A Typed Lambda Calculus for Input Sanitation

Nathan Fulton
Carthage College
nfulton@carthage.edu

April 11, 2013

Abstract

Programmers often wish to validate or sanitize user input. One common approach to this problem is the use of regular expressions. Unvalidated or insufficiently sanitized user input can cause security problems. Therefore, a compile-time guarantee that input sanitation algorithms are implemented and used correctly could ensure the absence of certain sorts of vulnerabilities. This thesis presents λ_{CS}, a typed lambda calculus which captures the essential properties necessary to achieve such a guarantee.

1 Introduction

Common input sanitation algorithms can be verified by enriching the subtyping relation of a simply typed lambda calculus with types indexed by regular languages.

This is significant because improper sanitation of user input is often the cause of severe security vulnerabilities. Consider the task of constructing an SQL expression from user input. If `u` and `p` are arbitrary strings provided by the user, then such an expression might take the form following:

```
"SELECT * FROM users WHERE name='" + u + "'AND pword= '" + p + "'"
```

Expressions such as the one above are often at the core of authentication mechanisms. In this case, an adversarial user could authenticate as any user by letting `p := 'OR '1'='1`. Aside from the use of stored procedures, a developer could *sanitize* `u` and `p` so that each contains only a small subset of valid SQL tokens.

Although this example is trivial, code injection attacks are notoriously widespread and difficult to prevent. Furthermore, code injection is not the only sort of vulnerability made possible by a lack of proper input validation. For these reasons, insufficient input sanitation remains the top cause of vulnerabilities in web applications [OWASP].

One solution to the input sanitation problem is to provide libraries or frameworks which guarantee user input is properly sanitized or validated. Empirical studies suggest that developers misuse these tools [Scholte et al]. More importantly, bugs sometimes exist in the input validation algorithms used by web frameworks. Therefore, web developers would benefit from a verifier for input sanitation algorithms even if such a verifier were used only to verify the correctness of algorithms used by frameworks.

Verifying the correctness of input sanitation algorithms requires the existence of a formal system in which such algorithms – and their specifications – can be easily defined. This paper defends the thesis that $\lambda_{<:}$ extended with types indexed by the regular languages can be used to verify the correctness of some common input sanitation algorithms. As a matter of taste, we believe that this language-based approach makes verification particularly easy.

2 Definitions and Development

The main result of this paper is constructed by embedding regular expressions into an extension of the simply typed lambda calculus with subtyping. This section presents the central theorems which are used and extended in the results section. Cognoscenti may briefly skim this section, but should take note of Theorem 5 and §2.2.4.

2.1 Regular Languages and Finite Automata

The regular languages are a well-studied class of formal languages. Regular languages are generated by regular expressions, which often find use in input sanitiation and input validation algorithms. For example, a developer hoping to correctly sanitize `u` and `p` could do so by replacing any substring which is not alphanumeric with an empty string. This section establishes a few well-known results about regular languages.

$\langle r \rangle ::= \emptyset \mid \epsilon \mid a \mid rr \mid r + r \mid r*$

Figure 1: Syntax of regular expressions

Definition 1. A **regular expression** (REGEX) is an expression generated by the grammar in Figure 1, where ϵ is an empty string, a is an element of some finite alphabet Σ, rr denotes a union, $r + r$ denotes concatenation, and $r*$ is the Kleene closure of r.

We sometimes restrict $. \notin \Sigma$ (as defined in 1), then use $.$ to range over Σ. For instance, $.*$ matches every string.

Definition 2. A **finite automaton** (FA) is a 5-tuple $M =< Q, \Sigma, \delta, q_0, F >$ where Q is a set of states, q_0 is an initial state, Σ is a finite alphabet, δ is a transition function, and F is a set of final states.

A string is said to be *accepted* by a finite automaton if, when provided as an input stream for M, M terminates in a final state. We may also discuss the language generated by M, which is the set of all strings accepted by M. Also, any non-deterministic FA (NFA) is equivalent to a deterministic FA (FA), and we use this equivalence freely.

Theorem 3. *Correspondence Theorem. The finite automoton and regular expressions recognize exactly the **Regular Languages**, and every FA corresponds to a REGEX.*

Proof. See [Hopcroft and Ullman]. □

Theorem 4. *Closure Properties. If r is a regular expression, then its complement (r') is a regular expression. If r and s are regular expressions, then $r \backslash s$ is a regular expression.*

Proof. See [Hopcroft and Ullman]. □

Theorem 5. *Coercion Theorem. Suppose that R and L are regular expressions, and that $s \in R$ is a finite string. Let $s' := coerce(R, L, s) := s$ with all maximal substrings recognized by L replaced with ϵ. Then s' is recognized by $(R \backslash L) + \emptyset$ and the question 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 G). Define an FA H as a DFA corresponding to the NFA constructed by combining F and G' such that H accepts only if F and G' accept or if s is empty (this construction may result in an exponential blowup in state size). Clearly, H corresponds to $R \backslash L + \emptyset$ under the Correspondence Theorem (3). 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. □

Part of the Coercion Theorem may be a novel result, but is fairly trivial. In any case, this face is a necessary component of later proofs.

2.2 A Lambda Calculus with Subtyping

The typed lambda calculi are a family of formal systems often used by programming language researchers. Practically, type systems are useful because they rule out the presence of certain classes of runtime errors. Theoretically, type systems are interesting because many correspond to logic systems. In both cases, a typing relation is a mapping from terms to types that validates *type safety*, which states that evaluation of well-typed terms always results in a value (or else evaluation never halts). The central result of this paper is a proof of type safety for λ_{CS}.

This paper presents three type systems, each as an extension of the previous. Each system is defined in three parts – a syntax, typing relation and operational semantics. In systems with a subtype relation, the subtype relation is defined separately from the rest of the typing relation. The first system defined is the simply-typed lambda calculus, as presented in [Pierce].

2.2.1 The Simply Typed Lambda Calculus [Pierce]

$\langle t \rangle ::=$		terms:
	x	variable
$\mid$	$\lambda x : T.t$	abstraction
$\mid$	$t\ t$	application
$\langle v \rangle ::=$		values:
	$\lambda x : T.t$	abstraction
$\langle T \rangle ::=$		types:
	$T \to T$	type of functions
$\langle \Gamma \rangle ::=$		typing context:
	$\emptyset$	empty context
$\mid$	$\Gamma, x : T$	term variable binding

Figure 2: Syntax of λ

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-VAR}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2} \qquad \text{T-ABS}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \qquad \text{T-APP}$$

Figure 3: Typing relation for λ

$$\frac{t_1 \Rightarrow t_2}{t_1 t_2 \Rightarrow t_1' t_2} \qquad \text{E-App1}$$

$$\frac{t_2 \Rightarrow t_2'}{v_1 t_2 \Rightarrow v_1 t_2'} \qquad \text{E-App2}$$

$$(\lambda x : T_{11}.t_{12}) v_2 \Rightarrow [x/v_2] t_{12} \qquad \text{E-AppAbs}$$

Figure 4: Operational semantics for λ

Figure 2 defines the grammar of λ, which consists of terms, values, types and a typing context. Terms have only three forms – variables, abstractions and applications. Only a subset of terms generated by this grammar are well-typed.

The typing relation for λ defines two rules for the type constructor $T \to T$: `T-ABS` and `T-APP`.

Definition 6. An **introduction form** for a type constructor T is a typing rule with a conclusion of the form $\Gamma \vdash t : T$ and a hypothesis which does not contain a judgement of the form $\vdash t : T$.

For example, the rule `T-ABS` is sometimes called an *introduction form* because it defines a way in which terms with type of the form $T \to T$ may be constructed.

Definition 7. An **elimination form** for a type constructor T is a typing rule with a hypothesis which contains a judgement of the form $\Gamma \vdash t : T$ and a conclusion which does not contain a judgement of similar form.

For example, `T-APP` is called an *elimination form* for the type constructor $T \to T$ because it describes how terms of this type can be used.

Definition 8. Type Safety is a relationship between the typing relation and evaluation rules for a type system. A system is said to be type safe if it satisfies two criteria:

1. If $\Gamma \vdash e : T$ and $e \Rightarrow e'$ then $\Gamma \vdash e' : T$,
2. If $t : T$ then t is a value or $t \Rightarrow t'$ for some t'.

A useful characterization of *type safety* is as a guarantee about the behavior of evaluation of well-typed terms. Evaluation is said to have gone *wrong* if a well-typed term is not a value and cannot be evaluated. In this sense, *type safety* is the guarantee that evaluation of well-typed terms never goes wrong.

2.2.2 Subtyping

Definition 9. A **subtype relation** ($<:$) is a pre-order on types that validates the *subsumption principle*, which states if $S <: T$ then a value of type S may be provided whenever a value of type T is required [Harper].

The obvious intuition is to consider types as sets, and the subtyping relation as set inclusion. This intuition is problematic because it considers introductory forms but says nothing of elimination forms [Harper].

Those familiar with Object-oriented Programming languages may recognize inheritance as a (rather complicated) form of subtyping. The subtype relation provided by modern programming languages is rich and expressive. In this section, we focus on a very simple type system in hopes of illuminating the characteristics of subtyping essential to λ_{CS}.

2.2.3 Definition of $\lambda_{<:}$

The system $\lambda_{<:}$ is a simply typed lambda calculus extended with a subtype relation. The following presentation of $\lambda_{<:}$ is derivative of the presentation in [Pierce]. Our slight modifications are merely cosmetic (see lemmas below), but simplify the definition of extensions in §3.

...extends figure 2

$\langle t \rangle ::=$	terms:
$\quad x$	variable
$\quad \mid \lambda x : T.t$	abstraction
$\quad \mid t\ t$	application
$\langle v \rangle ::= \lambda x : T.t$	values
$\langle T \rangle ::=$	types:
$\quad$ Top	Maximum type
$\quad \mid$ NSTop	Maximum type for non-strings
$\quad \mid T \to T$	type of functions
$\langle \Gamma \rangle ::= \emptyset \mid \Gamma, x : T$	typing context

Figure 5: Syntax of $\lambda_{<:}$

The choice of which syntactic categories to collapse in Figure 5 foreshadows §3. The inclusion of `NSTop` diverges from the presentation in [Pierce].

$$\frac{t_1 \Rightarrow t_2}{t_1 t_2 \Rightarrow t_1' t_2} \quad \textsc{E-App1}$$

$$\frac{t_2 \Rightarrow t_2'}{v_1 t_2 \Rightarrow v_1 t_2'} \quad \textsc{E-App2}$$

$$\frac{T_{11} <: NSTop}{(\lambda x : T_{11}.t_{12}) v_2 \Rightarrow [x/v_2] t_{12}} \quad \textsc{E-AppAbs-CS}$$

Figure 6: Operational semantics for $\lambda_{<:}$

Figure 6 is not an extension of Figure 4 because the former restricts the `E-AppAbs-NS` rule on the type of the domain. This diverges from the presentation in [Pierce]. In the canonical presentation, evaluation rules are unchanged.

...extends figure 3

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-VAR}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \qquad \text{T-ABS}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \qquad \text{T-APP}$$

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \qquad \text{T-SUB}$$

Figure 7: Typing relation for $\lambda_{<:}$

$$S <: S \qquad \text{S-Refl}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \qquad \text{S-Trans}$$

$$S <: Top \qquad \text{S-Top}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{S-Arrow}$$

Figure 8: Subtype relation for $\lambda_{<:}$

Note that `S-Arrow` is contravariant. For intution, consider that we may safely restrict a function's domain and expand its codomain. We could write one of these results as an admissible covariant rule and derive the other using `T-Sub`, but this only complicates the proofs.

Theorem 10. *Progress. If $t : T$ for some T then t is a value or there is some t' with $t \to t'$.*

Proof. See [Pierce]. □

Theorem 11. *Preservation. If $t : T$ and $t \Rightarrow t'$ then $t' : T$.*

Proof. See [Pierce]. □

3 Results

In this section we define λ_{CS}, an extension of $\lambda_{<:}$. The system λ_{CS} is constructed from $\lambda_{<:}$ in three steps.

1. **Syntax.** Extend the syntax of $\lambda_{<:}$ with regular expressions, a type `STop` disjoint from `NSTOP` and a set of `CS_L` types indexed by regular expressions.
2. **Subtype Relation.** Modify the subtype relation so that all `CS_L` types are subtypes of `STop`, and all non-CS types are subtypes of `NSTop`.
3. **Evaluation.** Extend the evaluation relation of $\lambda_{<:}$ with an `E-AppAbs-S` which provides a semantics for the elimination form of the type constructor $T \to T$ when the left type is a subtype of `STop`.

3.1 Definition of λ_{CS}

This section defines λ_{CS} as an extension of $\lambda_{<:}$, discusses some rationale for the system's definition, then proceeds with the type safety proof.

...extends Figure 5 and embeds Figure 1

$\langle t \rangle$ ::=	terms:
x	variable
$\mid\ \lambda x : T.t$	abstraction
$\mid\ t\ t$	application
$\mid\ 's'$	string
$\mid\ \text{replace}\langle T \rangle('s')$	string replacement
$\langle v \rangle$ ::=	values:
$\lambda x : T.t$	abstraction
$\mid\ 's'$	string
$\langle l \rangle ::= \langle r \rangle$	regular expressions
$\langle T \rangle$ ::=	types:
Top	Maximum type
$\mid$ NSTop	Maximum type for non-strings
$\mid$ STop	Maximum type for strings
$\mid\ CS_l$	Constrained String type
$\mid\ T \to T$	type of functions
$\langle \Gamma \rangle ::= \emptyset \mid \Gamma, x : T$	typing context

Figure 9: Syntax of λ_{CS}

...extends figure 7

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-VAR}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \qquad \text{T-ABS}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \qquad \text{T-APP}$$

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \qquad \text{T-SUB}$$

$$\frac{\Gamma \vdash t : CS_S \qquad CS_S <: STop \qquad CS_T <: STop}{\Gamma \vdash replace < T > (t) : CS_{(S \setminus T) + \emptyset}} \qquad \text{T-AppAbs-CS}$$

Figure 10: Typing relation for λ_{CS}

...extends Figure 8

$$S <: S \qquad \text{S-Refl}$$

$$S <: Top \qquad \text{S-Top}$$

$$NStop \cap STop = \emptyset \qquad \text{S-Disjoint}$$

$$STop = CS_{.*} \qquad \text{S-STop}$$

$$\frac{l \in r}{CS_l <: CS_r} \qquad \text{S-RE}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \qquad \text{S-Trans}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{S-Arrow}$$

Figure 11: Subtype relation for λ_{CS}

...extends Figure 6

$$\frac{t_1 \Rightarrow t_2}{t_1 t_2 \Rightarrow t_1' t_2} \qquad \text{E-App1}$$

$$\frac{t_2 \Rightarrow t_2'}{v_1 t_2 \Rightarrow v_1 t_2'} \qquad \text{E-App2}$$

$$\frac{T_{11} <: NSTop}{(\lambda x : T_{11}.t_{12}) v_2 \Rightarrow [x/v_2] t_{12}} \qquad \text{E-AppAbs-NS}$$

$$\frac{v : CS_T \qquad CS_T <: STop \qquad CS_S <: STop}{replace < S > (v) \Rightarrow coerce(S, T, [v]))} \qquad \text{E-AppAbs-CS}$$

Figure 12: Operational semantics for λ_{CS}

The coerce function referred to in Figure 12 is extra-linguisitic, and was defined in Theorem 5. The brackets around v indicate that v should be evaluated before the rule is applied.

3.2 Discussion

The subtyping relation between Constrained Strings is defined using language inclusion (`S-RE`). The definition flirts with the problematic set inclusion interpretation of subtyping. As proven in §3.3, `E-AppAbs-CS` provides a sufficient elimination form. In other words, terms of the form replace$< T >$ (t) are always evaluated by *coerce*. This realization that terms of the form replace$< T > t$ are evaluated by *coerce* gives rise to the progress lemma. The use of $NStop$ and $STop$ reduce the amount of work necessary to convert the type safety proofs for $\lambda_{<:}$ into type safety proofs for λ_{CS}.

3.3 Type Safety Proof

Theorem 12. *Structural Properties. The extension of the subtype and typing relations preserves the structural properties of* $\Gamma \vdash t : T$ *(permutation, weakening and substitution.)*

Proof. The extension of the subtype relation is consistent and the extension of the typing relation introduces only an elimination form. Therefore, the structural properties follow from the structural properties of $\lambda_{<:}$. □

Theorem 13. *Progress. If $t : T$ for some T then t is a value or there is some t' with $t \Rightarrow t'$.*

Proof. If $T <: NSTop$ then Progress follows from Theorem 10.

If $T <: STop$ then either t is either a string or has the form replace$< T >$ $('e')$. In the first case t is already a value. In the second, t can make a step. □

Theorem 14. *Preservation. If $\Gamma \vdash t : T$ and $t \to t'$ then $\Gamma \vdash t : T'$.*

Proof. If $T <: NSTop$ then Preservation follows from Theorem 11.

If $T <: STop$, then either t is a string or *replace* $< T >$ $('e')$ for some e. The first case is impossible because t is already a value. In the second case, there is one subcase for `E-AppAbs-CS`. Preservation follows from Theorem 5. □

4 Future Work

The primary features of λ_{CS} are most useful as an extension to a general purpose programming language. For this reason, λ_{CS} is implemented as an extension of the Ace programming language [Fulton]. In this paper, λ_{CS} is presented as an extension to the subtype relation. We are interested in whether λ_{CS} could be combined with other extensions to the subtype relation. Ace has a core type theory called λ_A, which allows programmers to extend the type system without violating the safety guarantees of of the underlying language or other extensions. We are in the process of mechanically verifying the soundness of λ_A.

5 Conclusion

Incorporating regular expressions into the type system allows programmers to *verify* user input. Defining a subtype relation between these types allows programmers to capture common input sanitation algorithms. The formal system presented in this paper provides a simple setting for understanding the most important characteristics of such a system.

References

[Fulton] Fulton, Nathan. Security Through Extensible Type Systems. SPLASH 2012.

[Harper] Harper, Robert. Practical Foundations for Programming Languages. 2012.

[Hopcroft and Ullman] Hopcroft, John E. and Ullman, Jefferey D. Introduction to Automata Theory, Languages, and Computation. 1979.

[OWASP] OWASP. Top Ten Project. 2010.

[Pierce] Pierce, Benjamin. Types and programming languages. MIT Press. 2002.

[Scholte et al] Scholte, Balzarotti, Kirda. Quo vadis? A study of the evolution of input validation vulnerabilities in Web applications. FCDS 2011.