TODO-nrf spell check.

TODO-nrf Include functions (app, abs)?

TODO-nrf Change icheck in eval rule to Python.

TODO-nrf Is it ok to make the proofs single-column? If not clean up overflows.

TODO-nrf Make sure failure of unicity isn't causing any problems and write down lemmas about forms (e.g. $e \in \mathcal{L}\{r\} \implies e =$ “s” etc).

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 write develop 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 common sequences.

This appendix presents a type system called 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 moving portions of the regular expression library into the type system. Therefore, type safety relies upon several closure and decidability results about regular languages.

Although Ur/Web achieves a similar safety guarantee, its type system is significantly more complicated than the system defined below. We believe this simplicity demonstrates the power of extensibility. Instead of introducing new abstractions based upon subtle analyses and requiring the adoption of new programming languages, extension designers may capture natural programming idioms directly. TODO-nrf this seems unsubstantiated or dangerously vague.

Finally, XDuce typechecks XML. Like our work, XDuce relies upon some properties of regular expression to establish soundness and copmleteness 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 “filter” function for regular expression is novel.

An outline of this appendix follows:

- In §2, we define the type system's static and dynamic semantics.

$\langle r \rangle ::= \epsilon \mid . \mid a \mid r \cdot r \mid r + r \mid r*$	Regex ($a \in \Sigma$).
$\langle t \rangle ::=$	terms:
$\texttt{filter}(\texttt{string\_in}[r], t)$	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:
‘s’	internal string
$\mid \texttt{ifilter}(r, \texttt{istring}(s))$	internal filter
$\langle v \rangle ::=$	values:
“s”	string
$\langle T \rangle ::=$	types:
$\texttt{string}$	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}

- 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. Definition of λ_{CS}

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.

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

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

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

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

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

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

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

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

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

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

E-CONCAT

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

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

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

E-CONVERTVAL

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

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) [1]

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

E-DCONVERT (OPTIONAL) [2]

Figure 3. Operational semantics for λ_{CS}

2.1 Typing

The $\texttt{string\_in}[r]$ type is parameterized by regular expressions; if $e : \texttt{string\_in}[r]$, then $e \in r$. Mapping from an arbitrary $\texttt{string}$ to a $\texttt{string\_in}[r]$ requires defining a algorithm – in terms of filter – for converting a $\texttt{string\_in}[.*]$ into a $\texttt{string\_in}[r]$. 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 $\texttt{ival}$.

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

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

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

3. Type Safety

The type safety proof relies upon some established results about regular languages. The important definitions and theorems follow.

3.1 Properties of Regular Languages

3.2 Type Safety Proof

Before presenting progress and preservation theorems, we establish some definitions and lemmas necessary for the proofs.

Definition 2 (Types of internal values)**.** The types of internal values are defined 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 $e : \texttt{istring}$.

Definition 2 is an additional assumption about the internal language. To summarize, we assume an internal language containing strings together with operations for string concatentation and filtering. We expect closure over strings for both operations. Recall that ifilter(,i) s only needed for dynamic casts, which may be removed without descreasing the expressivity or even usability of the language.

Theorem 3 (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 $i : \sigma$ for some σ.*
- *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’ $: \texttt{istring}$. It suffices to show that $h(\texttt{string}) = \texttt{istring}$, which follows from the definition of h.

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

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

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

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

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

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

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

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

□

Theorem 4 (Progress)**.** *If $e : T$ then $e : T \Rightarrow^* e' \rightsquigarrow^* i$ where i* $\texttt{ival}$.

Proof. By induction on the derivation of $e : T$.

T-Equiv-Include. Since $e \in \mathcal{L}\{r\}$, $e =$ “s” for some s; therefore, e $\texttt{val}$ by E-Strinval. We have that e $\texttt{val}$ and $e : \texttt{string\_in}[r]$, so $e \rightsquigarrow$ ‘e’ by E-string_in.

T-Minimum. Result follows immediately by induction and E-string_in.

T-Concat. Result follows immediately by induction and E-concat.

T-Convert. Result follows immeidately by induction, E-Convert and E-Convertval.

T-Dconvert. Result follows immediately by induction, E-Dconvertval and E-Dconvert.

T-Filter. Result follows immediately by induction and E-Filter.

□

4. Implementation in Ace