Abstract parsing: static analysis of dynamically generated string output 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

August 9, 2009

Outline

Overview

- Statically analyze the documents generated dynamically by a program
    - LR(k)-parsing technology + data-flow analysis
- Based on the document language's context-free reference grammar and the program's control structure, the analysis *predicts* how the documents will be generated and *parses* the predicted documents.
- Our strategy remembers context-free structure by computing *abstract LR-parse stacks*.
- The technique is implemented in Objective Caml and has statically validated a suite of PHP programs that dynamically generate HTML documents.

Problem

- Scripting languages like PHP, Perl, Ruby, and Python use strings as a “universal data structure” to communicate values, commands, and programs.
  - Example: 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.

Solution

- Dynamic solution
  - 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.
- Static solution
  - 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.

This Paper

- we employ LR(k)-parsing technology and data-flow analysis to *analyze* statically a program that dynamically generates documents as strings, and at the same time, *parse* the dynamically generated strings with the context-free reference grammar for the document language.
- We compute *abstract parse stacks* that remember the context-free structure of the strings.

Outline

Example

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

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

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

- Does an output string generated by the above script conform to this LR(0) grammar?

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

- Perhaps we require this program to print only well-formed S-phrases — the occurrence of `x` at "`print x`" is a "hot spot" and we must analyze `x`'s possible values.

Analysis based on Type Checking

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

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

- 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.

Analysis based on Regular Expressions

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

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

- An analysis based on *regular expressions* (works by Christensen, Møller, & Schwartzbach; Minamide; Wasserman & Su) solves flow equations shown above in the 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.

Grammar-based Analysis

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

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

- A *grammar-based analysis* (Thiemann [**?**]) treats the flow equations as a set of grammar rules. The “type” of `x` at the hot spot is $X3$. Next, a language-inclusion check tries to prove that all $X3$-generated strings are S-generable.

Our Approach

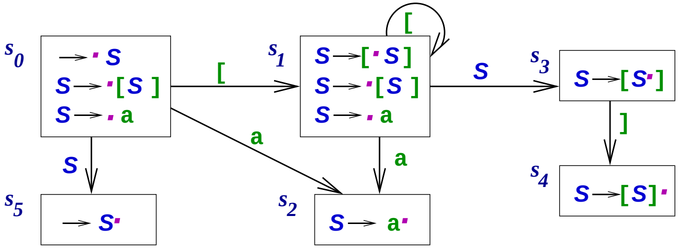
```
x = 'a'
r = ']'
while ...
  x = '[' . x . r
print x
```

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

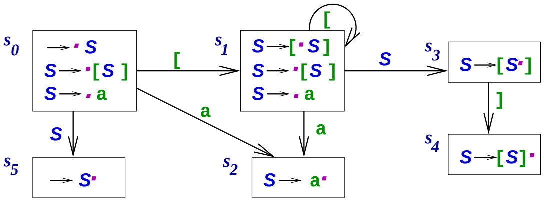
- Our approach solves the flow equations in the domain of *parse stacks*
- $X3$'s meaning is the *set of LR-parses* of the strings that might be denoted by `x`.

Parse Controller for $S \rightarrow [S] \mid \mathtt{a}$

- Calculate its LR-items
- Build its parse (“*goto*”) controller.



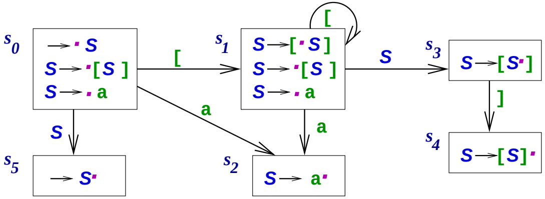
Parse of "[[a]]"



parse stack (top lies at right)	input sequence (front lies at left)	
s_0	[[a]]	
$s_0 :: s_1$	[a]]	(because $goto(s_0, [) = s_1$)
$s_0 :: s_1 :: s_1$	a]]	
$s_0 :: s_1 :: s_1 :: s_2$	]]	(reduce: $S \rightarrow$ a)
$s_0 :: s_1 :: s_1$	S]]	
$s_0 :: s_1 :: s_1 :: s_3$	]]	(because $goto(s_1, S) = s_3$)
$s_0 :: s_1 :: s_1 :: s_3 :: s_4$	]	(reduce: $S \rightarrow [S]$)
$s_0 :: s_1$	S]	
$s_0 :: s_1 :: s_3$	]	
$s_0 :: s_1 :: s_3 :: s_4$		(reduce: $S \rightarrow [S]$)
s_0	S	
$s_0 :: s_5$		(finished)

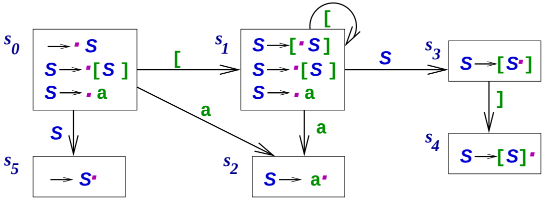
Abstract Parsing

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



- Interpret the flow equations as functions that map an input parse state to (a set of) output parse stacks.
- To analyze the hot spot at $X3$, we generate the function call, $X3(s_0)$, where s_0 is the start state for parsing an S-phrase.

Abstract Parsing



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

The flow equation, $X3 = X1$, generates

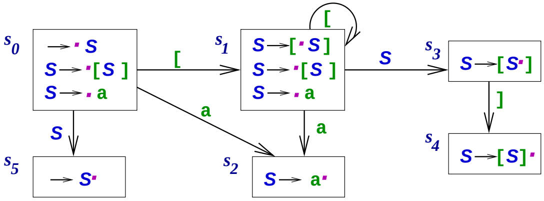
$$X3(s_0) = X1(s_0)$$

which itself 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.

Abstract Parsing



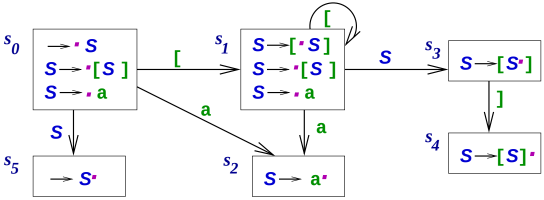
$X0 = \mathtt{a}$
$R = \mathtt{]}$
$X1 = X0 \sqcup X2$
$X2 = \mathtt{[} \cdot X1 \cdot R$
$X3 = X1$

Consider $X0(s_0)$:

$$\begin{aligned} X0(s_0) &= goto(s_0, \mathtt{a}) = s_2 \quad (\text{reduce}:\ S \rightarrow \mathtt{a}) \\ &\Rightarrow goto(s_0, S) = s_5 \end{aligned}$$

A parse of string `'a'` from s_0 generates s_2, a final state, that reduces to nonterminal S, which generates s_5 — an S-phrase has been parsed. (The $\Rightarrow$ signifies a *reduce* step to a nonterminal.) The completed stack is therefore $s_0 :: s_5$.

Abstract Parsing



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

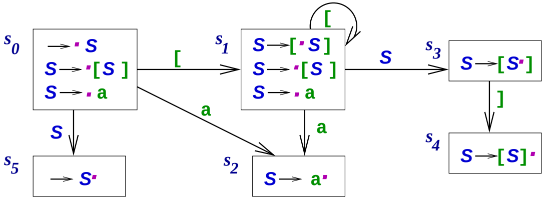
The remaining call, $X2(s_0)$, goes

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

The $\oplus$ operator sequences the parse steps: for parse stack, st, and function, E, $st \oplus E = st :: E(top(st))$, that is, the stack made by *appending* st to the stack returned by $E(top(st))$.

Abstract Parsing

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



Then, $X1(s_1) = X0(s_1) \cup X2(s_1)$ computes to s_3, and

$$\begin{aligned} X2(s_0) &= s_1 :: (X1(s_1) \oplus R) = s_1 :: (s_3 \oplus R) = s_1 :: s_3 :: R(s_3) \\ &= s_1 :: s_3 :: s_4 \quad (\text{reduce: } S \to [S]) \\ &\Rightarrow goto(s_0, S) = s_5 \end{aligned}$$

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

Abstract Parsing

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

Here is the complete list of solved function calls:

$$\begin{aligned}
X3(s_0) &= X1(s_0) \\
X1(s_0) &= X0(s_0) \cup X2(s_0) = \cdots = s_5 \cup s_5 = s_5 \\
X0(s_0) &= goto(s_0, \mathtt{a}) = s_2 \Rightarrow goto(s_0, S) = s_5 \\
X2(s_0) &= goto(s_0, \mathtt{[}) \oplus (X1 \cdot R) = s_1 :: X1(s_1) \oplus R \\
&= \cdots = s_1 :: s_3 :: R(s_3) = s_1 :: s_3 :: s_4 \Rightarrow goto(s_0, S) = s_5 \\
R(s_3) &= goto(s_3, \mathtt{]}) = s_4 \\
X1(s_1) &= X0(s_1) \cup X2(s_1) = \cdots = s_3 \cup s_3 = s_3 \\
X0(s_1) &= goto(s_1, \mathtt{a}) = s_2 \Rightarrow goto(s_1, S) = s_3 \\
X2(s_1) &= goto(s_1, \mathtt{[}) \oplus (X1 \cdot R) \\
&= s_1 :: (X1(s_1) \oplus R) = \cdots = s_1 :: s_3 :: R(s_3) \\
&= s_1 :: s_3 :: s_4 \Rightarrow goto(s_1, S) = s_3
\end{aligned}$$

The solution is $X3(s_0) = s_5$, validating that the strings printed at the hot spot must be S-phrases.

Abstract Parsing

$$\begin{aligned}
X3(s_0) &= X1(s_0) \\
X1(s_0) &= X0(s_0) \cup X2(s_0) = \cdots = s_5 \cup s_5 = s_5 \\
X0(s_0) &= goto(s_0, \texttt{a}) = s_2 \Rightarrow goto(s_0, S) = s_5 \\
X2(s_0) &= goto(s_0, \texttt{[}) \oplus (X1 \cdot R) = s_1 :: X1(s_1) \oplus R \\
&= \cdots = s_1 :: s_3 :: R(s_3) = s_1 :: s_3 :: s_4 \Rightarrow goto(s_0, S) = s_5 \\
R(s_3) &= goto(s_3, \texttt{]}) = s_4 \\
X1(s_1) &= X0(s_1) \cup X2(s_1) = \cdots = s_3 \cup s_3 = s_3 \\
X0(s_1) &= goto(s_1, \texttt{a}) = s_2 \Rightarrow goto(s_1, S) = s_3 \\
X2(s_1) &= goto(s_1, \texttt{[}) \oplus (X1 \cdot R) \\
&= s_1 :: (X1(s_1) \oplus R) = \cdots = s_1 :: s_3 :: R(s_3) \\
&= s_1 :: s_3 :: s_4 \Rightarrow goto(s_1, S) = s_3
\end{aligned}$$

- Each equation instance, $X_i(s_j) = E_{ij}$, is *a first-order data-flow equation*.

Abstract Parsing

$$
\begin{aligned}
X3(s_0) &= X1(s_0)\\
X1(s_0) &= X0(s_0) \cup X2(s_0) = \cdots = s_5 \cup s_5 = s_5\\
X0(s_0) &= goto(s_0, \mathtt{a}) = s_2 \Rightarrow goto(s_0, S) = s_5\\
X2(s_0) &= goto(s_0, \mathtt{[}) \oplus (X1 \cdot R) = s_1 :: X1(s_1) \oplus R\\
&= \cdots = s_1 :: s_3 :: R(s_3) = s_1 :: s_3 :: s_4 \Rightarrow goto(s_0, S) = s_5\\
R(s_3) &= goto(s_3, \mathtt{]}) = s_4\\
X1(s_1) &= X0(s_1) \cup X2(s_1) = \cdots = s_3 \cup s_3 = s_3\\
X0(s_1) &= goto(s_1, \mathtt{a}) = s_2 \Rightarrow goto(s_1, S) = s_3\\
X2(s_1) &= goto(s_1, \mathtt{[}) \oplus (X1 \cdot R)\\
&= s_1 :: (X1(s_1) \oplus R) = \cdots = s_1 :: s_3 :: R(s_3)\\
&= s_1 :: s_3 :: s_4 \Rightarrow goto(s_1, S) = s_3
\end{aligned}
$$

- Each equation instance, $X_i(s_j) = E_{ij}$, is *a first-order data-flow equation*.
- In the example, $X1(s_1)$ and $X2(s_1)$ are mutually recursively defined, and their solutions are obtained by iteration-until-convergence.

Abstract Parsing

$$
\begin{aligned}
X3(s_0) &= X1(s_0) \\
X1(s_0) &= X0(s_0) \cup X2(s_0) = \cdots = s_5 \cup s_5 = s_5 \\
X0(s_0) &= goto(s_0, \mathtt{a}) = s_2 \Rightarrow goto(s_0, S) = s_5 \\
X2(s_0) &= goto(s_0, \mathtt{[}) \oplus (X1 \cdot R) = s_1 :: X1(s_1) \oplus R \\
&= \cdots = s_1 :: s_3 :: R(s_3) = s_1 :: s_3 :: s_4 \Rightarrow goto(s_0, S) = s_5 \\
R(s_3) &= goto(s_3, \mathtt{]}) = s_4 \\
X1(s_1) &= X0(s_1) \cup X2(s_1) = \cdots = s_3 \cup s_3 = s_3 \\
X0(s_1) &= goto(s_1, \mathtt{a}) = s_2 \Rightarrow goto(s_1, S) = s_3 \\
X2(s_1) &= goto(s_1, \mathtt{[}) \oplus (X1 \cdot R) \\
&= s_1 :: (X1(s_1) \oplus R) = \cdots = s_1 :: s_3 :: R(s_3) \\
&= s_1 :: s_3 :: s_4 \Rightarrow goto(s_1, S) = s_3
\end{aligned}
$$

- The flow-equation set is *generated dynamically while the equations are being solved.*

Abstract Parsing

$$\begin{aligned}
X3(s_0) &= X1(s_0) \\
X1(s_0) &= X0(s_0) \cup X2(s_0) = \cdots = s_5 \cup s_5 = s_5 \\
X0(s_0) &= goto(s_0, \mathtt{a}) = s_2 \Rightarrow goto(s_0, S) = s_5 \\
X2(s_0) &= goto(s_0, \mathtt{[}) \oplus (X1 \cdot R) = s_1 :: X1(s_1) \oplus R \\
&= \cdots = s_1 :: s_3 :: R(s_3) = s_1 :: s_3 :: s_4 \Rightarrow goto(s_0, S) = s_5 \\
R(s_3) &= goto(s_3, \mathtt{]}) = s_4 \\
X1(s_1) &= X0(s_1) \cup X2(s_1) = \cdots = s_3 \cup s_3 = s_3 \\
X0(s_1) &= goto(s_1, \mathtt{a}) = s_2 \Rightarrow goto(s_1, S) = s_3 \\
X2(s_1) &= goto(s_1, \mathtt{[}) \oplus (X1 \cdot R) \\
&= s_1 :: (X1(s_1) \oplus R) = \cdots = s_1 :: s_3 :: R(s_3) \\
&= s_1 :: s_3 :: s_4 \Rightarrow goto(s_1, S) = s_3
\end{aligned}$$

- The flow-equation set is *generated dynamically while the equations are being solved.*
- This is a demand-driven analysis (Agrawal 99, Duesterwald 97, Horwitz 95), called *minimal function-graph semantics* (Jones & Mycroft 86), computed by a worklist algorithm.

Worklist-algorithm calculation of call, $X3(s_0)$

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 $Cache[X(s)] := P$

$$
\begin{aligned}
&X3(s_0) \mapsto \emptyset\\
&X1(s_0) \mapsto \emptyset\\
&X0(s_0) \mapsto \emptyset\\
&X2(s_0) \mapsto \emptyset\\
&X0(s_0) \mapsto reduce(s_0, goto(s_0, \texttt{a})) = reduce(s_0, s_2)\\
&\quad = reduce(s_0, goto(s_0, S)) = reduce(s_0, s_5) = \{s_5\}\\
&X1(s_1) \mapsto \emptyset\\
&X1(s_0) \mapsto \{s_5\}\\
&X0(s_1) \mapsto \emptyset\\
&X2(s_1) \mapsto \emptyset\\
&X3(s_0) \mapsto \{s_5\}\\
&X0(s_1) \mapsto reduce(s_1, goto(s_1, \texttt{a})) = \{s_3\}\\
&X1(s_1) \mapsto \{s_3\}\\
&R(s_3) \mapsto \emptyset\\
&R(s_3) \mapsto reduce(s_3, goto(s_3, \texttt{]})) = \{s_4\}\\
&X2(s_0) \mapsto (\texttt{[} :: X1 :: R)(s_0)\\
&\quad = s_1 \oplus (X1 :: R) = (s_1 :: X1(s_1)) \oplus R\\
&\quad = s_1 :: s_3 :: R(s_3) = reduce(s_0, s_1 :: s_3 :: s_4)\\
&\quad = reduce(s_0, goto(s_0, S)) = \{s_5\}\\
&X2(s_1) \mapsto (\texttt{[} :: X1 :: R)(s_1) = \{s_3\}
\end{aligned}
$$

Generated call graph:

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

Worklist-algorithm calculation of call, $X3(s_0)$

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.

Outline

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' . ']'
```

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

  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).

Abstract Parse Stacks

The abstract interpretation based on abstract, folded stacks can be calculated; here is the intuition:

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

Abstract Parse Stacks

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

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

The abstract interpretation of the above loop program is defined with abstract stacks in the next slide.

Iterative solution with folded parse stacks

Flow equation set generated from demand, $X3(s_0)$:

$$X0(s_0) = \texttt{[}(s_0) \qquad X2(s_0) = X1(s_0) \oplus \texttt{[}$$
$$X1(s_0) = X0(s_0) \cup X2(s_0) \qquad X3(s_0) = X1(s_0) \oplus (\texttt{a.]})$$

Least fixed-point solution expressed with abstract parse stacks:

$X0(s_0) \;=\; \texttt{[}(s_0) \;=\; \{s_1\}$

Because $X1$ and $X2$ are mutually defined, we iterate to a solution, where Xi's value at iteration j is denoted Xi_j:

$X1_1(s_0) \;=\; \{s_1\} \cup \emptyset \;=\; \{s_1\}$

$X2_1(s_0) \;=\; X1_1(s_0) \oplus \texttt{[} \;=\; fold\{s_1 :: s_1\} \;=\; \{s_1^+\}$

$X1_2(s_0) \;=\; \{s_1\} \cup \{s_1^+\} \;=\; \{s_1, s_1^+\}$

$= \{s_1^+\}$. (We can merge the two stack segments since the first is a prefix of the second and has the same bottom and top states.)

$X2_2(s_0) \;=\; X1_2(s_0) \oplus \texttt{[} \;=\; \{s_1^+ :: \texttt{[}(s_1)\} \;=\; fold\{s_1^+ :: s_1\} \;=\; \{s_1^+\}$

$X1_3(s_0) \;=\; \{s_1\} \cup \{s_1^+\} \;=\; \{s_1, s_1^+\} \;=\; \{s_1^+\} \;=\; X1_2(s_0)$

$X2_3(s_0) \;=\; \{s_1^+\} \;=\; X2_2(s_0)$

$X3(s_0) \;=\; \{s_1^+ :: \texttt{a}(s_1) \oplus \texttt{]}\}$

First, $s_1^+ :: \texttt{a}(s_1) \;=\; s_1^+ :: s_2 \;\Rightarrow\; s_1^+ :: goto(s_1, S) \;=\; s_1^+ :: s_3$.

$= \{s_1^+ :: s_3 :: \texttt{]}(s_3)\} \;=\; \{s_1^+ :: s_3 :: s_4\}$

The reduction, $S \longrightarrow [S]$, splits the stack into two cases:
(i) there are multiple s_1s within s_1^+; (ii) there is only one s_1:

$= (i)\{s_1^+ :: goto(s_1, S)\} \;\cup\; (ii)\{goto(s_0, S)\}$

$= \{s_1^+ :: s_3, \quad s_5\}$

Abstract Parse Stacks

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

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

Flow equation set generated from demand, $X3(s_0)$:

$$X0(s_0) = \texttt{[}(s_0) \qquad X2(s_0) = X1(s_0) \oplus \texttt{[}$$
$$X1(s_0) = X0(s_0) \cup X2(s_0) \qquad X3(s_0) = X1(s_0) \oplus (\texttt{a.]})$$

The result, $X3(s_0) = \{s_1^+ :: s_3,\ s_5\}$, asserts that

- the string at $X3$ might be a well-formed S phrase, or
- it might contain a surplus of unmatched left brackets.

Abstract Parse Stacks

Least fixed-point solution expressed with abstract parse stacks:

$X3(s_0) \quad = \{s_1^+ :: \mathtt{a}(s_1) \oplus \mathtt{]}\}$

First, $s_1^+ :: \mathtt{a}(s_1) \;=\; s_1^+ :: s_2 \;\Rightarrow\; s_1^+ :: goto(s_1, S) \;=\; s_1^+ :: s_3$.

$= \{s_1^+ :: s_3 :: \mathtt{]}(s_3)\} \;=\; \{s_1^+ :: s_3 :: s_4\}$

The reduction, $S \longrightarrow [S]$, splits the stack into two cases:

(i) there are multiple s_1s within s_1^+; (ii) there is only one s_1:

$= (i)\{s_1^+ :: goto(s_1, S)\} \cup (ii)\{goto(s_0, S)\}$

$= \{s_1^+ :: s_3, \quad s_5\}$

- The reduction of $S \longrightarrow [S]$ is done on $s_1^+ :: s_3 :: s_4$.
- That is, the complete stack is $\begin{tikzcd} s_0 & s_1 \arrow[l] \arrow[loop above] & s_3 \arrow[l] & s_4 \arrow[l] & {} \arrow[l] \end{tikzcd}$, meaning that three states must be popped: we traverse s_4, s_3, and s_1, and follow the links from the last state, s_1, to see what the remaining stack might be.
- There are two possibilities: $\begin{tikzcd} s_0 & s_1 \arrow[l] \arrow[loop above] & {} \arrow[l] \end{tikzcd}$ and $\begin{tikzcd} s_0 & {} \arrow[l] \end{tikzcd}$.
- We compute the result for each case, as shown in the above Figure.

Outline

String-update Operations

- String-manipulating languages use operations like `replace` and `substring`, which can be employed foolishly or sensibly.
- An example of the former is

  ```
  x = '[[a]]';
  replace('a', '[', x)
  ```

  which replaces occurrences of `'a'` in `x` by `'['`, changing `x`'s value to the grammatically ill-formed phrase, `'[[[]]'`.
- A more sensible replacement would be

  ```
  replace('[a]', 'a', x)
  ```

  which preserves `x`'s grammatical structure.

String-update Operations

- To validate an operation, `replace(U,V,x)`, we require that `U` and `V` "parse the same" in every possible context where they might appear (within `x`):
- Say that `replace(U,V,x)` is *update-invariant for* `x` iff for all (nonfinal) parse states, $s \in \Sigma$, $U(s) = V(s)$.
- This means replacing `U` by `V` preserves `x`'s parse.

String-update Operations

- When we analyze a program, we may first *ignore* the `replace` operations, treating them as “no-ops.”
- Once the flow equations are solved, we validate the invariance of each `replace(U,V,x)` by generating hot-spot requests for strings `U` and `V` for all possible parse states, building on the cached results of the worklist algorithm.
- Finally, we compare the results to see if `replace(U,V,x)` is update-invariant for `x`.

String-update Operations

Here is an example:

```
y = '[[[a]]]'
x = 'a'
while ...
  x = '['.  x .']'
replace(x, 'a', y)
```

$$Y0 = \texttt{[} \cdot \texttt{[} \cdot \texttt{[} \cdot \texttt{a} \cdot \texttt{]} \cdot \texttt{]} \cdot \texttt{]}$$
$$X0 = \texttt{a}$$
$$X1 = X0 \cup X2$$
$$X2 = \texttt{[} \cdot X1 \cdot \texttt{]}$$
$$Y1 = \mathit{replace}(X1, \texttt{a}, Y0)$$

- Say that the program must be analyzed for `y`'s final value: $Y1(s_0)$.
- We initially ignore the replacement operation at $Y1$ and solve the simpler equation, $Y1(s_0) = Y0(s_0)$, instead, which quickly computes to $\{s_5\}$.
- Next, we analyze the `replace` operation by generating these hot-spot requests for all the nonfinal parse states:

$$a(s_0),\ X1(s_0),\ a(s_1),\ X1(s_1),\ a(s_3),\ X1(s_3)$$

String-update Operations

```
y = '[[[a]]]'
x = 'a'
while ...
  x = '['.  x .']'
replace(x, 'a', y)
```

$$
\begin{aligned}
Y0 &= [\cdot[\cdot[\cdot \mathtt{a} \cdot]\cdot]\cdot] \\
X0 &= \mathtt{a} \\
X1 &= X0 \cup X2 \\
X2 &= [\cdot X1 \cdot] \\
Y1 &= \mathit{replace}(X1, \mathtt{a}, Y0)
\end{aligned}
$$

- For example, the first request computes to
$$a(s_0) = goto(s_0, a) = s_2 \Rightarrow goto(s_0, S) = s_5$$
and the second repeats an earlier example,
$$\begin{aligned} X1(s_0) &= X0(s_0) \cup X2(s_0) \\ X2(s_0) &= \cdots = s_1 :: s_3 :: s_4 \Rightarrow goto(s_0, S)) = s_5 \end{aligned}$$
showing that both strings compute to the same parse-stack segments in starting context s_0.
- The other hot spots compute this same way.
- Once all the hot spots are solved, we confirm that $X1$ and a have identical outcomes for all possible parse contexts.
- This validates the invariance of `replace(x,'a',y)` at $Y1$.

String-update Operations

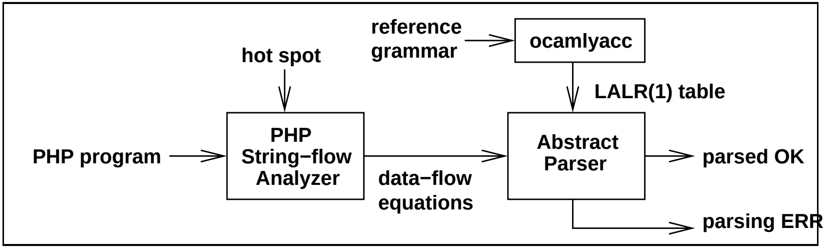
- It is important that we validate update-invariance for *all* possible contexts.
- Consider the reference grammar,

$$N \longrightarrow \texttt{a} \mid \texttt{b} \mid \texttt{[a]}$$

- Although both `a` and `b` are N-phrases, `replace('a','b','[a]')` violates `[a]`'s grammatical structure.

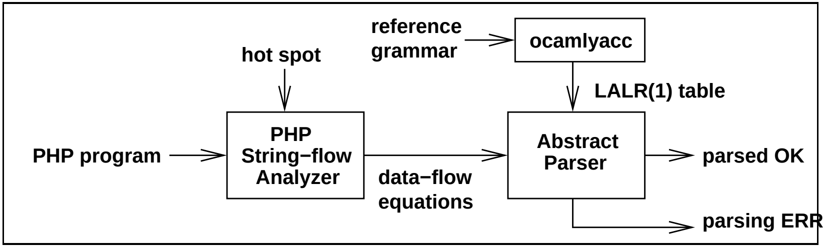
Outline

Implementation



- The implementation is in Objective Caml.
- The front end of Minamide's analyzer for PHP was modified to accept a PHP program with a hot-spot location and to return data-flow equations with string operations for the hot spot.
- `ocamlyacc` produces an LALR(1) parsing table, and the abstract parser uses the data-flow equations and the parsing table to parse statically the strings generated by the PHP program.

Implementation



- Abstract parsing works directly on characters (not tokens), so the reference grammar is written for scannerless parsing.
- Performance was good enough for practical use.

Outline

Experiments

- We applied our tool to a suite of PHP programs that dynamically generate HTML documents, the same one studied by Minamide, using a MacOSX with an Intel Core 2 Duo Processor (2.56GHz) and 4 GByte memory:

	webchess	faqforge	phpwims	timeclock	schoolmate
files	21	11	30	6	54
lines	2918	1115	6606	1006	6822
no. of hot spots	6	14	30	7	1
no. of parsings	6	16	36	7	19
parsed OK	5	1	19	0	1
parsed ERR	1	15	17	7	18
no. of alarms	1	31	16	14	20
true positives	1	31	13	14	17
false positives	0	0	3	0	3
time(sec)	0.224	0.155	1.979	0.228	2.077

Experiments

- We manually identified the hot spots and ran our abstract parser for each hot spot.
- Since we do not yet have parse-error recovery, each time a parse error was identified by our analyzer, we located the source of the error, fixed it, and tried again until no parse errors were detected.

	webchess	faqforge	phpwims	timeclock	schoolmate
files	21	11	30	6	54
lines	2918	1115	6606	1006	6822
no. of hot spots	6	14	30	7	1
no. of parsings	6	16	36	7	19
parsed OK	5	1	19	0	1
parsed ERR	1	15	17	7	18
no. of alarms	1	31	16	14	20
true positives	1	31	13	14	17
false positives	0	0	3	0	3
time(sec)	0.224	0.155	1.979	0.228	2.077

Experiments

- All the false-positive alarms were caused by ignoring the tests within conditional commands.
- The parsing time shown in the table is the sum of all execution times needed to find all parsing errors for all hot spots.
- The reference grammar's parse table took 1.323 seconds to construct; this is not included in the analysis times.

	webchess	faqforge	phpwims	timeclock	schoolmate
files	21	11	30	6	54
lines	2918	1115	6606	1006	6822
no. of hot spots	6	14	30	7	1
no. of parsings	6	16	36	7	19
parsed OK	5	1	19	0	1
parsed ERR	1	15	17	7	18
no. of alarms	1	31	16	14	20
true positives	1	31	13	14	17
false positives	0	0	3	0	3
time(sec)	0.224	0.155	1.979	0.228	2.077

Experiments

- The alarms are classified below:

classification	occurrences
open/close tag syntax error	11
open/close tag missing	45
superfluous tag	5
improperly nested	14
misplaced tag	5
escaped character syntax error	2

- All in all, our abstract parser
  - works without limiting the nesting depth of tags,
  - validates the syntax reasonably fast, and
  - is guaranteed to find all parsing errors reducing inevitable false alarms to a minimum.

Experiments

- Minamide excluded one PHP application, named `tagit`, from his experiments, since `tagit` generates an arbitrary nesting depth of tags.
- In principle, our abstract parser should be able to validate `tagit`, but we also excluded `tagit` from our studies because the current version of our abstract parser checks that string-update operations satisfy the update-invariance property.
- Unexpectedly (to us!), so many string updates in `tagit` violated update invariance that our abstract parser generated too many false-positives to be helpful.

Experiments

- We can reduce false positives due to violation of update invariance by selectively employing Minamide's f.s.a.-transducer technique, where a string update is analyzed separately from the flow analysis with its own f.s.a. transducer.
- For example, the last flow equation in this program,

```
x = 'a'
while ...
  x = '[['.  x .']'
replace('[[', '[', x)
```

$$
\begin{aligned}
X0 &= \mathtt{a} \\
X1 &= X0 \cup X2 \\
X2 &= \mathtt{[} \cdot \mathtt{[} \cdot X1 \cdot \mathtt{]} \\
X3 &= \mathit{replace}(\mathtt{[[}, \mathtt{[}, X1)
\end{aligned}
$$

  could be replaced by just $X3 = X1$, and we would use a separate transducer to analyze $\mathit{replace}(\mathtt{[[}, \mathtt{[}, X1)$.
- We leave this as a future work.

Experiments

- On the other hand, one might argue that *any* string-update operator that violates update invariance is dubiously employed and deserves closer scrutiny.
- In this regard, the abstract parser's “false positives” are healthy warnings.

Outline

Conclusion

- Injection and cross-site-scripting attacks can be reduced by analyzing the programs that dynamically generate documents (Ref: A series of works by Wassermann & Su).
- 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.

Future Work

- 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 an example semantic property that can be encoded this way.
- The semantics can then be approximated by the static analysis so that abstract parsing and abstract semantic processing proceed simultaneously.