Abstract parsing with string updates using LR-parsing technology

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

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

Abstract. We combine LR(k)-parsing technology and data-flow analysis to analyze, in advance of execution, the documents generated dynamically by a program. Based on the document language's context-free reference grammar and the program's control structure, formatted as a set of flow equations, the analysis *predicts* how the documents will be generated and simultaneously *parses* the predicted documents. Recursions in the flow equations cause the analysis to emit a set of residual equations that are solved by least-fixed point calculation in the domain of *abstract (folded) LR-parse stacks*.
Since the technique accommodates LR(k) grammars, it can also handle string-update operations in the programs by translating the updates into finite-state transducers, whose controllers are composed with the LR(k)-parser controller.

1 Motivation

Scripting languages like PHP, Perl, Ruby, and Python use strings as a "universal data structure" to communicate values, commands, and programs. For example, one might write a PHP script that assembles within a string variable an SQL query or an HTML page or an XML document.

Typically, the well-formedness of the assembled string is verified when the string is supplied as input to its intended processor (database, web browser, or interpreter), and an incorrectly assembled string might cause processor failure. Worse still, a malicious user might deliberately supply misleading input that generates a document that attempts a cross-site-scripting or injection attack.

As a first step towards preventing failures and attacks, the well-formedness of a dynamically generated, "grammatically structured" string (document) should be checked with respect to the document's context-free *reference grammar* (for SQL or HTML or XML) before the document is supplied to its processor. Better still, the document generator program *itself* should be analyzed to validate that all its generated documents are well formed with respect to the reference grammar, like an application program is type checked in advance of execution.

2 Motivating example

Say that a script must generate an output string that conforms to this grammar,

$$S \rightarrow \mathtt{a} \mid \mathtt{[}\, S \,\mathtt{]}$$

where S is the only nonterminal. (HTML, XML, and SQL are such bracket languages.) The grammar is LR(0), but it can be difficult to enforce even for simple programs, like the one in Figure 1, left column. Perhaps we require this program to print only well-formed S-phrases — the occurrence of

```
x = 'a'                    X0 = a
r = ']'                    R = ]
while ...                  X1 = X0 ⊔ X2
  x = '[' . x . r          X2 = [ · X1 · R
print x                    X3 = X1
```

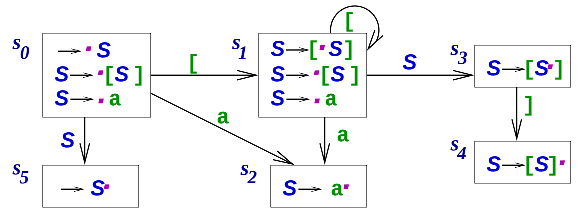
(Read `.` as an infix string-append operation.)

Fig. 1. Sample program and its data-flow equations

`x` at "`print x`" is a "hot spot" and we must analyze `x`'s possible values.

1. An analysis based on *type checking* assigns types (reference-grammar nonterminals) to the program's variables. The occurrences of `x` can indeed be data-typed as S, but `r` has no data type that corresponds to a nonterminal.
2. An analysis based on *regular expressions* (Christensen [2], Minamide [6], Wasserman [8]) solves flow equations shown in Figure 1's right column in the domain of regular expressions, determining that the hot spot's ($X3$'s) values conform to the regular expression, $\mathtt{[}^* \cdot \mathtt{a} \cdot \mathtt{]}^*$, but this does not validate the assertion.
3. A *grammar-based analysis* (Thiemann [7]) treats the flow equations as a set of grammar rules. A language-inclusion check based on Early's algorithm tries to prove that all $X3$-generated strings are S-generable.

Our approach solves the flow equations in the domain of *parse stacks* — $X3$'s meaning is the *set of LR-parses* of the strings that might be denoted by `x`. The technique "unfolds" the strings denoted by $X3$ at the same time that it executes the LR(k) parser. This is more precise than the techniques listed above.



Shift transitions:

$[\mathtt{[} \hookrightarrow s_0] \Rightarrow s_0 :: [s_1]$

$[S \hookrightarrow s_0] \Rightarrow [s_5]$

$[\mathtt{a} \hookrightarrow s_0] \Rightarrow s_0 :: [s_2]$

$[\mathtt{[} \hookrightarrow s_1] \Rightarrow s_1 :: [s_1]$

$[\mathtt{a} \hookrightarrow s_1] \Rightarrow s_1 :: [s_2]$

$[S \hookrightarrow s_1] \Rightarrow s_1 :: [s_3]$

$[\mathtt{]} \hookrightarrow s_3] \Rightarrow s_3 :: [s_4]$

Reduce transitions:

$s :: [s_2] \Rightarrow [S \hookrightarrow s]$

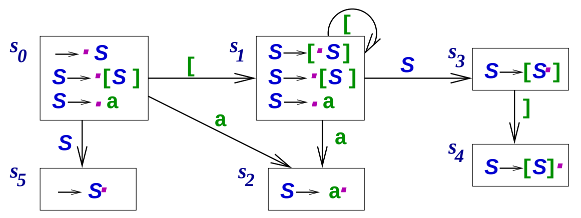
$s_i :: s_j :: s_k :: [s_4] \Rightarrow [S \hookrightarrow s_i]$

parse stack (top lies at right)	input sequence (front lies at left)
$[s_0]$	`[[a]]`
$[\mathtt{[} \hookrightarrow s_0]$	`[a]]`
$s_0 :: [s_1]$	`[a]]`
$s_0 :: [\mathtt{[} \hookrightarrow s_1]$	`a]]`
$s_0 :: s_1 :: [s_1]$	`a]]`
$s_0 :: s_1 :: [\mathtt{a} \hookrightarrow s_1]$	`]]`
$s_0 :: s_1 :: s_1 :: [s_2]$	`]]` (reduce: $S \rightarrow \mathtt{a}$)
$s_0 :: s_1 :: [S \hookrightarrow s_1]$	`]]`
$s_0 :: s_1 :: s_1 :: [s_3]$	`]]`
$s_0 :: s_1 :: s_1 :: [\mathtt{]} \hookrightarrow s_3]$	`]`
$s_0 :: s_1 :: s_1 :: s_3 :: [s_4]$	`]` (reduce: $S \rightarrow \mathtt{[}S\mathtt{]}$)
$s_0 :: [S \hookrightarrow s_1]$	`]`
$s_0 :: s_1 :: [s_3]$	`]`
$s_0 :: s_1 :: [\mathtt{]} \hookrightarrow s_3]$	
$s_0 :: s_1 :: s_3 :: [s_4]$	(reduce: $S \rightarrow \mathtt{[}S\mathtt{]}$)
$[S \hookrightarrow s_0]$	
$[s_5]$	(finished)

Fig. 2. Parse controller for $S \rightarrow \mathtt{[}S\mathtt{]} \mid \mathtt{a}$ and an example parse of `[[a]]`

```
x = 'a'
r = ']'
while ...
  x = '[' . x . r
print x
```

$X0 = \mathtt{a}$
$R = \mathtt{]}$
$X1 = X0 \sqcup X2$
$X2 = \mathtt{[} \cdot X1 \cdot R$
$X3 = X1$



To analyze the hot spot at $X3$, we must $LRparse(s_0, X3)$, which we portray as a function call, $X3[s_0]$. The program's flow equation, $X3 = X1$, generates this call step:

$$X3[s_0] = X1[s_0]$$

which demands a parse of the string generated at point $X1$ from state s_0:

$$X1[s_0] = X0[s_0] \cup X2[s_0]$$

The union of the parses from $X0$ and $X2$ must be computed. (*IMPORTANT:* this computes *sets of parse stacks*. In this example, all the sets are singletons.) Consider $X0[s_0]$:

$$X0[s_0] = [\mathtt{a} \hookrightarrow s_0] \Rightarrow s_0 :: [s_2] \Rightarrow [S \hookrightarrow s_0] \Rightarrow [s_5].$$

That is, a parse of `'a'` from s_0 generates s_5 — an S-phrase has been parsed. Finally,

$$\begin{aligned} X2[s_0] &= (\mathtt{[} \cdot X1 \cdot R)[s_0] = [\mathtt{[} \hookrightarrow s_0] \oplus (X1 \cdot R) \\ &\Rightarrow (s_0 :: [s_1]) \oplus (X1 \cdot R) \\ &= s_0 :: (X1 \cdot R)[s_1] = s_0 :: (X1[s_1] \oplus R) \end{aligned}$$

For parse stack, st, and function, E, define $st \oplus E = st :: E[top(st)]$. That is, stack st's top state feeds to E. Next, $X1[s_1] = X0[s_1] \cup X2[s_1]$ computes to $s_1 :: [s_3]$, and this means

$$\begin{aligned} X2[s_0] &= s_0 :: (X1[s_1] \oplus R) = (s_0 :: s_1 :: [s_3]) \oplus R = s_0 :: s_1 :: R[s_3] = s_0 :: s_1 :: [\mathtt{]} \hookrightarrow s_3] \\ &\Rightarrow s_0 :: s_1 :: s_3 :: [s_4] \Rightarrow [S \hookrightarrow s_0] \Rightarrow [s_5] \end{aligned}$$

That is, $X2[s_0]$ built the stack, $s_0 :: s_1 :: s_3 :: s_4$, denoting a parse of $\mathtt{[}\,S\,\mathtt{]}$, which reduced to S, giving s_5.

`x = 'a'`	$X0 = \mathtt{a}$
`r = ']'`	$R = \mathtt{]}$
`while ...`	$X1 = X0 \sqcup X2$
`  x = '[' . x . r`	$X2 = \mathtt{[} \cdot X1 \cdot R$
`print x`	$X3 = X1$

Here is the complete list of generated function calls; these are *residual equations* of a kind of *partial evaluation* [**?**] of the parser with the approximate string set, $X3$:

$$\begin{aligned}
&X3[s_0] = X1[s_0]\\
&X1[s_0] = X0[s_0] \cup X2[s_0]\\
&X0[s_0] = [s_5]\\
&X2[s_0] = s_0 :: (X1[s_1] \oplus R)\\
&X1[s_1] = X0[s_1] \cup X2[s_1]\\
&X0[s_1] = s_1 :: [s_3]\\
&X2[s_1] = s_1 :: (X1[s_1] \oplus R)\\
&R[s_3] \;\;\, = s_3 :: [s_4]
\end{aligned}$$

Each function call, $X_i[s_j] = E_{ij}$, is *a first-order equation*. The equations for $X1[s_1]$ and $X2[s_1]$ are mutually recursively defined, and their solutions are obtained by iteration-until-convergence.

The equation set is *generated dynamically while the equations are being solved*. (This is where $R[s_3]$ appears.) The solutions of the equations are

$$\begin{aligned}
&X3(s_0) = X1[s_0] = [s_5]\\
&X1[s_0] \;\, = X0[s_0] \cup X2[s_0] = [s_5] \cup [s_5] = [s_5]\\
&X0(s_0) = [s_5]\\
&X2[s_0] \;\, = s_0 :: (X1[s_1] \oplus R) \Rightarrow s_0 :: s_1 :: R[s_3] = s_0 :: s_1 :: s_3 :: [s_4] \Rightarrow [s_5]\\
&R[s_3] \;\;\;\, = s_3 :: [s_4]\\
&X1[s_1] \;\, = X0[s_1] \cup X2[s_1] = (s_1 :: [s_3]) \cup (s_1 :: [s_3]) = s_1 :: [s_3]\\
&X0[s_1] \;\, = s_1 :: [s_3]\\
&X2[s_1] \;\, = s_1 :: (X1[s_1] \oplus R) \Rightarrow s_1 :: s_1 :: R[s_3] \Rightarrow s_1 :: [s_3]
\end{aligned}$$

The solution is $X3[s_0] = [s_5]$, validating that the strings printed at the hot spot must be S-phrases. (Note: all of these stacks are really singleton sets.)

The algorithm is based on a demand-driven analysis [1, 3, 4], similar to *minimal function-graph semantics* [5], and is computed by a worklist algorithm.

Worklist,
added and processed
from top to bottom:

$X3[s_0]$
$X1[s_0]$
$X0[s_0]$
$X2[s_0]$
$X1[s_0]$
$X1[s_1]$
$X3[s_0]$
$X0[s_1]$
$X2[s_1]$
$X1[s_1]$
$X2[s_0]$
$X2[s_1]$
$R[s_3]$
$X2[s_0]$
$X2[s_1]$
$X1[s_0]$
$X1[s_1]$

Cache updates, inserted from top to bottom,
where $X[s] \mapsto P$ abbreviates $Cache[X(s)] := P$

$X3[s_0] \mapsto \emptyset$
$X1[s_0] \mapsto \emptyset$
$X0[s_0] \mapsto \emptyset$
$X2[s_0] \mapsto \emptyset$
$X0[s_0] \mapsto [\mathtt{a} \hookrightarrow s_0] \Rightarrow s_0 :: [s_2] \Rightarrow [S \hookrightarrow s_0] \Rightarrow [s_5].$
$X1[s_1] \mapsto \emptyset$
$X1[s_0] \mapsto X0[s_0] \cup X2[s_0] = [s_5]$
$X0[s_1] \mapsto \emptyset$
$X2[s_1] \mapsto \emptyset$
$X3[s_0] \mapsto X1[s_0] = [s_5]$
$X0[s_1] \mapsto [\mathtt{a} \hookrightarrow s_1] \Rightarrow s_1 :: [s_2] \Rightarrow [S \hookrightarrow s_1] \Rightarrow s_1 :: [s_3]$
$X1[s_1] \mapsto X0[s_1] \cup X2[s_1] = s_1 :: [s_3]$
$R[s_3] \mapsto \emptyset$
$R[s_3] \mapsto \mathtt{]} \hookrightarrow s_3] = s_3 :: [s_4]$
$X2[s_0] \mapsto (\mathtt{[} :: X1 :: R)[s_0] = s_0 :: (X1(s_1) \oplus R) = s_0 :: s_1 :: R[s_3]$
$\qquad = s_0 :: s_1 :: s_3 :: [s_4] \Rightarrow [S \hookrightarrow s_0] \Rightarrow [s_5]$
$X2[s_1] \mapsto (\mathtt{[} :: X1 :: R)[s_1] = \cdots = s_1 :: [s_3]$

Generated call graph:

$$\begin{tikzcd}
 & & X0(s_0) \arrow[dl] & & R(s_3) \arrow[ddll] \arrow[dd, bend left] \\
X3(s_0) & X1(s_0) \arrow[l] & & & X0(s_1) \arrow[dl] \\
 & & X2(s_0) \arrow[ul] & X1(s_1) \arrow[l] \arrow[r, bend right] & X2(s_1) \arrow[l]
\end{tikzcd}$$

Fig. 3. Worklist-algorithm calculation of call, $X3[s_0]$, in Figure 1

The initialization step places initial call, $X0[s_0]$, into the worklist and into the call graph and assigns to the cache the partial solution, $Cache[X0[s_0]] \mapsto \emptyset$. The iteration step repeats the following until the worklist is empty:

1. Extract a call, $X[s]$, from the worklist, and for the corresponding flow equation, $X = E$, compute $E[s]$, folding abstract stacks as necessary.
2. While computing $E[s]$, if a call, $X'[s']$ is encountered, *(i)* add the dependency, $X'[s'] \rightarrow X[s]$, to the call graph [if it is not already present]; *(ii)* if there is no entry for $X'[s']$ in the cache, then assign $Cache[X'[s']] \mapsto \emptyset$ and place $X'[s']$ on the worklist.
3. When $E[s]$ computes to an answer set, P, and P contains an abstract parse stack not already listed in $Cache[X[s]]$, then assign $Cache[X[s]] \mapsto [Cache[X[s]] \cup P]$ and add to the worklist all $X''[s'']$ such that the dependency, $X[s] \rightarrow X''[s'']$, appears in the flowgraph.

3 Abstract parse stacks

In the previous example, the result for each $X_i[s_j]$ was a single stack. In general, a set of parse stacks can result, e.g., for

```
x = '['
while ...
  x = x . '['
x = x . 'a' . ']'
```

$$\begin{aligned} X0 &= \texttt{[} \\ X1 &= X0 \sqcup X2 \\ X2 &= X1 \cdot \texttt{[} \\ X3 &= X1 \cdot \texttt{a} \cdot \texttt{]} \end{aligned}$$

at conclusion, `x` holds zero or more left brackets and an S-phrase; $X3[s_0]$ is the infinite set, $\{[s_5],\ s_1 :: [s_3],\ s_1 :: s_1 :: [s_3],\ s_1 :: s_1 :: s_1 :: [s_3],\ \cdots\}$.

To bound the set, we abstract it by "folding" its stacks so that no parse state repeats in a stack. Since Σ, the set of parse-state names, is finite, folding produces a finite set of finite-sized stacks (that contain cycles).

A stack segment like $p = s_1 :: [s_1]$ is a linked list, a graph, $\leftarrow s_1 \leftarrow s_1 \leftarrow$, where the stack's top and bottom are marked by pointers; when we push a state, e.g., $p :: [s_2]$, we get $\leftarrow s_1 \leftarrow s_1 \leftarrow s_2 \leftarrow$.

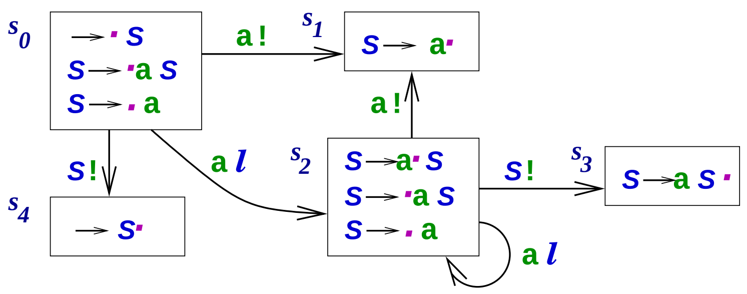
The folded stack is formed by merging same-state objects and retaining all links: $$\begin{tikzcd} {} & s_1 \arrow[l] \arrow[loop above] & s_2 \arrow[l] & {} \arrow[l] \end{tikzcd}$$. (This can be written as the regular expression, $s_1^+ :: [s_2]$.) Folding can apply to multiple states, e.g.,

$\leftarrow s_6 \leftarrow s_7 \leftarrow s_6 \leftarrow s_7 \leftarrow s_6 \leftarrow s_8 \leftarrow$ folds to $$\begin{tikzcd} {} & s_6 \arrow[l] \arrow[r, bend left] & s_7 \arrow[l] & s_8 \arrow[ll, bend left] & {} \arrow[l] \end{tikzcd}$$.

For the above example, $X3[s_0] = \{[s_5],\ s_1^+ :: [s_3]\}$.

4 LR(k) grammars are accommodated the same way

Parser for LR(1) grammar, $S \rightarrow \mathtt{a}S \mid \mathtt{a}$. Input and lookahead symbols are saved in the current state:



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

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)
$s_0 :: [\mathtt{a} \hookrightarrow s_2]$	!
$s_0 :: [! \hookrightarrow \mathtt{a} \hookrightarrow s_2]$	(do shift transition)
$s_0 :: s_2 :: [! \hookrightarrow s_1]$	(reduce $S \rightarrow \mathtt{a}$; pop one state and insert S)
$s_0 :: [! \hookrightarrow S \hookrightarrow s_2]$	(do shift transition)
$s_0 :: s_2 :: [! \hookrightarrow s_3]$	(reduce $S \rightarrow \mathtt{a}S$; pop two states and insert S)
$[! \hookrightarrow S \hookrightarrow s_0]$	(do final transition)
$s_0 :: [! \hookrightarrow s_4]$	(finished)

Fig. 4. An LR(k) grammar uses a state of form, $[\ell_k \hookrightarrow \ell_{k-1} \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$.

When a program is statically parsed with an LR(k) grammar, $k > 0$, the generated equations have form,

$$Xi[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] = E$$

for $0 < j < k$. Alas, this means a residual-equation set of order $(k+1)!$.

We mainly use the LR(k) technique to abstractly parse *string-replacement operators*, user input, and test expressions in conditional commands:

– 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. Since the transducer is a finite-state machine, we *compose its controller with the controller of the LR(k)-parser*. The compound controller analyzes the equations generated by the input program. We see this in an example to come.

– If unknown user input can be described by a nonterminal symbol, then that nonterminal can be input for abstract parsing — Our analyzer *treats grammar nonterminals as valid inputs*.
  For example, if the range of user inputs is contained within this syntax: $S ::= \mathtt{a} \mid \mathtt{a}S$,
  then the script on the left generates the equations on the right:

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

  When X is called with parse state, $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, the generated first-order equation is

$$S[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] = [S \hookrightarrow \ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$$

  which continues the parse with input symbol S. (Note: S is unfolded if it feeds directly into a string-replacement automaton.)

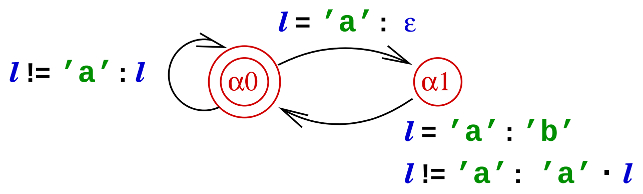
– Conditional tests on strings can often be expressed by finite automata or grammar nonterminals. We use the techniques mentioned above.

5 Abstract parsing with string-replacement operations

Recall that the current parse state has form, $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, $0 < j < k$. The state is updated by the parse controller, which is a finite automaton. A string update operation, 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".) 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. From the 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, $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, we generate this first-order equation:

$$X[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] = erase_\alpha(E[\alpha_0, \ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s])$$

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:

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

For this script and its flow equations,

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

$$\begin{aligned}Y &= \mathtt{b}\\ X &= \mathtt{a} \cdot \mathit{replace}_\beta(Y)\end{aligned}$$

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

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

Once all of Y's string is processed, automaton β is erased from the compound parse state:

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

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 [2, ?,6], because we are *not* generating a new grammar to approximate and check.

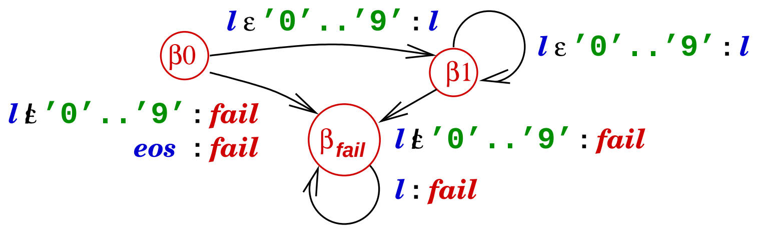
6 Using string-replacement automata to implement conditional tests

One fundamental technique needed for implementing taint analysis [9, 10, 8] 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 ···
```

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 ···x···`	$\cdots X_B \cdots$
`else ···x···`	$X_{\neg B} = replace_{\neg\beta} X$
	$\cdots X_{\neg B} \cdots$

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

and

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

Hence,

$$\begin{aligned} X1[s_0] &= erase_\beta(\bot) = \bot \\ X2[s_0] &= \bot \oplus \mathtt{!} = \bot \end{aligned}$$

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

7 Modelling user input with nonterminals and unfolding

Say that a module uses a string-valued global variable that is initialized outside of the module. If we assume the variable's value has the structure named by a nonterminal, then the global variable can be used in an abstract parse. For example, assume global variable `g` holds an S-structured string:

```
x = 'a'.g
print x !
```

$$\begin{aligned} G &= S \\ X &= \mathtt{a} \cdot G \\ X' &= X \cdot \mathtt{!} \end{aligned}$$

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

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

Hence,

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

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()
x = 'a'.g
print x !
```

$$\begin{aligned} G &= S \\ X &= \mathtt{a} \cdot G \\ X' &= X \cdot \mathtt{!} \end{aligned}$$

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 unfolding the nonterminal, supplying the generated strings to the string-replacement automaton:

```
x = read_S()
y = replace 'aa' by 'a' in x
print y !
```

$$\begin{aligned} X &= S \\ S &= \mathtt{a} \cdot S \sqcup \mathtt{a} \\ Y &= \mathit{replace}_\gamma X \\ Y' &= Y \cdot \mathtt{!} \end{aligned}$$

where automaton γ is defined,

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

where γ_0 is the final state.

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

$$\begin{aligned} Y'[s_0] &= Y[s_0] \oplus \mathtt{!} \\ Y[s_0] &= \mathit{erase}_\gamma(X[\gamma_0, s_0]) \\ X[\gamma_0, s_0] &= S[\gamma_0, s_0] \end{aligned}$$

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 Definitions of parsing and collecting semantics

An LR(k) *parse-stack configuration* is a sequence, $s_0 :: s_1 :: \cdots s_i :: [\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, $0 < j < k$, 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 4 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, $N \rightarrow U_1U_2 \cdots U_m$, 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,
    $[\ell_k \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] \rightarrow [\ell_k \hookrightarrow \cdots \hookrightarrow \ell_1 \hookrightarrow s']$,
  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:

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

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

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

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

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

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

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.

9 Conclusion

Injection and cross-site-scripting attacks can be reduced by analyzing the programs that dynamically generate documents [10]. In this paper, we have improved the precision of such analyses by employing LR-parsing technology to validate the context-free grammatical structure of generated documents.

A parse tree is but the first stage in calculating a string's meaning. The parsed string has a semantics (as enforced by its interpreter), and one can encode this semantics with semantics-processing functions, like those written for use with a parser-generator. (Tainting analysis — tracking unsanitized data — is a simplistic semantic property that is encoded this way.) The semantics can then be approximated by the static analysis so that abstract parsing and abstract semantic processing proceed simultaneously.

This talk is saved at `www.cis.ksu.edu/∼schmidt/papers/hometalks.html`

It is based on a paper published at the 2009 Static Analysis Symposium (Springer LNCS 5673) as well as an unpublished report. The paper and report are found at `www.cis.ksu.edu/∼schmidt/papers`

References

1. G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *Proc. Int'l. Conf. Software Maintenance, Oxford*, 1999.
2. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Static analysis for dynamic XML. In *Proc. PLAN-X-02*, 2002.
3. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Extending Java for high-level web service construction. *ACM TOPLAS*, 25, 2003.
4. 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 Symposium*. Springer LNCS 5673, 2009.
5. E. Duesterwald, R. Gupta, and M.L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM TOPLAS*, 19:992–1030, 1997.
6. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engg.*, 1995.
7. 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.
8. 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.
9. P. Thiemann. Grammar-based analysis of string expressions. In *Proc. ACM workshop Types in languages design and implementation*, pages 59–70, 2005.
10. 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.
11. G. Wassermann and Z. Su. The essence of command injection attacks in web applications. In *Proc. 33d ACM Symp. POPL*, pages 372–382, 2006.
12. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. ACM PLDI*, pages 32–41, 2007.