Abstract parsing of string updates and user input

Kyung-Goo Doh[1] *, Hyunha Kim[1]*, and David A. Schmidt[2] **

[1] Hanyang University, Ansan, South Korea
[2] Kansas State University, Manhattan, Kansas, USA

Abstract. We extend our formulation of demand-driven, static-analysis-based, abstract parsing of the strings generated by PHP scripts to include strings that are generated from string-replacement operators and user input. Our approach combines LR(k)-parsing technology and data-flow analysis to analyze, in advance of execution, the documents generated dynamically by a script. String-replacement operations are computed statically by *composing* the finite-state automaton defined by a string replacement with the finite-state control of the LR(k)-parser, and user input is predicted and processed by characterizing the input by an LR(k)-grammer and analyzing the strings generated by the grammer. Our work is implemented in Objective Caml.

1 Introduction

Scripting languages like PHP, Perl, Ruby, and Python use string values to encode data structures, queries, web pages, etc., and then pass the strings to another processor. For example, one might write a PHP script that assembles as a string an SQL query or an HTML page or an XML document. The script might assemble the string incorrectly or include input submitted by a malicious user, generating a cross-site-scripting or injection attack.

In earlier work [6], we showed how to employ LR(k)-parsing technology and data-flow analysis to analyze statically a program that dynamically generates documents as strings and to parse statically those strings with respect to the the *context-free reference grammar* for the document language to which the strings must conform. We did this by implementing a demand-drived abstract interpretation that computed not the strings themselves but *abstract parse stacks* that encode the context-free structure of the strings that would be generated by the script at run-time. (For example, if a script must generate well-formed SQL queries, then the analyzer uses a reference grammar for SQL to analyze

* Supported in part by grant R01-2006-000-10926-0 from the Basic Research Program of the Korea Science and Engineering Foundation and in part by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF), R11-2008-007-01003-0.

** Supported by NSF CNS-0939431.

the strings that will be constructed by the script, in advance of running the script.) This is done by using the states within the LR(k) parser's controller, organized into parse stacks, as abstract-interpretation values, so that rather than tracking the approximate value of a string assigned to a variable, we track the state of the parser applied to that string. This gives better precision, because the approximate parse stacks represent context-free structure rather than just regular-language structure [2–5, 10, 11].

The implementation proved successful at analyzing PHP programs that dynamically generate HTML documents [6], detecting coding errors that generate misspelled, missing, and mismatched tags in generated documents.

This paper documents the next stage of work: extending the demand-driven static analysis to handle string-replacement operators and to handle user input. Both problems and their solutions are nontrivial:

- A PHP-style string replacement operator, e.g.,

  ```
  y = replace 'aa' by 'b' in x
  ```

  defines a finite-state string transducer to which `x`'s string-value is supplied as input. The transducer's output is saved in `y`'s cell. The static analyzer models the transducer by *composing it with the finite-state control of the LR(k)-parser for the reference grammar*. The compound parser is used to verify that the string-replacement's output conforms to the reference grammar.
- It is impossible to know in advance the input a user will supply to a script. But if the input is limited to fall within the set of strings generated by a reference grammar, then that grammar's start nonterminal can be supplied an the input to the static analyzer for abstract parsing. Our analyzer *treats grammar nonterminals as valid inputs, just like sequences of terminal symbols.* For example, if the range of user inputs is contained within this syntax:

  $$S ::= \mathtt{a} \mid \mathtt{a}S$$

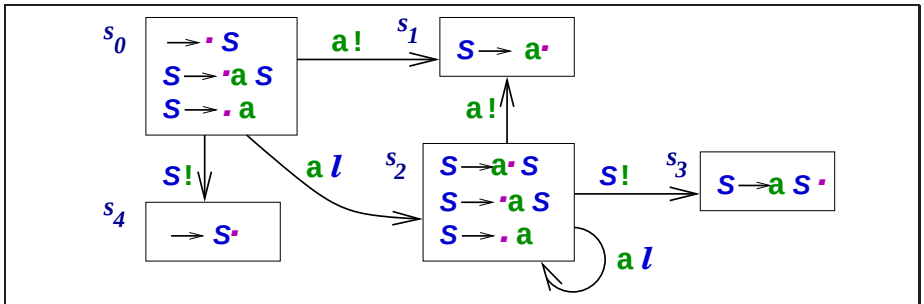
  then for this script,

`x = getinput`$_S$	$S = \mathtt{a} \cup \mathtt{a} \cdot S$
`y = replace 'aa' by 'b' in x`	$X = S$
`print y`	$Y = replace(\mathtt{aa}, \mathtt{b}, X)$

  The analyzer will generate the data-flow equations on the right, which indicate that `x`'s value can be any S-string. `x`'s values are composed with the transducer defining `y` to ensure that the printed string conforms to the reference grammar.

2 LR(k)-parsing

Our treatment of string-replacement operators depends on LR(k)-parsing technology. For review, Figure 1 shows an LR(1)-grammar, its parse controller, its



Shift transitions:

$[\ell \hookrightarrow \mathtt{a} \hookrightarrow s_0] \rightarrow s_0 :: [\ell \hookrightarrow s_2]$
$[\mathtt{!} \hookrightarrow \mathtt{a} \hookrightarrow s_0] \rightarrow s_0 :: [\mathtt{!} \hookrightarrow s_1]$
$[\mathtt{!} \hookrightarrow S \hookrightarrow s_0] \rightarrow s_0 :: [\mathtt{!} \hookrightarrow s_4]$
$[\ell \hookrightarrow \mathtt{a} \hookrightarrow s_2] \rightarrow s_2 :: [\ell \hookrightarrow s_2]$
$[\mathtt{!} \hookrightarrow \mathtt{a} \hookrightarrow s_2] \rightarrow s_2 :: [\mathtt{!} \hookrightarrow s_1]$
$[\mathtt{!} \hookrightarrow S \hookrightarrow s_2] \rightarrow s_2 :: [\mathtt{!} \hookrightarrow s_3]$

Reduce transitions:

$s_i[m \hookrightarrow s_1] \rightarrow [m \hookrightarrow S \hookrightarrow s_i]$
$s_i :: s_j[m \hookrightarrow s_3] \rightarrow [m \hookrightarrow S \hookrightarrow s_i]$

where ! denotes end of input and ℓ denotes any non-! input symbol.

parse stack (top lies at right)	input sequence
$[s_0]$	aa!
$[\mathtt{a} \hookrightarrow s_0]$	a!
$[\mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_0]$	! (next, do shift transition)
$s_0 :: [\mathtt{a} \hookrightarrow s_2]$	!
$s_0 :: [\mathtt{!} \hookrightarrow \mathtt{a} \hookrightarrow s_2]$	(do shift transition)
$s_0 :: s_2 :: [\mathtt{!} \hookrightarrow s_1]$	(reduce $S \rightarrow \mathtt{a}$; pop one state and insert S)
$s_0 :: [\mathtt{!} \hookrightarrow S \hookrightarrow s_2]$	(do shift transition)
$s_0 :: s_2 :: [\mathtt{!} \hookrightarrow s_3]$	(reduce $S \rightarrow \mathtt{a}S$; pop two states and insert S)
$[\mathtt{!} \hookrightarrow S \hookrightarrow s_0]$	(do final transition)
$s_0 :: [\mathtt{!} \hookrightarrow s_4]$	(finished)

Fig. 1. *goto* controller for $S \rightarrow \mathtt{a}S \mid \mathtt{a}$ and an example parse of aa!

state-transition rules, and an example parse. An LR(k)-parse configuration is a parse-stack, current state pair, $s_i :: s_j :: \cdots :: s_n[\ell_k \hookrightarrow \ell_{k-1} \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s_m]$. That is, state s_m, on the top of the stack, determines the next parse move based on the inputs, $\ell_k \cdots \ell_0$, where $\ell_1 \cdots \ell_k$ form the lookahead and ℓ_0 is the input symbol.

3 Abstract parsing

Here is a small example that shows how to apply the parser in the previous section to predict the grammatical structure of a string to be generated by a program. See Figure 2, left column, for a sample script, and see the right column for the flow equations generated from the script. Say this program should print only well-formed S-phrases. We must analyze the value of x at point $X3$ to see

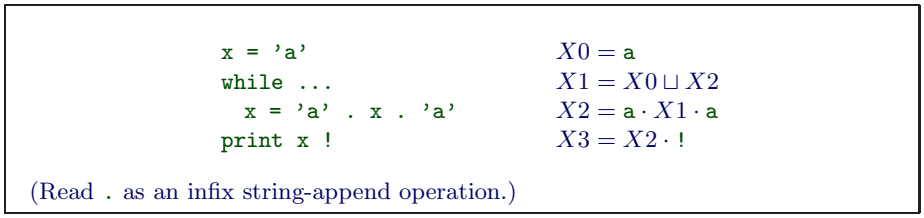


Fig. 2. Sample program and its data-flow equations

if it is S-structured. To do this, we both calculate the string *and* parse it — we interpret the flow equations in Figure 2 as *functions* that map an input parse state to (a set of) LR(1)-parse configurations.

We calculate the set of parses (parse configurations) for $X3$ from start state $[s_0]$. We write this as the function call, $X3[s_0]$, and generate this first-order flow equation:

$$X3[s_0] = (X2 \cdot \texttt{!})[s_0]$$

which itself demands a parse of the string generated at point $X2$ followed by a parse of the "end-of-input" token, `!`. The calculation proceeds like this:

$$\begin{aligned} X3[s_0] &= (X2 \cdot \texttt{!})[s_0] = X2[s_0] \oplus \texttt{!} \\ X2[s_0] &= (\texttt{a} \cdot X1 \cdot \texttt{a})[s_0] = \texttt{a}[s_0] \oplus (X1 \cdot \texttt{a}) \\ &= [\texttt{a} \hookrightarrow s_0] \oplus (X1 \cdot \texttt{a}) = (X_1 \cdot \texttt{a})[\texttt{a} \hookrightarrow s_0] \\ &= X1[\texttt{a} \hookrightarrow s_0] \oplus \texttt{a} \end{aligned}$$

Combinator $\cdot$ is composition, and $\oplus$ sequences the parse steps: for parse configuration, c, and function, F, $c \oplus F = \mathit{tail}(c) :: F(\mathit{head}(c))$. (E.g., $(s_0 :: [\texttt{a} \hookrightarrow s_2]) \oplus F = s_0 :: (F[\texttt{a} \hookrightarrow s_2])$ — F operates on the current parse state, and the underlying stack is left untouched.)

The above calculation demands a parse of the string defined by $X1$ starting from state $[\texttt{a} \hookrightarrow s_0]$:

$$\begin{aligned} X1[\texttt{a} \hookrightarrow s_0] &= X0[\texttt{a} \hookrightarrow s_0] \cup X2[\texttt{a} \hookrightarrow s_0] \\ X0[\texttt{a} \hookrightarrow s_0] &= \texttt{a}[\texttt{a} \hookrightarrow s_0] = [\texttt{a} \hookrightarrow \texttt{a} \hookrightarrow s_0] = s_0 :: [\texttt{a} \hookrightarrow s_2] \\ X2[\texttt{a} \hookrightarrow s_0] &= (\texttt{a} \cdot X1 \cdot \texttt{a})[\texttt{a} \hookrightarrow s_0] = \texttt{a}[\texttt{a} \hookrightarrow s_0] \oplus (X1 \cdot \texttt{a}) \\ &= s_0 :: [\texttt{a} \hookrightarrow s_2] \oplus (X1 \cdot \texttt{a}) = s_0 :: (X1[\texttt{a} \hookrightarrow s_2] \oplus \texttt{a}) \end{aligned}$$

That is, $X1[\texttt{a} \hookrightarrow s_0]$'s parse configurations is a set union, where $X0[\texttt{a} \hookrightarrow s_0]$ computes to a shift step, where s_0 is shifted onto the stack and the new state is $[\texttt{a} \hookrightarrow s_2]$. (Since our examples are simplistic and most all of the calculated sets are singletons, we normally omit the enclosing $\{\cdots\}$ brackets.)

The parse of $X2[\texttt{a} \hookrightarrow s_0]$ causes a shift of s_0 and a parse of $X1[\texttt{a} \hookrightarrow s_2]$. This last request generates these three new first-order equations:

$$\begin{aligned} X1[\texttt{a} \hookrightarrow s_2] &= X0[\texttt{a} \hookrightarrow s_2] \cup X2[\texttt{a} \hookrightarrow s_2] \\ X0[\texttt{a} \hookrightarrow s_2] &= s_2 :: [\texttt{a} \hookrightarrow s_2] \\ X2[\texttt{a} \hookrightarrow s_2] &= s_2 :: (X1[\texttt{a} \hookrightarrow s_2] \oplus \texttt{a}) \end{aligned}$$

This completes the set of equations that must be solved to compute the parse configurations for the original query, $X3[s_0]$. We use the usual least-fixed point calculations to solve the equations. First,

$$X1[\mathtt{a} \hookrightarrow s_2] = s_2 :: [\mathtt{a} \hookrightarrow s_2] \cup s_2 :: (X1[\mathtt{a} \hookrightarrow s_2] \oplus \mathtt{a})$$

The solution is $\{s_2^i :: [\mathtt{a} \hookrightarrow s_2] \mid i \in 1, 3, 5, \cdots\}$. Our analysis approximates this infinite set by the configuration, $s_2^+ :: [\mathtt{a} \hookrightarrow s_2]$, which is the least-fixed point in the finite-height lattice of finite stack configurations coded with regular-expression notation [6]. From this result, we obtain

$$\begin{array}{ll} X1[\mathtt{a} \hookrightarrow s_2] & = s_2^+ :: [\mathtt{a} \hookrightarrow s_2] \\ X1[\mathtt{a} \hookrightarrow s_0] & = s_0 :: s_2^* :: [\mathtt{a} \hookrightarrow s_2] \\ X2[s_0] & = s_0 :: s_2^+ :: [\mathtt{a} \hookrightarrow s_2] \end{array}$$

which let us calculate

$$\begin{array}{l} X3[s_0] = X2[s_0] \oplus \mathtt{!} = s_0 :: s_2^+ :: \mathtt{!}[\mathtt{a} \hookrightarrow s_2] = s_0 :: s_2^+ :: [\mathtt{!} \hookrightarrow \mathtt{a} \hookrightarrow s_2] \\ \quad = s_0 :: s_2^+ :: [\mathtt{!} \hookrightarrow s_1] \quad (s_2^+ :: s_2 \text{ is approximated to } s_2^+) \\ \quad = s_0 :: s_2^* :: [\mathtt{!} \hookrightarrow S \hookrightarrow s_2] \quad (\text{reduce } S \to a) \\ \quad = s_0 :: s_2^+ :: [\mathtt{!} \hookrightarrow s_3] \\ \quad = s_0 :: s_2 :: [\mathtt{!} \hookrightarrow s_3] \cup s_0 :: s_2^+ :: [\mathtt{!} \hookrightarrow s_3] \quad (\text{case split}) \\ \quad = [\mathtt{!} \hookrightarrow S \hookrightarrow s_0] \cup s_0 :: s_2^*[\mathtt{!} \hookrightarrow S \hookrightarrow s_2] \quad (\text{reduce } S \to \mathtt{a}S) \\ \quad = s_0 :: [\mathtt{!} \hookrightarrow s_4] \quad (\text{right operand repeats; adds nothing to fixed point}) \end{array}$$

This proves that all possible string values of `x` at the `print` command are well-structured S-phrases. The implementation calculates the answer with a demand-driven version of the classic least-fixed-point worklist algorithm [1, 7–9].

4 Definitions of parsing and collecting semantics

An LR(k) *parse-stack configuration* is a sequence, $s_0 :: s_1 :: \cdots s_i :: [\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, $0 \leq j \leq k$, where $s_0 \cdots s_i, s$ are states from the parser controller; ℓ_0 is the input symbol; and $\ell_1 \cdots \ell_j$ are the lookahead symbols. $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$ is the *parse state* and will always be presented as the "top" of the parse-stack configuration.

A parse of input symbols $a_1 \cdots a_n!$ is defined as $[\![a_1 \cdots a_n!]\!][s_0]$, where s_0 is the parse controller's start state. Let c stand for a parse state. The transition rules in Figure 1 can be formalized as

$[\![\mathtt{a}]\!][\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] = move([\mathtt{a} \hookrightarrow \ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s])$

$[\![E1 \cdot E2]\!]c = move([\![E1]\!]c \oplus [\![E2]\!])$
where $(s_0 :: s_1 :: \cdots s_i :: c') \oplus F = s_0 :: s_1 :: \cdots s_i :: F(c')$

$move(s_0 :: \cdots :: s_i :: [\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]) =$
if s is a final (reduce) state for grammar rule, $N \to U_1 U_2 \cdots U_m$, and $m \leq n$,
then return $move(s_0 :: \cdots :: s_{n-m} :: [\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow N \hookrightarrow s_{n-m+1}])$
(pop top m states, and insert N at front of input stream)
else if there is a match of $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$ to the left-hand-side of a transition rule,
$[\ell_k \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] \to [\ell_k \hookrightarrow \cdots \hookrightarrow \ell_1 \hookrightarrow s']$,
then return $move(s_0 :: s_1 :: \cdots s_i :: s :: [\ell_k \hookrightarrow \cdots \hookrightarrow \ell_1 \hookrightarrow s'])$
(shift)
else return $s_0 :: s_1 :: \cdots s_i :: [\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, as is.

The next definition of interest is the semantics of the flow equations extracted from a script. A flow equation takes the form, $X = E$, where

$$E ::= \mathtt{a} \mid E1 \cdot E2 \mid E1 \sqcup E2 \mid X_j$$

The semantics is called the *collecting semantics* and is defined like this:

$[\![E]\!] : ParseState \to \mathcal{P}(ParseConfiguration)$

$[\![\mathtt{a}]\!][\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] = \{move([\mathtt{a} \hookrightarrow \ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s])\}$

$[\![E1 \cdot E2]\!]c = \{move(c') \mid c' \in [\![E1]\!]c \oplus [\![E2]\!]\}$
where $S \oplus F = \{tail(c) :: F(head(c)) \mid c \in S\}$

$[\![E1 \sqcup E2]\!]c = [\![E1]\!]c \cup [\![E2]\!]c$

$[\![X_j]\!]c = [\![E_j]\!]c$, where $X_j = E_j$ is the corresponding flow equation

The definition shows that sets can result from the calculation of the collecting semantics. See [6] for examples.

From the collecting semantics domain of sets of parse configurations, one defines an abstract interpretation by approximating a set of configurations by a finite set of finite configurations or by just a single configuration, say, written in regular-expression notation. This is developed in [6].

The resulting interpretation can be applied to a set of flow equations and solved with the usual least-fixed-point techniques. This yields *abstract parsing* of the strings generated by a script.

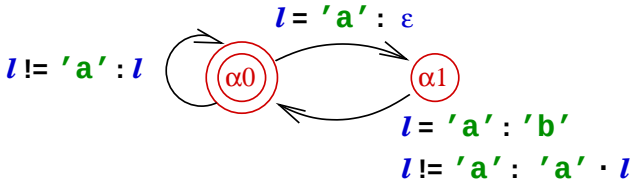
5 Abstract parsing with string-replacement operations

Recall that a parse configuration is a sequence of controller states topped off by a parse state of form, $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_1 \hookrightarrow s]$. The state is updated by the parse controller, which is a finite automaton.

Next, a script's string update operation, `replace`, e.g.,

```
y = replace 'aa' by 'b' in x
```

defines an automaton (more precisely, a *transducer*):



(We use $B : \ell$ to mean "take the transition if B holds true and emit letter ℓ as output".) Here is the linear encoding of the automaton's transitions:

$$\alpha_0(\mathtt{a}) \rightarrow \alpha_1/\epsilon$$
$$\alpha_0(\ell) \rightarrow \alpha_0/\ell, \text{ if } \ell \neq \mathtt{a}$$
$$\alpha_1(\mathtt{a}) \rightarrow \alpha_0/\mathtt{b}$$
$$\alpha_1(\ell) \rightarrow \alpha_0/\mathtt{a} \cdot \ell, \text{ if } \ell \neq \mathtt{a}$$

When a `replace` operation appears in a program, *the automaton defined by* `replace` *is composed with the parse-controller automaton* to consume the input stream. In effect, we generate a *new* parser to process the input.

From this assignment,

```
x = replace S1 by S2 in E
```

We generate this flow equation

$$X = replace_{\alpha} E$$

Where α names the finite automaton (transducer) generated from the string pattern, $S1$, and the replacement pattern, $S2$. When the above equation is called with a parse state, $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_1 \hookrightarrow s]$, we generate this first-order equation:

$$X[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_1 \hookrightarrow s] = erase_{\alpha}(E[\alpha_0, \ell_j \hookrightarrow \cdots \hookrightarrow \ell_1 \hookrightarrow s])$$

where α_0 is the start state of the α automaton that defines the string replacement.

The string generated from expression E is given to state α_0, which processes it and emits string output that is added to state s's input stream. For example, the operator, `replace 'b' by 'a' in Y`, generates this automaton, β:

$$\beta_0(\mathtt{b}) \rightarrow \beta_0/\mathtt{a}$$
$$\beta_0(\ell) \rightarrow \beta_0/\ell, \text{ if } \ell \neq \mathtt{b}$$

For this script and its flow equations,

```
y = 'b'
x = 'a'.(replace 'b' by 'a' in y)
```

$$Y = \mathtt{b}$$
$$X = \mathtt{a} \cdot replace_{\beta}(Y)$$

The abstract parse of $X \cdot \mathtt{!}$ would proceed like this:

$$
\begin{array}{ll}
(X \cdot \texttt{!})[s_0] & = X[s_0] \oplus \texttt{!} \\
X[s_0] & = (\texttt{a} \cdot \mathit{replace}_\beta(Y))[s_0] \\
& = \texttt{a}[s_0] \oplus \mathit{replace}_\beta(Y) \\
& = \mathit{replace}_\beta(Y)[\texttt{a} \hookrightarrow s_0] \\
& = \mathit{erase}_\beta(Y[\beta_0, \texttt{a} \hookrightarrow s_0]) \\
Y[\beta_0, \texttt{a} \hookrightarrow s_0] & = \texttt{b}[\beta_0, \texttt{a} \hookrightarrow s_0] \\
& = [\texttt{b} \hookrightarrow \beta_0, \texttt{a} \hookrightarrow s_0] \\
& = [\beta_0, \texttt{a} \hookrightarrow \texttt{a} \hookrightarrow s_0] \\
& = s_0 :: [\beta_0, \texttt{a} \hookrightarrow s_2]
\end{array}
$$

This shows how the string input is updated by the `replace` operator before the string is parsed. Once all of Y's string is processed, automaton β is erased from the compound parse state:

$$
\begin{array}{ll}
\mathit{erase}_\beta(Y[\beta_0, \texttt{a} \hookrightarrow s_0]) & = \mathit{erase}_\beta(s_0 :: [\beta_0, \texttt{a} \hookrightarrow s_2]) \\
& = s_0 :: [\texttt{a} \hookrightarrow s_2]
\end{array}
$$

Figure 3 displays a more complex example, where multiple string replacements compose with the parse-controller state.

There is a last, important, technical point: a string-replacement automaton must finish its work in a final state, e.g., for

```
y = replace 'aa' by 'b' in 'aaa'
```

whose automaton, α, uses α_0 as its final state, the string replacement of `'aaa'` causes α to finish in state α_1, implicitly holding the letter, `'a'` in its state. In this situation, the `'a'` must be "flushed" out; this is done by adding this last transition to α:

$$\alpha(\text{endOfString}) \rightarrow \alpha_0/\texttt{a}$$

This transition is enacted by the erase_α operation. Similar transitions are added to all non-final states in the automata generated from string-replacement operations.

Our embedding of string-replacement operations into the parse state lets us retain the existing least-fixed point machinery for computing the solutions to the a script's flow equations. So, it is perfectly acceptable to allow string replacements within loop bodies — this surmounts existing techniques [4, 5, 10], because we are *not* generating a new grammar to approximate and check.

6 Using string-replacement automata to implement conditional tests

One fundamental technique needed for implementing taint analysis [13, 14, 12] is implementing filter functions for the tests of conditional commands. For example,

```
read x
if isAllDigits(x) :
then ··· the analysis assumes that x holds all digits ···
```

```
x = 'a'
y = replace 'aa' by  'b' in x . x
z =  x . replace 'b' by 'a' in (y.'a')
print z !
```

$X = \mathtt{a}$
$Y = replace_\alpha(X \cdot X)$
$Z = X \cdot replace_\beta(Y \cdot \mathtt{a})$
$Z' = Z \cdot \mathtt{!}$

$$
\begin{array}{ll}
Z'[s_0] & = Z[s_0] \oplus \mathtt{!} \\
Z[s_0] & = X[s_0] \oplus (replace_\beta Y \cdot \mathtt{a}) = (replace_\beta Y \cdot \mathtt{a})[\mathtt{a} \hookrightarrow s_0] \\
 & = erase_\beta((Y \cdot \mathtt{a})[\beta_0, \mathtt{a} \hookrightarrow s_0] = erase_\beta(Y[\beta_0, \mathtt{a} \hookrightarrow s_0] \oplus \mathtt{a}) \\
Y[\beta_0, \mathtt{a} \hookrightarrow s_0] & = (replace_\alpha X \cdot X)[\beta_0, \mathtt{a} \hookrightarrow s_0] \\
 & = erase_\alpha((X \cdot X)[\alpha_0, \beta_0, \mathtt{a} \hookrightarrow s_0]) \\
(X \cdot X)[\alpha_0, \beta_0, \mathtt{a} \hookrightarrow s_0] & = X[\mathtt{a} \hookrightarrow \alpha_0, \beta_0, \mathtt{a} \hookrightarrow s_0] \\
 & = X[\alpha_1, \beta_0, \mathtt{a} \hookrightarrow s_0] = [\mathtt{a} \hookrightarrow \alpha_1, \beta_0, \mathtt{a} \hookrightarrow s_0] \\
 & = [\alpha_0, \mathtt{b} \hookrightarrow \beta_0, \mathtt{a} \hookrightarrow s_0] = [\alpha_0, \beta_0, \mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_0] \\
 & = s_0 :: [\alpha_0, \beta_0, \mathtt{a} \hookrightarrow s_2]
\end{array}
$$

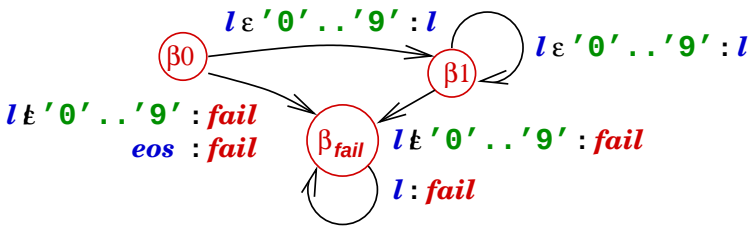
Hence,

$$
\begin{array}{ll}
Y[\beta_0, \mathtt{a} \hookrightarrow s_0] & = erase_\alpha(s_0 :: [\alpha_0, \beta_0, \mathtt{a} \hookrightarrow s_2]) = s_0 :: [\beta_0, \mathtt{a} \hookrightarrow s_2] \\
Z[s_0] & = erase_\beta(s_0 :: \mathtt{a}[\beta_0, \mathtt{a} \hookrightarrow s_2]) \\
 & = erase_\beta(s_0 :: [\mathtt{a} \hookrightarrow \beta_0, \mathtt{a} \hookrightarrow s_2] \\
 & = erase_\beta(s_0 :: [\beta_0, \mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_2] \\
 & = erase_\beta(s_0 :: s_2 :: [\beta_0, \mathtt{a} \hookrightarrow s_2] \\
 & = s_0 :: s_2 :: [\mathtt{a} \hookrightarrow s_2] \\
Z'[s_0] = (Z \cdot \mathtt{!})[s_0] & = s_0 :: s_2 :: \mathtt{!}[\mathtt{a} \hookrightarrow s_2] \\
 & = s_0 :: s_2 :: [\mathtt{!} \hookrightarrow \mathtt{a} \hookrightarrow s_2] \\
 & = \cdots \text{ see Figure 1} \cdots \\
 & = s_0 :: [\mathtt{!} \hookrightarrow s_4]
\end{array}
$$

Fig. 3. Compound string replacement and abstract parse

Think of the test expression, `isAllDigits(x)`, as an automaton (transducer) that reads the string contents of `x` and emits *failure* if a character of the input string is a nondigit. A failure causes the subsequent analysis to fail, too. In this fashion, the automaton acts as a "diode" or "filter function" that prevents non-digit string input from entering the conditional's body.

Here is the filter automaton for the test, `isAllDigits(x)`:



The filter automaton is a string-replacement automaton that emits *fail* when the input string does not satisfy the boolean test. The complement automaton, $\neg\beta$, merely swaps the outputs, ℓ and $fail$.

Our approach to analyzing conditional statements goes as follows:

For the conditional,	generate these flow equations:
`if B(x):`	$X_B = replace_\beta X$
`then` $\cdots$`x`$\cdots$	$\cdots X_B \cdots$
`else` $\cdots$`x`$\cdots$	$X_{\neg B} = replace_{\neg\beta} X$
	$\cdots X_{\neg B} \cdots$

where β is the automaton that implements test B and $\neg\beta$ implements $\neg B$.

The $fail$ character is special — when it is processed as an input, it causes the parse itself to denote $\bot$ (empty set in the powerset lattice):

$$[\cdots, fail, \cdots] = \bot$$

For example,

`x = 'a'`	$X0 =$ `a`
`if isAllDigits(x):`	$X1 = replace_\beta X0$
`  print x !`	$X2 = X1 \cdot$ `!`

and

$$\begin{array}{ll} X2[s_0] & = X1 \cdot \texttt{!}[s_0] = X1[s_0] \oplus \texttt{!} \\ X1[s_0] & = replace_\beta X0[s_0] = erase_\beta(X0[\beta_0, s_0]) \\ X0[\beta_0, s_0] & = \texttt{a}[\beta_0, s_0] = [\texttt{a} \hookrightarrow \beta_0, s_0] = [\beta_0, fail \hookrightarrow s_0] = \bot \end{array}$$

Hence,

$$\begin{array}{l} X1[s_0] = erase_\beta(\bot) = \bot \\ X2[s_0] = \bot \oplus \texttt{!} = \bot \end{array}$$

The analysis correctly predicts that nothing prints within the body of the conditional.

7 Modelling user input with nonterminals and unfolding

One of the advantages of our abstract parsing technique is that it can process a grammar's nonterminal symbol as input exactly the same way it processes terminal symbols as input: the symbol is supplied to the parse state, which can shift or reduce. For example, say that a module uses a string-valued global variable that is initialized outside of the module. If we can assume that the variable's value has the structure named by a nonterminal, then the global variable can be used in an abstract parse. For example, if and we assume global variable `g` holds an S-structured string, we can readily define the flow equations for this sequence,

`x = 'a'.g`	$G = S$
`print x !`	$X = \texttt{a} \cdot G$
	$X' = X \cdot$ `!`

and compute the abstract parse for $X'[s_0]$:

$$\begin{array}{ll}(\mathtt{a} \cdot G \cdot \mathtt{!})[s_0] & = G[\mathtt{a} \hookrightarrow s_0] \oplus \mathtt{!} \\ G[\mathtt{a} \hookrightarrow s_0] & = [S \hookrightarrow \mathtt{a} \hookrightarrow s_0] = s_0 :: [S \hookrightarrow s_2]\end{array}$$

Hence,

$$\begin{array}{rl}G[\mathtt{a} \hookrightarrow s_0] \oplus \mathtt{!} & = s_0 :: \mathtt{![}S \hookrightarrow s_2] = s_0 :: [\mathtt{!} \hookrightarrow S \hookrightarrow s_2] \\ & = s_0 :: s_2 :: [\mathtt{!} \hookrightarrow s_3] = [\mathtt{!} \hookrightarrow S \hookrightarrow s_0] \\ & = s_0 :: [\mathtt{!} \hookrightarrow s_4]\end{array}$$

In a similar way, user input, supplied via `read` commands, can be assumed to have structure named by a nonterminal, and abstract parsing can be undertaken:

`g = read`$_S()$	$G = S$
`x = 'a'.g`	$X = \mathtt{a} \cdot G$
`print x !`	$X' = X \cdot \mathtt{!}$

This proceeds just like the previous example. (Of course, we must supply a script that parses the input at runtime, to ensure that the input assumption is not violated.)

But there is a rub — say that the script includes string-replacement operations, which cannot process nonterminals. We solve this problem by generating the strings named by a nonterminal, and supplying the generated strings to the string-replacement automaton. Since the grammar is defined by a finite number of rules and there are a finite quantity of parse states, there are a finite number of reachable configurations to be analyzed in the abstract parse — the least-fixed-point semantics finitely solves the generated configurations. Here is an example:

`x = read`$_S()$	$X = S$
`y = replace 'aa' by 'a' in x`	$S = \mathtt{a} \cdot S \sqcup \mathtt{a}$
`print y !`	$Y = replace_\gamma X$
	$Y' = Y \cdot \mathtt{!}$

where automaton γ is defined,

$$\begin{array}{ll}\gamma_0(\mathtt{a}) = \gamma_1/\epsilon & \gamma_1(\ell) = \gamma_0/\mathtt{a} \cdot \ell, \text{ if } \ell \neq \mathtt{a} \\ \gamma_0(\ell) = \gamma_0/\ell, \text{ if } \ell \neq \mathtt{a} & \gamma_0(eos) = \gamma_0/\epsilon \\ \gamma_1(\mathtt{a}) = \gamma_0/\mathtt{a} & \gamma_1(eos) = \gamma_0/\mathtt{a}\end{array}$$

where γ_0 is the final state.

The analysis of the `print` command generates these first-order equations to solve:

$$\begin{array}{ll}Y'[s_0] & = Y[s_0] \oplus \mathtt{!} \\ Y[s_0] & = erase_\gamma(X[\gamma_0, s_0]) \\ X[\gamma_0, s_0] & = S[\gamma_0, s_0]\end{array}$$

The call to S generates these equations, which explain how to replace and parse all strings generated from nonterminal, S:

$$
\begin{aligned}
S[\gamma_0, s_0] &= (\mathtt{a} \cdot S)[\gamma_0, s_0] \cup \mathtt{a}[\gamma_0, s_0] = [\mathtt{a} \hookrightarrow \gamma_0, s_0] \oplus S \cup [\mathtt{a} \hookrightarrow \gamma_0, s_0] \\
&= [\gamma_1, s_0] \oplus S \cup [\gamma_1, s_0] \\
&= S[\gamma_1, s_0] \cup [\gamma_1, s_0] \\
S[\gamma_1, s_0] &= (\mathtt{a} \cdot S)[\gamma_1, s_0] \cup \mathtt{a}[\gamma_1, s_0] = S[\gamma_0, \mathtt{a} \hookrightarrow s_0] \cup [\gamma_0, \mathtt{a} \hookrightarrow s_0] \\
S[\gamma_0, \mathtt{a} \hookrightarrow s_0] &= S[\gamma_1, \mathtt{a} \hookrightarrow s_0] \cup [\gamma_1, \mathtt{a} \hookrightarrow s_0] \\
S[\gamma_1, \mathtt{a} \hookrightarrow s_0] &= [\gamma_0, \mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_0] \oplus S \cup [\gamma_0, \mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_0] \\
&= s_0 :: S[\gamma_0, \mathtt{a} \hookrightarrow s_2] \cup s_0 :: [\gamma_0, \mathtt{a} \hookrightarrow s_2] \\
S[\gamma_0, \mathtt{a} \hookrightarrow s_2] &= S[\gamma_1, \mathtt{a} \hookrightarrow s_2] \cup [\gamma_1, \mathtt{a} \hookrightarrow s_2] \\
S[\gamma_1, \mathtt{a} \hookrightarrow s_2] &= s_2 :: S[\gamma_0, \mathtt{a} \hookrightarrow s_2] \cup s_2 :: [\gamma_0, \mathtt{a} \hookrightarrow s_2]
\end{aligned}
$$

All reachable combinations of the string-replacement automaton and parse controller are generated. This completes the equation set, which is solved in the usual way.

The state explosion that is typical in such examples can be controlled by using SLR(k) or LALR(k) grammars to define string structure.

With the technique just illustrated, we can show the correctness of input-validation codings. For example, a script that goes

```
x = read_S()
if isAllDigits(x):
then···
```

can be analyzed with respect to the automaton defined by `isAllDigits` and this reference grammar:

$$
\begin{aligned}
S &::= C \mid CS \\
C &::= D \mid N \\
D &::= \mathtt{0} \cdots \mathtt{9} \\
N &::= \text{ all characters not in } D
\end{aligned}
$$

From here, it is only a small step to analyzing string-replacement and conditional-test automata to check for language inclusion, that is, all strings generated by a grammar nonterminal are accepted by the automaton.

8 Conclusion

We have demonstrated the applicability of LR(k) parsing and finite automata to static enforcement of correct dynamic string generation in scripts. The techniques described in this paper have been implemented are currently under evaluation.

Acknowledgements: We thank GTOne's CEO Soo-Yong Lee for inspiration and support and the anonymous referees for valuable suggestions and comments.

References

1. G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *Proc. Int'l. Conf. Software Maintenance, Oxford*, 1999.

2. C. Brabrand, A. Møller, and M.I. Schwartzbach. The <bigwig> project. *ACM Trans. Internet Technology*, 2, 2002.
3. T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In *Proc. Asian Symp. Prog. Lang. and Systems*, pages 374–388. Springer LNCS 4279, 2006.
4. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Static analysis for dynamic XML. In *Proc. PLAN-X-02*, 2002.
5. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Extending Java for high-level web service construction. *ACM TOPLAS*, 25, 2003.
6. K.-G. Doh, H. Kim, and D.A. Schmidt. Abstract parsing: static analysis of dynamically generated string output using lr-parsing technology. In *Proc. Static Analysis Symp.*, pages 256–272. Springer LNCS 5673, 2009.
7. E. Duesterwald, R. Gupta, and M.L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM TOPLAS*, 19:992–1030, 1997.
8. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engg.*, 1995.
9. N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proc. 13th Symp. POPL*, pages 296–306. ACM Press, 1986.
10. Y. Minamide. Static approximation of dynamically generated web pages. In *Proc. 14th ACM Int'l Conf. on the World Wide Web*, pages 432–441, 2005.
11. Y. Minimide and A. Tozawa. XML validation for context-free grammars. In *Proc. Asian Symp. Prog. Lang. and Systems*, pages 357–373. Springer LNCS 4279, 2006.
12. G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static checking of dymanically generated queries in database applications. *ACM Trans. Software Engineering and Methodology*, 16(4):14:1–27, 2007.
13. G. Wassermann and Z. Su. The essence of command injection attacks in web applications. In *Proc. 33d ACM Symp. POPL*, pages 372–382, 2006.
14. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. ACM PLDI*, pages 32–41, 2007.