases, input sanitation techniques are ultimately implemented using the language's regex library.

Formally, input sanitation is the problem of ensuring that an arbitrary string is converted into a safe form before potentially unsafe use. For example, consider SQL injection attacks. To prevent such attacks, we might ensure that any string used as input to a query does not contain unescaped SQL command sequences.

This appendix presents a type system, λ_{CS}, for ensuring that input sanitation algorithms are implemented correctly with respect to use site specifications.

Unlike frameworks provided by general purpose languages such as Haskell and Ruby, our type system provides a *static* guarantee that input is always properly sanitized before use. We achieve this by defining a typing relation which captures idiomatic sanitation algorithms. Type safety relies upon several closure and decidability results about regular languages.

XDuce typechecks XML. Like our work, XDuce relies upon some properties of regular expressions to establish soundness and completeness results. Unlike XDuce, our work is motivated by input sanitation and therefore considers arbitrary strings (as opposed to tree-structured XML documents). Furthermore, our static treatment of the ubiquitous regex filter (replace) function for regular expression is novel.

Finally, some research languages achieve similar safety guarantees via a specialized type system. The existence of such domain-specific type systems is suggestive, and we believe the simplicity of λ_{CS} demonstrates the effectiveness of extensibility. Instead of introducing entirely new languages for each domain, language extension developers may invest incrementally in obtaining static guarnatees which address common mistakes.

An outline of this paper follows:

TODO-nrf real outline!

- In §2, we define the type system's static and dynamic semantics.
- Section 3 recalls some classical results about regular expressions and presents a type safety proof for λ_{CS} based upon these properties.
- Finally, §4 discusses our implemention of λ_{CS} as a type system extension within the Ace programming language.

2. A Type System for String Sanitation

The λ_{CS} language is characterized by a type of strings indexed by regular expressions, together with operations on such strings which correspond to common input sanitation patterns.

This section presents the grammar and semantics of λ_{CS}. The semantics are defined in terms of an internal language with at least strings and a regex filter function. These constraints are captured by the internal term valuations (`ival`). The internal language does not necessarily need a regex filter function because any dynamic conversion is easily definable using a combination of filters and safe casting.

The λ_{CS} language gives static semantics for common regular expression library functions. In this treatment, we include concatenation and filtering. The `filter` function removes all instances of a regular expression in a string, while concatenation ($+$) concatenates two strings.

$\langle r \rangle ::= \epsilon \mid . \mid a \mid r \cdot r \mid r + r \mid r*$	Regex ($a \in \Sigma$).
$\langle t \rangle ::=$	terms:
$\quad \texttt{filter}(\texttt{string\_in}[r], t)$	filter substrings
$\mid \; [\texttt{string\_in}[r]](t)$	safe conversion
$\mid \; \texttt{dconvert}(\texttt{string\_in}[r], t)$	unsafe conversion
$\langle iv \rangle ::=$	internal values:
$\quad$ ‘s’	internal string
$\mid \; \texttt{ifilter}(r, \texttt{istring}(s))$	internal filter
$\langle v \rangle ::=$	values:
$\quad$ “s”	string
$\langle T \rangle ::=$	types:
$\quad \texttt{string}$	Strings
$\mid \; \texttt{string\_in}[r]$	Regular language strings
$\langle \Gamma \rangle ::= \emptyset \mid \Gamma, x : T$	typing context

Figure 1. Syntax of λ_{CS}

2.1 Typing

The string_in[r] type is parameterized by regular expressions; if e : string_in[r], then $e \in r$. Mapping from an arbitrary string to a string_in[r] requires defining an algorithm – in terms of filter – for converting a string_in[$.*$] into a string_in[r]. The static semantics of the language defines the types of operations on regular expressions in terms of well-understood properties about regular lanugages; we recall these properties in section 3.

2.2 Dynamics

There are two evaluation judgements: $e : T \Rightarrow e'$ and $e : T \rightsquigarrow i$. The $\Rightarrow$ relation is between λ_{CS} expressions, while the $\rightsquigarrow$ relation is a mapping from λ_{CS} expressions into internal language expressions i such that i ival.

Safety of the evaluation relation depends upon an injective mapping from λ_{CS} types info internal language types. This relation, h, is defined below.

2.3 Type Safety

The type safety proof relies upon some assumptions about the type system and dynamics of the internal language, as well as some properties of regular languages.

There must exist a translation from λ_{CS} types to the types of the internal language. For the remainder of this paper, we call the type translation function h.

Definition 1 (Type Translation Function h)**.** The type translation function $h : Type \rightarrow IType$ is defined as follows:

- $\forall r.h(\texttt{string\_in}[r]) = \texttt{istring}$
- $h(\texttt{string}) = \texttt{istring}$

Additionally, we assume that the internal language contains an implementation of strings, together with operations for concatenation and filtering by regular expression.

Definition 2 (Types of internal values)**.** Let ‘s’ range over string literals and r over regular expressions. Internal values are typed as follows:

- If $e =$ ‘s’ then $e : \texttt{istring}$.
- If $e = \texttt{ifilter}(r,$ ‘s’$)$ then $e : \texttt{istring}$.
- If $e =$ ‘s_1’ $+$ ‘s_2’ then $e : \texttt{istring}$.

For simplicity, we assume a fixed translation from λ_{CS} regular expressions to regular expressions recognizable by the internal language’s regex library (in practice, a fixed translation is acceptable.) To summarize, we assume an internal language containing a string type together with operations for string concatentation and filtering. We expect closure over strings for both operations. Finally, recall that ifilter is only needed for dynamic casts, which may be removed without descreasing the expressivity or even usability of the language. Finally, the semantics of the filter function are defined in terms of rl_filter, which is a static version of ifilter.

2.4 Properties of Regular Languages

The regular languages are the smallest set generated by regular expressions defined in Figure 1.

Theorem 3. *Closure Properties. The regular expressions are closed under complements and concatenation.*

Proof. See [**?**]. □

Theorem 4. *Coercion Theorem. Suppose that R and L are regular expressions, and that $s \in R$ is a finite string. Let $s' := coerce(R, L, s)$ with all maximal substrings recognized by L replaced with ϵ. Then s' is recognized by $(R \backslash L) + \emptyset$ and the construction of $R \backslash L$ is decidable.*

Proof. Let F, G be FAs corresponding to R and L, and let G' be G with its final states inverted (so that G' is the complement of L). Define an FA H as a DFA corresponding to the NFA found by combining F and G' such that H accepts only if R and L' accept or if s is empty (this construction may result in an exponential blowup in state size.) Clearly, H corresponds to $R \backslash L + \emptyset$. Thus, the construction of $R \backslash L + \emptyset$ is decidable.

If $R \subset L$, $s' = \emptyset$. If $L \subset R$, either $s' = \emptyset$, or $s' \in R$ and $s' \notin L$. If R and L are not subsets of one another, then it may be the case that L recognizes part of R. Consider L as the union of two languages, one which is a subset of R and one which is disjoint. The subset language is considered above and the disjoint language is inconsequential. □

$$\frac{}{\texttt{string} \equiv \texttt{string\_in}[.*]} \qquad \textsc{T-Equiv-Top}$$

$$\frac{\mathcal{L}\{r\} = \mathcal{L}\{r'\}}{\texttt{string\_in}[r] \equiv \texttt{string\_in}[r']} \qquad \textsc{T-Equiv-string\_in}$$

$$\frac{e \in \mathcal{L}\{r\}}{\Gamma \vdash e : \texttt{string\_in}[r]} \qquad \textsc{T-Equiv-Include}$$

$$\frac{\Gamma \vdash e : \texttt{string}}{\Gamma \vdash e : \texttt{string\_in}[e]} \qquad \textsc{T-Minimum}$$

$$\frac{\Gamma \vdash e_1 : \texttt{string\_in}[r_1] \qquad \Gamma \vdash e_2 : \texttt{string\_in}[r_2]}{\Gamma \vdash e_1 + e_2 : \texttt{string\_in}[r_1 \cdot r_2]} \qquad \textsc{T-Concat}$$

$$\frac{\Gamma \vdash e : \texttt{string\_in}[r] \qquad \mathcal{L}\{r'\} \subseteq \mathcal{L}\{r\}}{\Gamma \vdash [\texttt{string\_in}[r']](e) : \texttt{string\_in}[r']} \qquad \textsc{T-Convert}$$

$$\frac{\Gamma \vdash e : \texttt{string\_in}[r]}{\Gamma \vdash \texttt{dconvert}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r']} \qquad \textsc{T-Dconvert (optional)}$$

$$\frac{\Gamma \vdash t : \texttt{string\_in}[r]}{\Gamma \vdash \texttt{filter}(\texttt{string\_in}[r'], t) : \texttt{string\_in}[(r \backslash r') + \emptyset]} \qquad \textsc{T-Filter}$$

Figure 2. Typing relation for λ_{CS}

$$\frac{}{\text{“}s\text{”} : \texttt{string\_in}[r]\ \texttt{val}}$$ E-STRINVAL

$$\frac{}{\text{‘}e\text{’} : \texttt{istring}\ \texttt{ival}}$$ E-IVAL

$$\frac{e\texttt{ival}}{\texttt{ifilter}(r, e)\ \texttt{ival}}$$ E-IFILTERVAL

$$\frac{e\ \texttt{val}}{e : \texttt{string} \rightsquigarrow \text{‘}e\text{’}}$$ E-STRING

$$\frac{e\ \texttt{val}}{e : \texttt{string\_in}[r] \rightsquigarrow \text{‘}e\text{’}}$$ E-STRING_IN

$$\frac{e_1\ \texttt{val} \qquad e_2\ \texttt{val}}{e_1 + e_2 : \texttt{string\_in}[r_1 + r_2] \rightsquigarrow \text{‘}e_1\text{’} + \text{‘}e_2\text{’}}$$ E-CONCATVAL

$$\frac{e_1 : \texttt{string\_in}[r_1] \Rightarrow {e_1}'}{e_1 + e_2 : \texttt{string\_in}[r_1 + r_2] \Rightarrow {e_1}' + e_2}$$ E-CONCATL

$$\frac{e_2 : \texttt{string\_in}[r_2] \Rightarrow {e_2}'}{e_1 + e_2 : \texttt{string\_in}[r_1 + r_2] \Rightarrow e_1 + {e_2}'}$$ E-CONCATR

$$\frac{e\ \texttt{val}}{\texttt{filter}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r \backslash r' + \emptyset] \Rightarrow \texttt{rl\_filter}(r, e)}$$ E-FILTERVAL

$$\frac{e : \texttt{string\_in}[r] \Rightarrow e'}{\texttt{filter}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r \backslash r' + \emptyset] \Rightarrow \texttt{filter}(\texttt{string\_in}[r], e')}$$ E-FILTER

$$\frac{e\ \texttt{val}}{[\texttt{string\_in}[r']](e) : \texttt{string\_in}[r'] \rightsquigarrow \text{‘}e\text{’}}$$ E-CONVERTVAL

$$\frac{e : \texttt{string\_in}[r] \Rightarrow e'}{[\texttt{string\_in}[r']](e) : \texttt{string\_in}[r'] \Rightarrow [\texttt{string\_in}[r']](e')}$$ E-CONVERT

$$\frac{e : \texttt{string\_in}[r] \Rightarrow e'}{\texttt{dconvert}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r'] \Rightarrow \texttt{dconvert}(\texttt{string\_in}[r'], e')}$$ E-DCONVERTVAL (OPTIONAL)

$$\frac{e\ \texttt{val}}{\texttt{dconvert}(\texttt{string\_in}[r'], e) : \texttt{string\_in}[r'] \rightsquigarrow \texttt{ifilter}(r', \text{‘}e\text{’})}$$ E-DCONVERT (OPTIONAL)

Figure 3. SOS rules for λ_{CS}

2.5 Type Safety Proof

Theorem 5 (Preservation)**.** *Let T be a type in λ_{CS} and $h(T) = \sigma$ the corresponding type in the internal language. For all terms e:*

- *If $e : T$ and $e : T \rightsquigarrow i$ then $i : \sigma$ such that $h(T) = \sigma$.*
- *If $e : T$ and $e : T \Rightarrow e'$ then $e' : T$.*

Proof. The proof is a straightforward induction on the derivation of the combined evaluation relation.

E-Ival, E-Ifilterval. Both cases hold since the terms at hand are not λ_{CS} terms.

E-Stringval, E-strval. Both cases hold since no reduction is possible.

E-String. By the definition of typing for internal terms, ‘e’ : $\mathtt{istring}$. It suffices to show that $h(\mathtt{string}) = \mathtt{istring}$, which follows from the definition of h.

E-String_in. By the definition of typing for internal terms, ‘e’ : $\mathtt{istring}$. It suffices to show that $h(\mathtt{string\_in}[r]) = \mathtt{istring}$, which follows from the definition of h for arbitrary r.

E-concatval. By the definition of typing for internal terms, ‘e_1’ + ‘e_2’ : $\mathtt{istring}$. So it suffices to show that $h(\mathtt{string\_in}[r_1{+}r_2]) = \mathtt{istring}$, which follows from the definition of h for arbitrary r_1, r_2.

E-concatR, E-concatL. Consider E-concatR. By induction, $e_1' : \mathtt{string\_in} l_1$. By inversion of T-Concat at the premise, $e_2 : \mathtt{string\_in} r_2$. Therefore, $e_1 + e_2 : \mathtt{string\_in}[r_1 + r_2]$. The left rule is symmetric.

E-Filterval. We have that $\mathtt{filter}(\mathtt{string\_in}[r'], e) : \mathtt{string\_in}[r \backslash r' + \emptyset]$. By inversion of T-Filter, $e : \mathtt{string\_in}[r]$. By T-Equiv-string_in (which is bidirectional), $e \in \mathcal{L}\{r\}$. By the Coercion Theorem, $\mathtt{rl\_filter}(r, e) \in \mathcal{L}\{r \backslash r' + \emptyset\}$. By T-Equiv-string_in, $e \in \mathcal{L}\{r\}$ and $\mathtt{rl\_filter}(r, e) \in \mathcal{L}\{r \backslash r' {+} \emptyset\}$ implies $\mathtt{rl\_filter}(r, e) : \mathtt{string\_in}[r \backslash r' + \emptyset]$.

E-Filter. By inversion, $e : \mathtt{string\_in}[r'] \Rightarrow e'$ so by the induction $e' : \mathtt{string\_in}[r']$. Therefore, $\mathtt{filter}(\mathtt{string\_in}[r], e') : \mathtt{string\_in}[r \backslash r' + \emptyset]$ by T-Filter.

E-Convertval. It suffices to show that $h(\mathtt{string\_in}[r]) = \mathtt{istring}$, which is true by definition.

E-Convert. By inversion and induction, $e' : \mathtt{string\_in}[r]$. We know that $[\mathtt{string\_in}[r']](e) : \mathtt{string\_in}[r']$, so by inversion of T-Convert $\mathcal{L}\{r'\} \subseteq \mathcal{L}\{r\}$. It follows that $[\mathtt{string\_in}[r']](e') : \mathtt{string\_in}[r']$.

E-DConvert. By inversion, $e : \mathtt{string\_in}[r'] \Rightarrow e'$. By the induction hypothesis, $e' : \mathtt{string\_in}[r']$; therefore, by T-Dconvert, $\mathtt{dconvert}([, \mathtt{string\_in})[r]](e) : \mathtt{string\_in}[r]$.

E-DConvertval. By the definition of typing for internal terms, $\mathtt{ifilter}((, r), \text{‘}e\text{’}) : \mathtt{istring}$. It suffices to show that $h(\mathtt{string\_in}[r_1 + r_2]) = \mathtt{istring}$, which follows from the definition of h.

□

Theorem 6 (Progress)**.** *If $e : T$ then $e : T \Rightarrow^* e' \rightsquigarrow^* i$ where i $\mathtt{ival}$.*

Proof. By induction on the derivation of $e : T$. For **T-Equiv-Include**, note that $e \in \mathcal{L}\{r\}$, $e =$ “s” for some s; therefore, e $\mathtt{val}$ by E-Strinval. We have that e $\mathtt{val}$ and $e : \mathtt{string\_in}[r]$, so $e \rightsquigarrow$ ‘e’ by E-string_in. The remaining cases follow by induction on the hypotheses and application of corresponding evaluation rules. □

3. Implementation Inside a Python

TODO-nrf rewrite The λ_{CS} language is implemented as a type system extension in Ace; this extension is illustrated in the examples at the beginning of this paper.

Computing a regular expression representating the language $R \backslash S$ is necessary in order to type-check expressions in which the filter function occurs. This language is computed by translating R and S into finite automata, complementing the final states of S, then constructing the cross-product of R and S'. Type checking terminates only because this construction is decidable.

In addition to this construction, other more typical operations – such as equality checks for regular expressions and regular expression matching – are also necessary. For these reasons, the Ace extension implementing λ_{CS} includes a library implementing each of these constructions.

3.1 Background: Ace

TODO: Make a TR out of the OOPSLA submission.

3.2 Explicit Conversions

3.3 Adding Subtyping to Ace

3.4 Theory

4. Related Work

5. Discussion