# Security Through Extensible Type Systems

Nathan Fulton
Carthage College, Carnegie Mellon University
nfulton@carthage.edu

## Abstract

Researchers interested in security often wish to introduce new primitives into a language. Extensible languages hold promise in such scenarios, but only if the extension mechanism is sufficiently safe and expressive. This paper describes several modifications to an extensible language motivated by end-to-end security concerns.

***Categories and Subject Descriptors*** D.3.2 [*Programming Languages*]: Language Classifiers - Extensible languages

***Keywords*** Extensibility, Security

## 1. Problem and Motivation

Researchers involved in security research use language primitives to address software verification problems. However, many high-profile vulnerabilities have still not been addressed in popular languages. For example, although improperly sanitized user input is the top cause of vulnerabilities in mobile and web applications [3], popular languages have not incorporated language primitives that guarantee that input has been sanitized. Extensibility mechanisms capable of modularly expressing and verifying such primitives could help decrease the gap between research and practice.

## 2. Background and Related Work

Ace[1] is an extensible language being developed by our group. Ace compiles programs from a typed source language to a typed target language based on specifications associated directly with type definitions, written in a "type-level language". This approach is called "active compilation" [2]. In Ace, the typechecker and code generator call methods associated with user-defined types. This paper presents extensions within Ace, as well as modifications of Ace's extensibility mechanism.

Several languages provide mechanisms for extensibility. Ace uses a novel extension mechanism with novel type-directed specifications called Active Type Checking and Translation [2].

There are many proposed approaches to input sanitation. Unfortunately, many of these are domain specific. For instance, the various methods for sanitizing SQL queries may not be available when sanitizing strings used in other settings. Even when frameworks and libraries provide a sufficient sanitation mechanism, developers may misuse or fail to use these tools. This misuse suggests the need for a simple, universal language primitive that enforces best practices for string sanitation. In this paper, we present such a primitive based on regular expressions.

Regular expression types are used in several languages. Our subtyping extension is motivated by XDuce [1]. However, we believe our use of regular expression types for input sanitation is unique and that portions of our type system which enable this use are novel.

## 3. Approach and Uniqueness

This paper describes several contributions to Ace. This paper describes an extension within Ace motivated by the input sanitation problem. Our extension modifies

[1] https://github.com/cyrus-/ace

$$\frac{}{\mathit{Str} <: L_*}\textsc{CS-INTRO} \qquad \frac{r \text{ is a regular language (RL)}}{L_r <: \mathit{Str}}\textsc{CS-LANG} \qquad \frac{r_1 \in r_2}{L_{r_1} <: L_{r_2}}\textsc{CS-SUBTYPE}$$

$$\frac{t_1 : L_{r_1} \qquad t_2 : L_{r_2}}{t_1 + t_2 : L_{r_1 r_2}}\textsc{CS-CONCAT} \qquad \frac{t_1 : L_{r_1} \qquad r_2 \text{ is an RL}}{\text{remove } \langle r_2 \rangle\ t_1 : L_{r_1 \backslash r_2}}\textsc{CS-REMOVE}$$

$$\frac{t : L_{r_1} \qquad r_2 \text{ is an RL} \qquad r_1 \in r_2}{\text{scast } t, L_{r_2} : L_{r_2}}\textsc{CS-SCAST} \qquad \frac{t : L_{r_1} \qquad r_2 \text{ is an RL} \qquad r_2 \notin r_2}{\text{dyncast } t, L_{r_2} : L_{r_2}}\textsc{CS-DYNCAST}$$

**Figure 1.** Some of the type checking rules relevant to the `ConstrainedString` extension.

both the typing relation and semantics so that input sanitation algorithms may be specified using regular expressions, implemented using our extension, and statically verified during type checking. Implementing this extension required adding support for subtyping and type-casting to the Ace extension mechanism itself. We use an extensibility mechanism influenced by type-directed compilation [5] to enforce a specified correspondence between the source and target type systems.

These contributions demonstrate the efficacy of using an extensible type system to verify mission critical portions of a program.

## Constrained Strings in Ace

We address the need for a language-based approach to input sanitation by providing a type system extension capable of statically checking that a string is in a regular language. The inputs to insecure functions are constrained using a statically specified constrained string type. Our type system (see Figure 1, described later) provides a statically checked mechanism for converting strings into appropriately typed constrained strings. Values that have the type String must be cast to the specified type before use. Our type system provides a statically checked mechanism for achieving this conversion and updating the language a string belongs to when operations are performed on it.

```
1  CS = ConstrainedString
2  @ace.fn
3  def run_query(input):
4      return execute_query(input)
5  run_query = run_query.compile(backend,
6                           CS("(a-z0-9)+"))
7  @ace.fn
8  def sanitize_query(string):
9      return run_query(
10         `remove("^(a-z)+")`(string))
11 sanitize_query = sanitize_query.compile(
       backend, Str, run_query.type)
```

**Listing 1.** An Ace program using constrained strings.

Listing 1 is an Ace program demonstrating the constrained string extension. Ace functions are declared generically, then specialized with a `backend` (specifying the target language) and input types. Return types are locally inferred from input types. Functions are compiled and specialized, one function at a time, on lines 5 and 11. In Ace, code generation is defined in terms of a backend; here, we assume a backend is already defined. When `sanitize_query` is compiled, our typechecker ensures that the argument to `run_query` matches the input type specified on line 5.

The rest of Listing 1 is an application of the type-checking rules for constrained strings, as described in Figure 1. The expression surrounded by back-ticks on Line 7 uses static evaluation to parameterize the remove function with a regular expression. Calling the resulting function removes all substrings matching the constrained string type with the static string (see `CS-LANG` in Figure 1.) In this case, the value returned by `sanitary_query` has type `CS("(a-z)+")`. Subtyping between constrained strings permits this expression to be used where the larger type, `CS("(a-z0-9)+")`, is specified.

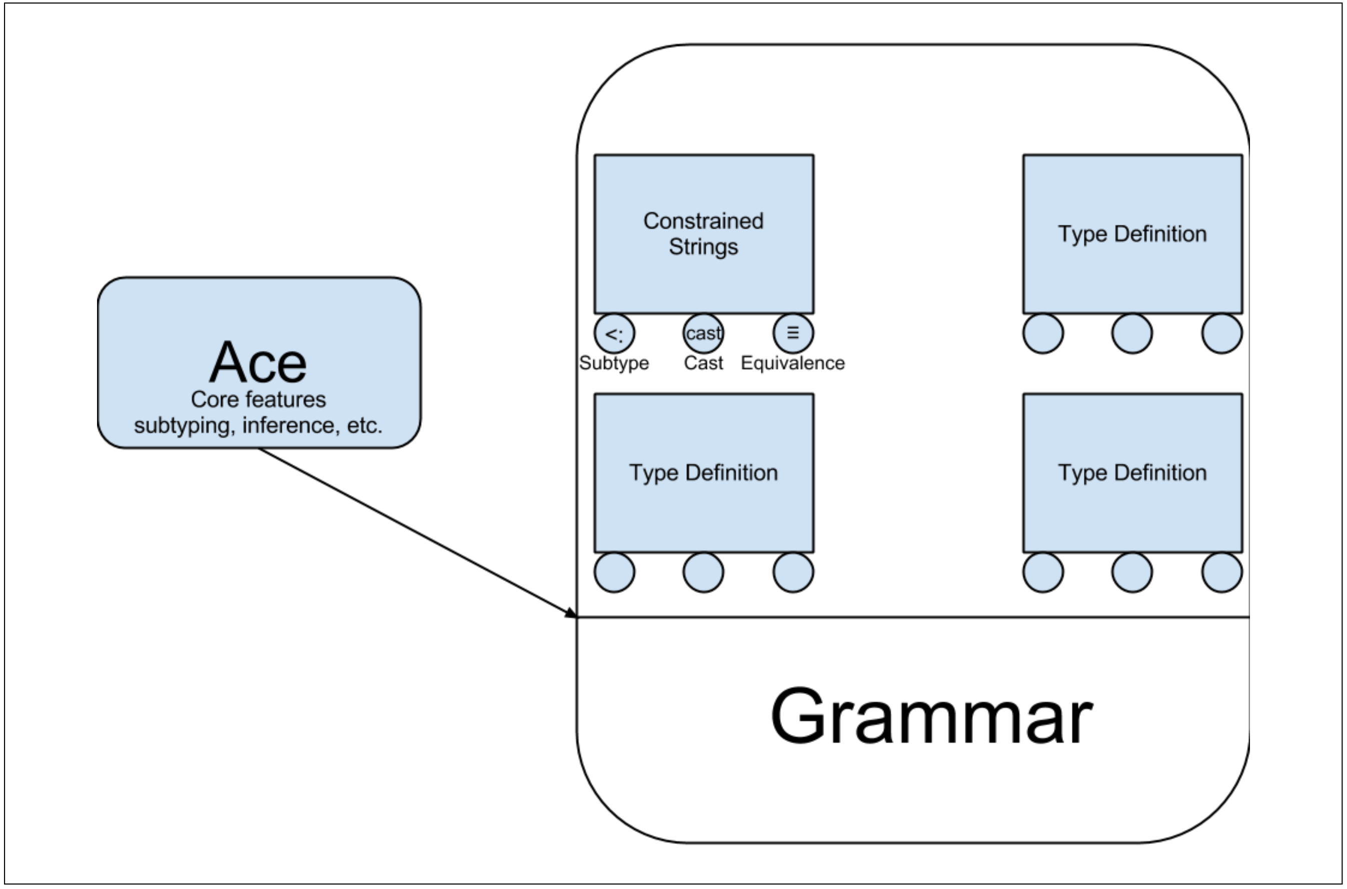

**Figure 2.** Structure of Ace

**Constrained Strings** Constrained strings are a family of types parameterized by the regular languages. We believe the checking rules provided in Figure 1 describe a sufficiently expressive relation for addressing these needs:

- The `CS-LANG` rule establishes a correspondence between each regular language $r$ and its constrained string type $L_r$.
- The `CS-SUBTYPE` defines subtyping as language inclusion: if the strings in one language are a subset of the strings in another language, then the smaller language is a subtype of the larger.
- The `CS-INTRO` rule establishes $Str$ as the top of the subtype relation between constrained strings.
- The `CS-INTRO` rule admits arbitrary strings into the typing relation for constrained strings.
- The `CS-CONCAT` rule provides a form for constrained concatenation.
- The `CS-REMOVE` rule removes all substrings of $t_1$ matching $r_2$. Many sanitation algorithms have this form. This rule captures a common pattern for input sanitation. It is worth noting that the language $r_1 \backslash r_2$ is defined (unconventionally) to also include the empty set.
- The `CS-SCAST` rule allows static casts between constrained string types when casting into a larger type. In Ace, static casts are implicit at function call sites. This is significant because static casts weaken the guarantee provided by $t : L_{r1}$.
- The `CS-DYNCAST` rule allows casts between constrained string types when casting into a smaller type. The evaluation rule for `CS-DYNCAST` inserts a runtime check.

Sanitation algorithms may be implemented in two ways. Rules such as `CS-CONCAT` may be used to build up strings from previously constrained input (using either `CS-DYNCAST` or `CS-REMOVE`. Alternatively, `CS-REMOVE` can be used to constrain an arbitrary string. Listing 1 demonstrates the latter approach.

In addition to the work described here, we have defined the constrained string types as an extension to $\lambda_{<:}$ (the simply-typed lambda calculus with subtyping) and proven type safety.

## Subtyping and Casting

The subtype relation in Figure 1 is not necessary to use constrained strings because `CS-SCAST` is not defined in terms of subtyping. However, subtyping allows the omission of these casts and is a common feature in programming languages.

Ace's extensibility mechanism provided no support for either subtyping or checked downcasts. Adding support for these features required modifying Ace's extensibility mechanism so that both backends and type definitions may define subtyping and checked downcasts.

## Checking Extensions to Ace

Figure 2 is a representation of the structure of Ace. Type checking and code generation are dispatched to each type definition. For instance, the Constrained String extension provides a mechanism for type-checking its subtype relation, and for both type-checking and performing code generation on downcasts of constrained strings.

Since Ace allows multiple extensions to both typing and evaluation, we must check that extensions themselves are implemented correctly. Otherwise, one extension would be capable of breaking the safety guarantees provided by both the underlying language and other extensions. In the case of our constrained strings extension, this is particularly important.

To have confidence in the security guarantees offered by an extension, we must check that the extension itself is correct. As an example, if Ace is compiling to C, then the constrained string type must enforce the typing relation given in Figure 1 and also generate code of type `char*`. The strategy for mechanizing this test follows from the design of Ace. Each expression generated by the compiler is tagged with an expected type, specified by the extension writer. The Ace type checker ensures that each generated expression has the expected type.

## Future Work

Empirical studies suggest that SQL injection attacks are easier to fix than other types of vulnerabilities due to the effectiveness of stored procedures [4]. We believe constrained strings may be capable of both finding and fixing these more difficult input sanitation errors. Testing this hypothesis requires implementing constrained strings as an extension to a popular language for web applications.

Only portions of the Ace type system used by a program get checked during compilation. We plan to statically check these extensions be using a dependently-typed type-level language.

## Acknowledgments

The author is advised by Cyrus Omar and Jonathan Aldrich and supported by a grant from the Army Research Office under Award No. W911NF-09-1-0273.

## References

[1] Hosoya and Pierce. XDuce: A Statically Typed XML Processing Language. ACM Transactions on Internet Technology, 3(2):117-148, May 2003.

[2] Omar, Cyrus. Active Type Checking and Translation. SPLASH SRC 2012.

[3] OWASP. Top Ten Project. 2010.

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

[5] Tarditi, Morrisett, Cheng, Stone, Harper, Lee. Til: a type-directed optimizing compiler for ML. PLDI '96.