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

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, 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.

1 Introduction

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.

In 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.

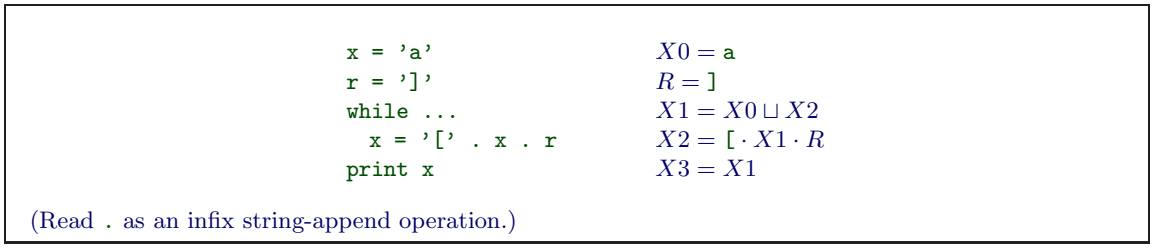


Fig. 1. Sample program and its data-flow equations

2 Motivating example

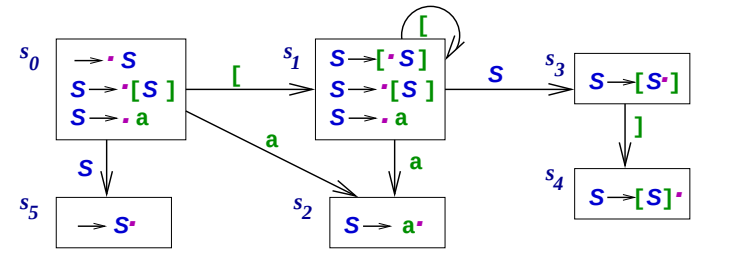
Say that a script must generate an output string that conforms to this grammar,

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

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

- 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.
- 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, $\texttt{[}^* \cdot \texttt{a} \cdot \texttt{]}^*$, but this does not validate the assertion.
- A *grammar-based analysis* (Thiemann [7]) 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 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`.



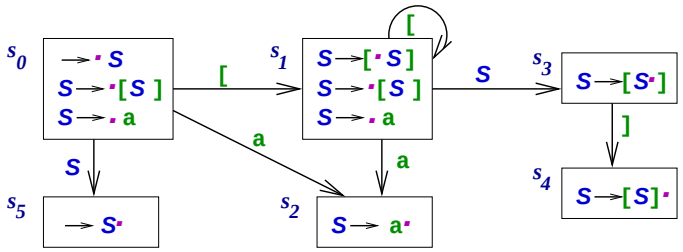
parse stack (top lies at right)	input sequence (front lies at left)	
s_0	`[[a]]`	
$s_0 :: s_1$	`[a]]`	(because $goto(s_0, \texttt{[}) = s_1$)
$s_0 :: s_1 :: s_1$	`a]]`	
$s_0 :: s_1 :: s_1 :: s_2$	`]]`	(reduce: $S \to \texttt{a}$)
$s_0 :: s_1 :: s_1$	S `]]`	
$s_0 :: s_1 :: s_1 :: s_3$	`]]`	(because $goto(s_1,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

Fig. 2. *goto* controller for $S \to \texttt{[}S\texttt{]} \mid \texttt{a}$ and an example parse of `[[a]]`

Assume that the reference grammar is LR(k); we first calculate its LR-items and build its parse ("*goto*") controller.

We interpret the flow equations in Figure 1 as *functions* that map an input parse state to (a set of) output parse stacks.

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



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. 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.[3] Consider $X0(s_0)$:

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

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 $s_0 :: s_5$. The remaining call, $X2(s_0)$, goes

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

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

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

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.

[3] In general, the functions compute sets of parse stacks. In this example, all the sets are singletons.

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

Here is the complete list of solved function calls:

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

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

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. The flow-equation set is *generated dynamically while the equations are being solved*. This is a demand-driven analysis [1, 3, 4], called *minimal function-graph semantics* [5], computed by a worklist algorithm.

Worklist, added and processed from top to bottom:	Cache updates, inserted from top to bottom, where $X(s) \mapsto P$ abbreviates $Cache[X(s)] := P$
	$X3(s_0) \mapsto \emptyset$
	$X1(s_0) \mapsto \emptyset$
$X3(s_0)$	$X0(s_0) \mapsto \emptyset$
$X1(s_0)$	$X2(s_0) \mapsto \emptyset$
$X0(s_0)$	$X0(s_0) \mapsto reduce(s_0, goto(s_0, \mathtt{a})) = reduce(s_0, s_2)$
$X2(s_0)$	$= reduce(s_0, goto(s_0, S)) = reduce(s_0, s_5) = \{s_5\}$
$X1(s_0)$	$X1(s_1) \mapsto \emptyset$
$X1(s_1)$	$X1(s_0) \mapsto \{s_5\}$
$X3(s_0)$	$X0(s_1) \mapsto \emptyset$
$X0(s_1)$	$X2(s_1) \mapsto \emptyset$
$X2(s_1)$	$X3(s_0) \mapsto \{s_5\}$
$X1(s_1)$	$X0(s_1) \mapsto reduce(s_1, goto(s_1, \mathtt{a})) = \{s_3\}$
$X2(s_0)$	$X1(s_1) \mapsto \{s_3\}$
$X2(s_1)$	$R(s_3) \mapsto \emptyset$
$R(s_3)$	$R(s_3) \mapsto reduce(s_3, goto(s_3, \mathtt{]})) = \{s_4\}$
$X2(s_0)$	$X2(s_0) \mapsto (\mathtt{[} :: X1 :: R)(s_0)$
$X2(s_1)$	$= s_1 \oplus (X1 :: R) = (s_1 :: X1(s_1)) \oplus R$
$X1(s_0)$	$= s_1 :: s_3 :: R(s_3) = reduce(s_0, s_1 :: s_3 :: s_4)$
$X1(s_1)$	$= reduce(s_0, goto(s_0, S)) = \{s_5\}$
	$X2(s_1) \mapsto (\mathtt{[} :: X1 :: R)(s_1) = \{s_3\}$

Generated call graph:

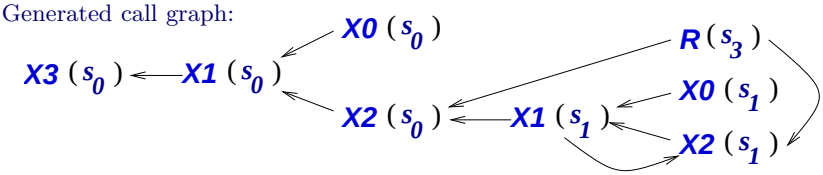


Fig. 3. Worklist-algorithm calculation of call, $X3(s_0)$, in Figure 1

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.

Concrete semantics: A source program computes a store that maps variables to strings. The *concrete collecting semantics* computes a set of stores for each program point; the collecting semantics is then abstracted so that it computes, for each program point, a single store that maps each variable to a set of strings.

The collecting semantics is overapproximated by the *data-flow semantics*, which uses flow equations to compute the set of strings denoted by each variable at each program point. In Figure 1, the data-flow semantics computes these values of variable `x` at the program points:

$$X0 = \{\texttt{a}\} \quad X2 = \{\texttt{[}s_1\texttt{]} \mid s_1 \in X1\} \quad R = \{\texttt{]}\} \quad X1 = X0 \cup X2 = X3$$

Let Σ name the states in the parser's *goto*-controller. A parse stack, $st \in \Sigma^+$, models those strings that parse to st. Function $\gamma : \mathcal{P}(\Sigma^+) \to \mathcal{P}(String)$ concretizes a set of parse stacks into a set of strings:

$$\gamma(S) = \{t \in String \mid s_0 :: s_1 :: \cdots :: s_k \in S \text{ and } parse(s_0, t) = s_0 :: s_1 :: \cdots :: s_k\}$$

The *abstract collecting interpretation*, $\mathcal{X}$, computes the set of parse stacks denoted by a program variable. For flow equation, $Xi = E_i$, the function, $\mathcal{X}_i : \Sigma \to \mathcal{P}(\Sigma^*)$, is defined as $\mathcal{X}_i(s) = [\![E_i]\!](s)$, where $s \in \Sigma$ and

$[\![\texttt{t}]\!]s = \{reduce(s, goto(s, \texttt{t}))\}$, where `t` is a terminal symbol

$[\![E_1 \sqcup E_2]\!]s = [\![E_1]\!]s \cup [\![E_2]\!]s$

$[\![Xj]\!]s = [\![E_j]\!]s$, where $Xj = E_j$ is the flow equation for Xj

$[\![E_1 \cdot E_2]\!]s = \{reduce(s, p') \mid p' \in ([\![E_1]\!]s) \oplus [\![E_2]\!]\}$,

where $S \oplus g = \{p :: g(top(p)) \mid p \in S\}$

where $reduce(s, p)$ reduces the final states within parse stack, $s :: p$.

$reduce(s, p) =$
  $t := top(p)$
  if $t = s_m$, the final state for item, $T \to U_1 U_2 \cdots U_m \cdot$,
  then $p' := pop(m, p)$ // pop m states, corresponding to $U_1 U_2 \cdots U_m$
    $p'' := p' :: goto(top(s :: p'), T)$
    return $reduce(s, p'')$ // repeat till finished
  else *return* p // t was not a final state, so nothing to reduce

Fig. 4. Abstract collecting interpretation: $\mathcal{X}_i(s) = [\![E_i]\!]s$ denotes the set of parse stacks generated by parsing the strings denoted by E_i, starting from parse state s.

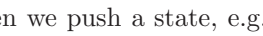
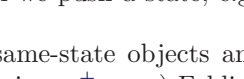
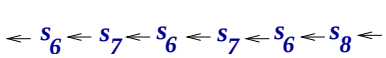
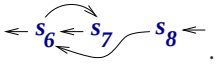
3 Abstract parse stacks

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

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,  , where the stack's top and bottom are marked by pointers; when we push a state, e.g., $p :: s_2$, we get  . The folded stack is formed by merging same-state objects and retaining all links:  . (This can be written as the regular expression, $s_1^+ :: s_2$.) Folding can apply to multiple states, e.g.,  folds to  .

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

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:

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

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

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

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

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

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

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

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

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

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

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

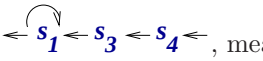
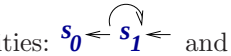
(i) there are multiple s_1s within s_1^+; (ii) there is only one s_1:

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

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

Fig. 5. Iterative solution with folded parse stacks, depicted as regular expressions

The result, $X3(s_0) = \{s_5, s_1^+ :: s_3\}$, asserts that the string at $X3$ might be a well-formed S phrase or it might contain a surplus of unmatched left brackets.

At the end of the calculation in Figure 5, the reduction of $S \rightarrow [S]$ is done on the folded stack segment, $s_1^+ :: s_3 :: s_4$, that is, the complete stack is  , 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:  and . We compute the result for each case, as shown in the Figure.

A set of parse stacks can be soundly approximated by a single, abstract stack: For label set Σ, a *Σ-labelled graph*, g, is a tuple, $\langle nodes_g, edges_g, label_g \rangle$, where

- $nodes_g$ is a set of nodes,
- $edges_g \subseteq nodes_g \times nodes_g$ is a set of directed edges (at most one per source, target node pair),
- and $label_g : nodes_g \to \Sigma$ assigns a label to each node.

Let $Graph_\Sigma$ be the set of Σ-labelled graphs.

An *abstract stack* is a triple, (g, bot, top), such that $g \in Graph_\Sigma$ and $bot, top \in nodes_g$ mark the bottom and top nodes of the stack. Let $AbsStack_\Sigma$ be the set of abstract stacks labelled with Σ-values.

Example: the stack, $s_1 :: s_1 :: s_3$, is modeled as $(\langle \{a, b, c\},\ \{(c, b), (b, a)\},\ [a \mapsto s_1, b \mapsto s_1, c \mapsto s_3] \rangle,\ a,\ c)$.

An abstract stack, $(g, bot, top) \in AbsStack_\Sigma$, concretizes to a set of parse stacks:

$$\gamma(g, bot, top) = \{st \in \mathcal{P}(\Sigma^+) \mid st \text{ is a finite path through } g \text{ from } top \text{ to } bot\}$$

Two abstract stacks, $G_1 = (g_1, bot_1, top_1)$ and $G_2 = (g_2, bot_2, top_2)$, are composed by :: into the disjoint union of g_1 and g_2 plus one new edge from bot_2 to top_1:

$$\begin{aligned} G_1 :: G_2 = (\langle & nodes_{g_1} \uplus nodes_{g_2}, \\ & edges_{g_1} \cup edges_{g_2} \cup \{(bot_2, top_1)\}, \\ & label_{g_1} + label_{g_2} \rangle,\quad bot_1,\quad top_2) \end{aligned}$$

An abstract stack is folded (widened) by merging all nodes that share the same label, in effect, equating the nodes with the labels:

$$\begin{aligned} fold(g, bot, top) = (\langle & \{s \in \Sigma \mid \exists n \in nodes_g, label_g(n) = s\}, \\ & \{(s, s') \mid \exists (n, n') \in edges_g, label_g(n) = s, label_g(n') = s'\}, \\ & \lambda s.s \rangle,\quad label_g(bot),\quad label_g(top)) \end{aligned}$$

The abstract interpretation of flow equation, $Xi = E_i$, is the function,
$\mathcal{X}_i : \Sigma \to \mathcal{P}_{fin}(AbsStack_\Sigma)$, defined as

$$\mathcal{X}_i(s) = \{fold(p) \mid p \in [\![E_i]\!](s)\}.$$

This interpretation is sound for the abstract collecting semantics in Figure 4.

A set of abstract stacks can be further abstracted into a *single* stack of form, $Graph_\Sigma \times \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, by unioning the stacks' node sets, edge sets, *bot*-values and *top*-values. The resulting "stack" is a *subgraph* of the parser's *goto*-controller.

Fig. 6. Abstract interpretation defined in terms of abstract, folded, parse stacks

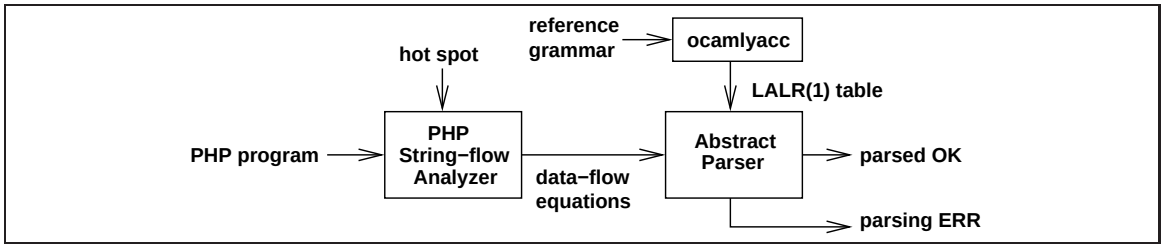


Fig. 7. Implementation

4 Implementation and experiments

The implementation is in Objective Caml. The front end of Minamide's analyzer for PHP [6] 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. Abstract parsing works directly on characters (not tokens), so the reference grammar is written for scannerless parsing. (Performance was good enough for practical use.)

We applied our tool to a suite of PHP programs that dynamically generate HTML documents, the same one studied by Minamide [6], 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

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.

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

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.

Minamide excluded one PHP application, named `tagit`, from his experiments [6], 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 an *update-invariance property* (a string update must preserve the existing parse). Unexpectedly (to us!), so many string updates in `tagit` violated update invariance that our abstract parser generated too many false-positives to be helpful.

We can eliminate these false positives with some easy finite-machine theory: every string update, e.g.,

```
x = ...
replace(pattern, target, x)
```

defines a f.s.a.-transducer (`pattern` is an f.s.a.; `target` is the f.s.a.'s output; `x` is the input). For each such transducer that appears in flow-equation generation, *we compose the transducer with the LR-parse controller, itself a transducer*. The compound transducer directs the abstract parse.

We are implementing this technique, which can also be used for input validation and sharpening the results of tests in conditional commands.

We are also implementing reference grammars for applications that generate XML structures.

5 Conclusion

Injection and cross-site-scripting attacks can be reduced by analyzing the programs that dynamically generate documents [9]. 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`

The paper was published at the 2009 Static Analysis Symposium (Springer LNCS 5673). The paper is saved locally 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. E. Duesterwald, R. Gupta, and M.L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM TOPLAS*, 19:992–1030, 1997.
4. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engg.*, 1995.
5. 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.
6. 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.
7. P. Thiemann. Grammar-based analysis of string expressions. In *Proc. ACM workshop Types in languages design and implementation*, pages 59–70, 2005.
8. 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.
9. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. ACM PLDI*, pages 32–41, 2007.