Abstract LR-parsing

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 explain and illustrate *abstract parsing*, a static-analysis technique based on abstract interpretation, LR-parsing, and partial evaluation for validating PHP-like scripts that generate HTML/XML-style documents. A validated script is guaranteed to generate documents that are well formed with respect to the document language's LR(k)-grammar. In this way, abstract parsing resembles compiler data-type checking: a validated script will "not go wrong" and output a malformed, dynamically generated document.

After presenting abstract parsing for LR(k)-grammars, we handle these important extensions: *(i)* String-replacement operations are analyzed by composing the finite-state automaton defined by a string replacement with the finite-state control of the LR(k)-parser. *(ii)* Conditional-test expressions are implemented by filter automata, which are also composed with the parser's finite-state control. *(iii)* Dynamically supplied and potentially malicious user input is predicted by characterizing it with an LR(k)-grammar and analyzing the strings generated by the grammar. *(iv)* Synthesized-attribute grammars are employed to calculate the semantics of the dynamically generated documents.

1 Introduction

Scripting languages use strings as a "universal data structure" to communicate documents, data structures, and programs. For example, a PHP script might assemble within one long string an entire HTML page or an XML document or an SQL query. An incorrectly assembled string-document might later cause failure when it is supplied as input to its intended processor (a web browser or database engine). Worse still, the string-document might contain textual input supplied by a malicious user and initiate a cross-site-scripting or injection attack [20].

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

In this paper, we marry techniques from LR-parsing theory, abstract interpretation, and partial evaluation to formulate a static analysis that validates that the string-documents generated by a script are grammatically well formed with respect to the reference grammar. We call the analysis

* doh@hanyang.ac.kr Supported by R01-2006-000-10926-0, the Basic Research Program of the Korea Science and Engineering Foundation and by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST)/Korea Science and Engineering Foundation(KOSEF) R11-2008-007-01003-0.

** das@ksu.edu Supported by NSF CNS-0939431.

abstract parsing because it is an abstract interpretation of the script conducted simultaneously with the LR-parsing of the string-documents generated by the script.

The meaning of each potentially generated string-document is not a set of strings or a regular expression but is (an approximation of) the *parse stack* that the LR-parser would generate when it parsed the string-document — the parse stack encodes both the string and its context-free structure, thus providing greater precision than techniques that approximate the string via regular expressions.

The paper proceeds as follows. After presenting abstract parsing for LR(0) and general LR(k) grammars, we handle these important extensions:

– String-replacement operations are analyzed by *composing* the finite-state automaton defined by a string replacement with the finite-state control of the LR(k)-parser.
– Conditional-test expressions are implemented by filter automata, which are also composed with the parser's finite-state control.
– Dynamically supplied, potentially malicious, user input is predicted and processed by characterizing it by an LR(k)-grammar and analyzing the strings generated from by grammar.
– Attribute grammar technology is added to calculate the semantic properties of dynamically generated string output.

2 Background example

Say that a script must generate output strings that conform to this grammar,

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

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 the example program should print only well-formed S-phrases — the occurrence of `x` at "`print x`" is a "hot spot," where we must analyze `x`'s possible values. Three approaches have been proposed to do this:

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

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

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

Fig. 1. Sample program and its flow equations

1. An analysis based on *type checking* assigns types (reference-grammar nonterminals) to the program's variables and uses them to validate that the program is well typed. The occurrences of `x` should be data-typed as S, but `r` has no data type that corresponds to a nonterminal.

2. An analysis based on *regular expressions* [2–6, 14, 15, 19] solves the program-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. Improvement in precision can be obtained with *parenthesis grammars* [13, 17], which generate good regular-expression approximations of simple bracket grammars but fail to express general context-free structure.
3. A *grammar-based analysis* [18] treats the flow equations as a set of grammar rules, and a language-inclusion check tries to prove that all $X3$-generated strings are S-generable. A useful instance of this technique is due to Møller and Schwarz, who check language inclusion with the more restrictive but useful SGML DTD [9] for HTML documents [16].

Our approach solves the program-flow equations in Figure 1 in the domain of *parse stacks* — $X3$'s meaning is the *set of parse stacks* of the strings that might be denoted by `x`. Our technique simultaneously unfolds and LR-parses the strings defined by $X3$, computing a parse stack that expresses both the structure in the flow equations and that of the reference grammar.

The technique is implemented by a partial-evaluation-style specialization of the program's flow equations applied to the LR-parser. When the specialized, residual flow equations are "executed" (solved with a least-fixed semantics), they generate (sets of) parse stacks as their answers.

Of course, a program might generate infinitely many different strings and therefore the analysis might compute an infinite set of parse stacks. We finitely approximate an infinite set of parse stacks by exploiting this key feature of LR-parse theory: Each parse stack is exactly a finite path through the LR-parser's finite-state control automaton and can be approximated by the smallest subgraph of the automaton that covers the path. The smallest-subgraph approximation is computed merely by *folding the parse stack on its repeating state(s)*.

3 Abstract LR(0)-parsing

We present the technique via the example program in Figure 1. For the example grammar, $S \rightarrow \texttt{a} \mid \texttt{[}S\texttt{]}$, Figure 2 gives the LR(0)-parse-controller automaton and a parse of the string, `[[a]]`. The parse-controller automaton is presented graphically, and its transitions are coded as shift/reduce rewriting rules, which we use to parse the string. The current state, $[s_i]$, of the parse appears as the top state in the parse stack, $s_0 :: s_1 :: \cdots :: [s_i]$. Input symbols, i, are supplied to state, s, in the format, $[i \hookrightarrow s]$. The parser's start state is $[s_0]$.

Say that we must validate that the program in Figure 1 prints only S-structured phrases. To analyze the program's hot spot at $X3$, we must $LRparse(X3, s_0)$, which we portray as a function call, $X3[s_0]$ — *we treat the program-flow equations in Figure 1 as functions defined in combinator notation* and we specialize (apply) a flow equation to the state used to parse it.

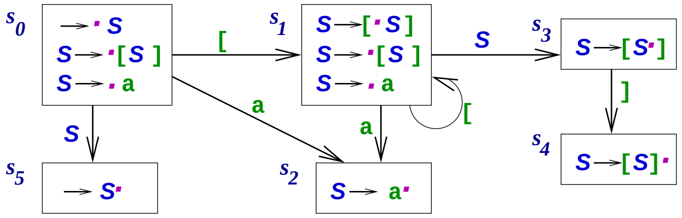
The flow equation, $X3 = X1$, generates this call step:

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

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

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

The union of the parses of strings at $X0$ and $X2$ from s_0 must be computed. (*Important:* this computes a set of parse stacks. In this example, all the sets are singletons, and we omit the set



Shift transitions:

$[\texttt{[} \hookrightarrow s_0] \Rightarrow s_0 :: [s_1]$
$[S \hookrightarrow s_0] \Rightarrow [s_5]$
$[\texttt{a} \hookrightarrow s_0] \Rightarrow s_0 :: [s_2]$

$[\texttt{[} \hookrightarrow s_1] \Rightarrow s_1 :: [s_1]$
$[\texttt{a} \hookrightarrow s_1] \Rightarrow s_1 :: [s_2]$
$[S \hookrightarrow s_1] \Rightarrow s_1 :: [s_3]$
$[\texttt{]} \hookrightarrow s_3] \Rightarrow s_3 :: [s_4]$

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]]`
$[\texttt{[} \hookrightarrow s_0]$	`[a]]`
$s_0 :: [s_1]$	`[a]]`
$s_0 :: [\texttt{[} \hookrightarrow s_1]$	`a]]`
$s_0 :: s_1 :: [s_1]$	`a]]`
$s_0 :: s_1 :: [\texttt{a} \hookrightarrow s_1]$	`]]`
$s_0 :: s_1 :: s_1 :: [s_2]$	`]]` (reduce: $S \rightarrow \texttt{a}$)
$s_0 :: s_1 :: [S \hookrightarrow s_1]$	`]]`
$s_0 :: s_1 :: s_1 :: [s_3]$	`]]`
$s_0 :: s_1 :: s_1 :: [\texttt{]} \hookrightarrow s_3]$	`]`
$s_0 :: s_1 :: s_1 :: s_3 :: [s_4]$	`]` (reduce: $S \rightarrow \texttt{[}S\texttt{]}$)
$s_0 :: [S \hookrightarrow s_1]$	`]`
$s_0 :: s_1 :: [s_3]$	`]`
$s_0 :: s_1 :: [\texttt{]} \hookrightarrow s_3]$	
$s_0 :: s_1 :: s_3 :: [s_4]$	(reduce: $S \rightarrow \texttt{[}S\texttt{]}$)
$[S \hookrightarrow s_0]$	
$[s_5]$	(finished)

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

braces to reduce notational clutter.) We consider first $X0[s_0]$:

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

That is, a parse of `'a'` from s_0 generates the one-element stack, s_5 (actually, $\{[s_5]\}$) — all strings denoted by $X0$ are S-phrases. Next,

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

The $\oplus$ is a "continuation operator": For parse stack, st, and combinator expression, E, define $st \oplus E = tail(st) :: E[head(st)]$. That is, stack st's top state feeds to E. (More generally, for a set of stacks, S, define $S \oplus E = \{tail(st) :: E[head(st)] \mid st \in S\}$.)

Next, $X1[s_1] = X0[s_1] \cup X2[s_1]$ computes to $s_1 :: [s_3]$ (as explained below, the recursion generated by $X2[s_1]$ is resolved by least-fixed-point iteration), so

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

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

Here is the list of residual equations generated from the partial evaluation of the initial call, $X3[s_0]$:

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

Each $X_i[s_j] = E_{ij}$ is *a first-order equation* whose answer is a set of parse stacks.

The equations for $X1[s_1]$ and $X2[s_1]$ are mutually recursively defined, and their solutions are computed by least-fixed-point iteration. Here are the solutions:

$$\begin{aligned}
&X1[s_1] = X0[s_1] \cup X2[s_1] = (s_1 :: [s_3]) \cup (s_1 :: [s_3]) = s_1 :: [s_3] \\
&X2[s_1] = s_1 :: (X1[s_1] \oplus R) \Rightarrow s_1 :: s_1 :: R[s_3] \Rightarrow s_1 :: [s_3]
\end{aligned}$$

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

$X3[s_0] = X1[s_1] = [s_5]$ validates that the strings printed at the hot spot must be S-phrases. (Note again: these answers are really sets, that is, $X3[s_0] = \{[s_5]\}$.) The algorithm that generates the residual equations and simultaneously solves them is a worklist algorithm like those used for demand-driven data-flow analyses [1, 8, 10]; it also resembles *minimal function-graph semantics* [12].

Figure 3 shows the worklist algorithm applied to the example. The algorithm uses three data structures: the worklist of unresolved calls, $Xi[s_j]$; a Cache ("seen-before list") that maps each call to its current (partial) solution (a set of abstract parse stacks); and the graph of call dependencies, which is dynamically constructed.

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

- Extract the front call, $X[s]$, from the worklist, and for its corresponding flow equation, $X = E$, compute $E[s]$, a set, using the parser's shift/reduce rules:
  1. While computing $E[s]$, if a call, $X'[s']$ is encountered, *(i)* add the dependency, $X'[s'] \to X[s]$, to the dependency graph (if not already present); *(ii)* if there is no entry for $X'[s']$ in the cache, then assign $\mathit{Cache}[X'[s']] = \emptyset$ to the cache and add $X'[s']$ to the end of the worklist; *(iii)* use $\mathit{Cache}[X'[s']]$ as the meaning of $X'[s']$ in the computation of $E[s]$.
  2. When $E[s]$ computes to an answer set, P, and P contains a parse stack not already listed in $\mathit{Cache}[X[s]]$, then set $\mathit{Cache}[X[s]] = \mathit{Cache}[X[s]] \cup P$ and add to the end of the worklist all $X''[s'']$ such that $X[s] \to X''[s'']$ appears in the dependency graph.

Worklist, 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

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

Generated dependency graph:

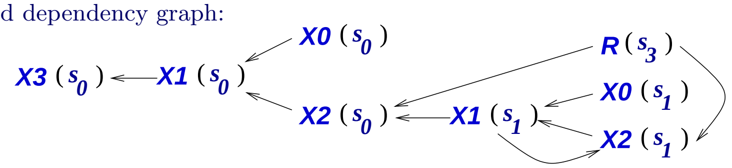


Fig. 3. Worklist-algorithm calculation of call, $X3[s_0]$, in Figure 1

4 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' . ']'
```

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

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

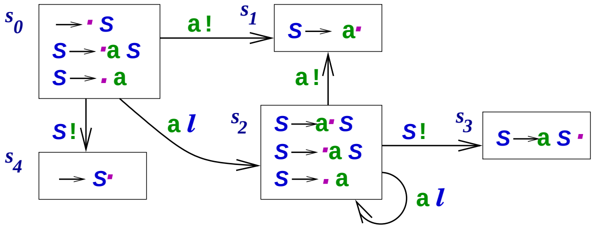
To bound the set, we abstract it by "folding" its stacks so that no state repeats in a stack. A stack segment like $s_1 :: s_1 :: [s_3]$ is a graph, $\leftarrow s_1 \leftarrow s_1 \leftarrow [s_3] \leftarrow$; the folded stack merges identical states: $\leftarrow \overset{\circlearrowleft}{s_1} \leftarrow [s_3] \leftarrow$. Since the set of parse-state names is finite, folding produces a finite set of finite-sized stacks (that contain cycles). For the previous example, the worklist algorithm calculates $X3[s_0] = \{[s_5],\ s_1^+ :: [s_3]\}$. As noted earlier, each parse stack is a finite path through the LR-parser's finite-state controller automaton, and folding the parse stack generates the smallest subgraph of the automaton that covers the path.

Stack folding can be profitably delayed when straightline code is analyzed, so we fold stacks *only if* there is backwards control flow: When calculating a call, $Xi[s_i] = \cdots Xj[s_j] \cdots$, *if* $Xj \longrightarrow Xi$ is a "back arc" in the program's control flow (that is, $j \geq i$), *only then* we fold the set of stacks defined by $Xj[s_j]$ to compute $Xi[s_i]$. This way, we lose precision exactly when the source program's control flow itself loses precision. Again, finite convergence is guaranteed.

5 LR(k) grammars are accommodated the same way

Abstract parsing also applies to LR(k) grammars, for $k > 0$. Figure 4 presents an LR(1) grammar, its controller, and an example parse. The parse states have form, $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, where $0 \leq j \leq k+1$. When a program is statically parsed with an LR(k) grammar, $k > 0$, the first-order

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:

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

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

$[\mathtt{!} \hookrightarrow S \hookrightarrow s_0] \Rightarrow [s_4]$

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

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

$[\mathtt{!} \hookrightarrow S \hookrightarrow s_2] \Rightarrow s_2 :: [\mathtt{!} \hookrightarrow s_3]$

Reduce transitions:

$s_i :: [\ell \hookrightarrow s_1] \Rightarrow [\ell \hookrightarrow S \hookrightarrow s_i]$

$s_i :: s_j :: [\ell \hookrightarrow s_3] \Rightarrow [\ell \hookrightarrow S \hookrightarrow s_i]$

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

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

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

residual equations have form,

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

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

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

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

its partial evaluation proceeds as follows:

$$\begin{aligned}
X3[s_0] &= (X1 \cdot \mathtt{!})[s_0] = X1[s_0] \oplus \mathtt{!}\\
X1[s_0] &= X0[s_0] \cup X2[s_0]\\
X0[s_0] &= \mathtt{a}[s_0] = \{[\mathtt{a} \hookrightarrow s_0]\}\\
X2[s_0] &= (\mathtt{a} \cdot X1 \cdot \mathtt{a})[s_0] = \mathtt{a}[s_0] \oplus (X1 \cdot \mathtt{a})\\
&= \{[\mathtt{a} \hookrightarrow s_0]\} \oplus (X1 \cdot \mathtt{a}) = \{X1[\mathtt{a} \hookrightarrow s_0] \oplus \mathtt{a}\}\\
X1[\mathtt{a} \hookrightarrow s_0] &= X0[\mathtt{a} \hookrightarrow s_0] \cup X2[\mathtt{a} \hookrightarrow s_0]\\
X0[\mathtt{a} \hookrightarrow s_0] &= \{[\mathtt{a} \hookrightarrow s_0]\} = \{[\mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_0]\} \Rightarrow \{s_0 :: [\mathtt{a} \hookrightarrow s_2]\}\\
X2[\mathtt{a} \hookrightarrow s_0] &= (\mathtt{a} \cdot X1 \cdot \mathtt{a})[\mathtt{a} \hookrightarrow s_0] = [\mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_0] \oplus (X1 \cdot \mathtt{a})\\
&= \{s_0 :: [\mathtt{a} \hookrightarrow s_2]\} \oplus (X1 \cdot \mathtt{a})\} = \{s_0 :: X1[\mathtt{a} \hookrightarrow s_2] \oplus \mathtt{a}\}\\
X1[\mathtt{a} \hookrightarrow s_2] &= X0[\mathtt{a} \hookrightarrow s_2] \cup X2[\mathtt{a} \hookrightarrow s_2]\\
X0[\mathtt{a} \hookrightarrow s_2] &= \mathtt{a}[\mathtt{a} \hookrightarrow s_2] = \{[\mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_2]\} = \{s_2 :: [\mathtt{a} \hookrightarrow s_2]\}\\
X2[\mathtt{a} \hookrightarrow s_2] &= (\mathtt{a} \cdot X1 \cdot \mathtt{a})[\mathtt{a} \hookrightarrow s_2] = [\mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_0] \oplus (X1 \cdot \mathtt{a})\\
&= \{s_2 :: [\mathtt{a} \hookrightarrow s_2]\} \oplus (X1 \cdot \mathtt{a}) = \{s_2 :: (X1[\mathtt{a} \hookrightarrow s_2] \oplus \mathtt{a})\}\\
X1[\mathtt{a} \hookrightarrow s_2] &= \{s_2 :: [\mathtt{a} \hookrightarrow s_2]\} \cup \{s_2 :: (X1[\mathtt{a} \hookrightarrow s_2] \oplus \mathtt{a})\}
\end{aligned}$$

The residual equations are solved by least-fixed point calculation; $X1[\mathtt{a} \hookrightarrow s_2]$ computes to $\{s_2^i :: [\mathtt{a} \hookrightarrow s_2] \mid i \in 1, 3, 5, \cdots\}$, which our analysis approximates by $\{s_2^+ :: [\mathtt{a} \hookrightarrow s_2]\}$. Using this result, we obtain

$$\begin{aligned}
X1[\mathtt{a} \hookrightarrow s_2] &= \{s_2^+ :: [\mathtt{a} \hookrightarrow s_2]\}\\
X2[\mathtt{a} \hookrightarrow s_0] &= \{s_0 :: s_2^+ :: [\mathtt{a} \hookrightarrow s_2]\}\\
X1[\mathtt{a} \hookrightarrow s_0] &= \{s_0 :: s_2^* :: [\mathtt{a} \hookrightarrow s_2]\}\\
X2[s_0] &= \{s_0 :: s_2^+ :: [\mathtt{a} \hookrightarrow s_2]\}\\
X1[s_0] &= \{[\mathtt{a} \hookrightarrow s_0]\} \cup \{s_0 :: s_2^+ :: [\mathtt{a} \hookrightarrow s_2]\}\\
X3[s_0] &= \{[\mathtt{!} \hookrightarrow \mathtt{a} \hookrightarrow s_0]\} \cup \{s_0 :: s_2^+ :: [\mathtt{!} \hookrightarrow \mathtt{a} \hookrightarrow s_2]\} = \{[s_4]\}
\end{aligned}$$

$$\begin{aligned}
&\text{since } \{[\mathtt{!} \hookrightarrow \mathtt{a} \hookrightarrow s_0]\} = \{s_0 :: [\mathtt{!} \hookrightarrow s_1]\} = \{[\mathtt{!} \hookrightarrow S \hookrightarrow s_0]\} = \{[s_4]\}\\
&\text{and } \{s_0 :: s_2^+ :: [\mathtt{!} \hookrightarrow \mathtt{a} \hookrightarrow s_2]\} = \{s_0 :: s_2^+ :: [\mathtt{!} \hookrightarrow s_1]\} \quad (s_2^+ :: s_2 \text{ is approximated to } s_2^+)\\
&\qquad = \{s_0 :: s_2^* :: [\mathtt{!} \hookrightarrow S \hookrightarrow s_2]\} \quad (\text{reduce } S \rightarrow a)\\
&\qquad = \{s_0 :: s_2^+ :: [\mathtt{!} \hookrightarrow s_3]\}\\
&\qquad = \{[\mathtt{!} \hookrightarrow S \hookrightarrow s_0],\ s_0 :: s_2^* :: [\mathtt{!} \hookrightarrow S \hookrightarrow s_2]\} \quad (\text{reduce } S \rightarrow \mathtt{a}S)\\
&\qquad = \{[s_4]\} \quad (\text{second set element adds nothing to the fixed point})
\end{aligned}$$

This proves that all possible string values of `x` at the end are well-structured S-phrases.

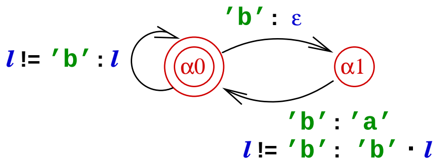
6 Abstract parsing with string-replacement operations

Because of the state explosion that arises with LR(k) grammars, our implementation uses LALR(k) grammars instead. The reason we introduced the LR(1) example was to generalize the parse state to hold multiple input symbols, which we now use to process string-update operations.

Scripting languages support string updates of this form,

```
y = replace 'bb' by 'a' in x
```

where the pattern (here, `bb`) can be a regular expression. The update operation defines an automaton (more precisely, a *transducer*):



$$\begin{array}{l}
\mathtt{b} \hookrightarrow \alpha_0 \longrightarrow \alpha_1 : \epsilon \\
\ell \neq \mathtt{b} \hookrightarrow \alpha_0 \longrightarrow \alpha_0 : \ell \\
\mathtt{b} \hookrightarrow \alpha_1 \longrightarrow \alpha_0 : \mathtt{a} \\
\ell \neq \mathtt{b} \hookrightarrow \alpha_1 \longrightarrow \alpha_0 : \mathtt{b} \cdot \ell
\end{array}$$

Both graphical and linear codings are displayed here. We use $: e$ to mean "emit e as output." When a `replace` operation appears in a program that is analyzed, *the transducer, α, defined by* `replace` *is composed with the parser automaton* — a state configuration now holds *two* states:

$$[\ell_{new} \hookrightarrow \alpha_m, \ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$$

Here, α_m is the current state of the transducer and s is the current state of the parser. A new input, ℓ_{new}, submits first to α_m, which transits and possibly emits input for s:

$$[\alpha_n, \ell_{j+1} \hookrightarrow \ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$$

In this way, strings are updated by `replace` before they are parsed. The assignment,

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

generates the flow equation

$$X = insert_\alpha \cdot E \cdot erase_\alpha$$

where α_0 names the start state of transducer α generated from patterns $S1$ and $S2$ and

$$\begin{array}{l}
insert_\alpha[\cdots s] \Rightarrow [\alpha_0, \cdots s] \\
erase_\alpha[\alpha_i, \cdots s] \Rightarrow [\cdots s]
\end{array}$$

Here is a small example, worked with the above transducer and the parser in Figure 2:

```
y = 'bb]'
x = '['.(replace 'bb' by 'a' in y)
```

$$\begin{array}{l}
Y = \mathtt{b} \cdot \mathtt{b} \cdot \mathtt{]} \\
X = \mathtt{[} \cdot (insert_\alpha \cdot Y \cdot erase_\alpha)
\end{array}$$

The abstract parse of $X[s_0]$ proceeds like this:

$$
\begin{aligned}
X[s_0] &= [\texttt{[} \hookrightarrow s_0] \oplus (insert_\alpha \cdot Y \cdot erase_\alpha) = s_0 :: (insert_\alpha \cdot Y \cdot erase_\alpha)[s_1] && (i)\\
&= s_0 :: (Y \cdot erase_\alpha)[\alpha_0, s_1] = s_0 :: (Y[\alpha_0, s_1] \oplus erase_\alpha) &&\\
Y[\alpha_0, s_1] &= (\texttt{b} \cdot \texttt{b} \cdot \texttt{]})[\alpha_0, s_1] = [\texttt{b} \hookrightarrow \alpha_0, s_1] \oplus (\texttt{b} \cdot \texttt{]}) = [\alpha_1, s_1] \oplus (\texttt{b} \cdot \texttt{]}) && (ii)\\
&= [\texttt{b} \hookrightarrow \alpha_1, s_1] \oplus \texttt{]} = [\alpha_0, \texttt{a} \hookrightarrow s_1] \oplus \texttt{]} && (iii)\\
&= s_1 :: ([\alpha_0, s_2] \oplus \texttt{]}) && (iv)\\
&= [\alpha_0, S \hookrightarrow s_1] \oplus \texttt{]} = s_1 :: [\alpha_0, s_3] \oplus \texttt{]} = s_1 :: [\texttt{]} \hookrightarrow \alpha_0, s_3] &&\\
&= s_1 :: [\alpha_0, \texttt{]} \hookrightarrow s_3] = s_1 :: s_3 :: [\alpha_0, s_4] &&
\end{aligned}
$$

So,

$$
\begin{aligned}
X[s_0] &= s_0 :: (Y[\alpha_0, s_1] \oplus erase_\alpha) = s_0 :: (s_1 :: s_3 :: [\alpha_0, s_4] \oplus erase_\alpha) &&\\
&= s_0 :: (s_1 :: s_3 :: erase_\alpha[\alpha_0, s_4]) && (v)\\
&= s_0 :: (s_1 :: s_3 :: [s_4]) = [S \hookrightarrow s_0] = [s_5] &&
\end{aligned}
$$

At point (i), input symbol `[` is supplied directly to the parser. The transducer's start state is then added to the state configuration, and the string generated by Y is supplied to the transducer *before* Y's string is parsed — see (ii). At point (iii), the sequence `bb` causes the transducer state to emit `a`, which is supplied to the parser state. Once the parser reduces `a` to nonterminal S, the transducer state is carried along in the state configuration; see (iv). At (v), the string transducer has finished its effects and is erased.

The composition of transducer with parser in our demand-driven, backwards, precondition-style analysis means there is no backtracking and reparsing because of string updates — there is only the one parse of the appropriately altered string.

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

```
y = replace 'bb' by 'a' in 'bbb'
```

where transducer α has α_0 as its final state, the processing of `'bbb'` causes α to finish in state α_1, implicitly holding `'b'` in its state. The `'b'` must be "flushed", so we add this last transition to α:

$$eos \hookrightarrow \alpha_1 \rightarrow \alpha_0 : \texttt{b}$$

Where eos denotes "end of string." This transition is enacted by the $erase_\alpha$ operation.

The embedding of the transducer state in the parse configuration does not affect the least-fixed point machinery for computing the solutions to the residual equations. (But a state explosion can result.) In addition, the residual-equation-least-fixed-point-calculation allows string replacements within loop bodies, avoiding difficulties encountered in related techniques [4, 5, 14].

7 Other applications of string transducers

Transducers have other applications in abstract parsing; here are two.

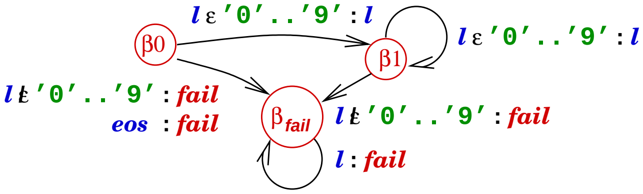
Composing a scanner with the parser The basic abstract-parsing algorithm does "scannerless parsing" — the characters of a string are input one at a time to the parser state, which must parse characters into words and words into phrases. We have found the technique acceptable in practice for HTML grammars, but for theoretical or practical reasons, one might wish to scan characters into tokens before parsing them.

A scanner defined as a transducer, σ, can be added to the state configuration so that abstract parsing is undertaken with configurations of the form, $[\sigma_i, \ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s_j]$, where σ_i is the state of the scanner, ℓ_i are the generated tokens, and s_j is the state of the parser. There is no resulting state explosion, since a scannerless parser must hold the same scanner-state information within its parse state, anyway, and there is the advantage that scanning and parsing can be defined separately.

String filtering through conditional commands A technique needed for taint analysis [20, 21, 19] is filter functions that model the tests of conditional commands. For example, a script might contain a conditional command that filters untrusted user input:

```
read x
if isAllDigits(x) :
then  ···assert here that x holds all digits···
```

The test expression, `isAllDigits(x)`, is defined as a transducer that reads string `x` and emits *failure* ($\perp$) if a character is a nondigit. A failure means that `x`'s value is filtered from entering the conditional's then-arm. The filter transducer for `isAllDigits(x)` appears:



The transducer emits *fail* when its input fails 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):`	$X_B = insert_\beta \cdot X \cdot erase_\beta$
`then ···x···`	$\cdots X_B \cdots$
`else ···x···`	$X_{\neg B} = insert_{\neg\beta} \cdot X \cdot erase_{\neg\beta}$
	$\cdots X_{\neg B} \cdots$

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

The *fail* character is special — when processed as an input, it causes the parse to denote $\perp$ (empty set in the powerset lattice): $[\cdots, \mathit{fail}, \cdots] = \perp$. For example,

```
x = 'a'
if isAllDigits(x):
   print x
```

$$X0 = \mathtt{a}$$
$$X1 = insert_\beta \cdot X0 \cdot erase_\beta$$
$$X2 = X1 \cdot \mathtt{!}$$

and

$$\begin{aligned} X2[s_0] &= X1[s_0] \\ X1[s_0] &= X0[\beta_0, s_0] \oplus erase_\beta \\ X0[\beta_0, s_0] &= \mathtt{a}[\beta_0, s_0] = [\mathtt{a} \hookrightarrow \beta_0, s_0] = [\beta_0, \mathit{fail} \hookrightarrow s_0] = \perp \end{aligned}$$

Hence,

$$X1[s_0] = erase_\beta \perp = \perp = X2[s_0]$$

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

8 Modelling global variables and user input by nonterminals

An abstract parser can process a grammar's nonterminal symbols as input just like terminal symbols: the symbol is supplied to the parse state, which shifts it. Say that a module uses a string-valued global variable that is initialized outside of the module. If we can assume the global 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 = '['.g .']'
print x
```

$$\begin{aligned} G &= S \\ X &= \texttt{[} \cdot G \cdot \texttt{]} \end{aligned}$$

We readily compute the abstract parse for $X[s_0]$, using Figure 2:

$$\begin{aligned} X[s_0] &= (\texttt{[} \cdot G \cdot \texttt{']'})[s_0] = [\texttt{[} \hookrightarrow s_0] \oplus (G \cdot \texttt{']'}) = s_0 :: (G[s_1] \oplus \texttt{]}) = s_0 :: ([S \hookrightarrow s_1] \oplus \texttt{]}) \\ &= s_0 :: (s_1 :: \texttt{]}[s_3]) = s_0 :: s_1 :: [\texttt{]} \hookrightarrow s_3] = s_0 :: s_1 :: s_3 :: [s_4] = \cdots = [s_5] \end{aligned}$$

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

```
g = read_S()
x = '['.g. ']'
print x
```

$$\begin{aligned} G &= S \\ X &= \texttt{[} \cdot G \cdot \texttt{]} \end{aligned}$$

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 — the program might contain string-replacement operations, which cannot process nonterminals. We solve this problem by unfolding the nonterminal, supplying the generated strings to the string-replacement transducer (recall that $S \rightarrow \texttt{a} \mid \texttt{[}S\texttt{]}$):

```
g = read_S()
y = replace '[' by '[[' in g
print y
```

$$\begin{aligned} S &= \texttt{a} \sqcup \texttt{[} \cdot S \cdot \texttt{]} \\ G &= S \\ Y &= \mathit{insert}_\gamma \cdot G \cdot \mathit{erase}_\gamma \end{aligned}$$

where transducer γ is the obvious one-state transducer. The analysis proceeds like this:

$$\begin{aligned} Y[s_0] &= G[\gamma_0, s_0] \oplus \mathit{erase}_\gamma \\ G[\gamma_0, s_0] &= S[\gamma_0, s_0] \\ S[\gamma_0, s_0] &= (\texttt{a}[\gamma_0, s_0]) \cup (\texttt{[} \cdot S \cdot \texttt{]}[\gamma_0, s_0]) \end{aligned}$$

The last residual equation, for $S[\gamma_0, s_0]$, shows how nonterminal S is unfolded and its symbols fed to γ. There is a tedious but finitely computable solution:

$$\begin{aligned} S[\gamma_0, s_0] &= (\texttt{a}[\gamma_0, s_0] \cup (\texttt{[} \cdot S \cdot \texttt{]}[\gamma_0, s_0]) \\ &= \{[\gamma_0, s_5]\} \cup \{s_0 :: s_1 :: S[\gamma_0, s_1] \oplus \texttt{]} \end{aligned}$$

The partial evaluation of $S[\gamma_0, s_1]$ unfolds almost identically, producing

$$\begin{aligned} S[\gamma_0, s_1] &= (\texttt{a}[\gamma_0, s_1] \cup (\texttt{[} \cdot S \cdot \texttt{]}[\gamma_0, s_1]) \\ &= \{s_1 :: [\gamma_0, s_3]\} \cup (\texttt{[} \cdot S \cdot \texttt{]}[\gamma_0, s_1]) \\ &= \{s_1 :: [\gamma_0, s_3]\} \cup \{s_1 :: s_1 :: [\gamma_0, s_1] \oplus (S \cdot \texttt{]})\} \\ &= \{s_1 :: [\gamma_0, s_3]\} \cup \{s_1^+ :: (S[\gamma_0, s_1] \cdot \texttt{]})\} \end{aligned}$$

The least fixed-point solution of $S[\gamma_0, s_1]$ is $\{s_1^+ :: [\gamma_0, s_3]\}$, which gives $Y[s_0] = \{[s_5], s_1^+ :: [s_3]\}$.

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:

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

9 Abstract parsing with semantic processing

Since we can predict the syntax of dynamically generated strings, we should be able to predict the semantics as well by adapting attribute-grammar techniques. Here is a simple but useful example. Binary numerals are generated by this LR(1) grammar,

$$\begin{array}{l} B \rightarrow D\,B \mid D \\ D \rightarrow \mathtt{0} \mid \mathtt{1} \end{array}$$

where B stands for the set of binary numerals and D stands for the set of binary digits. The semantics of binary numerals can be specified with *attributes* associated with the grammar symbols and *semantics rules* associated with the productions. Suppose we want to know whether or not a binary numeral is even-valued. The semantic rules associated with the productions below specify how to calculate the answer:

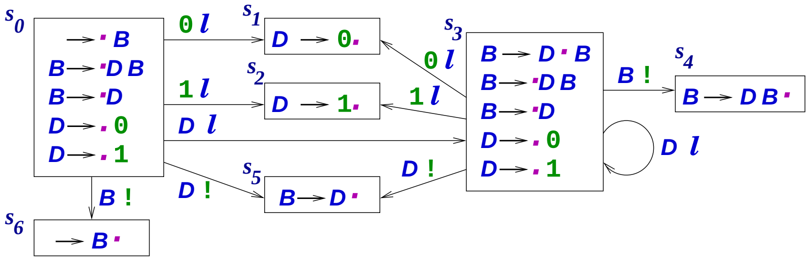
production	semantic rule
$\rightarrow B\,!$	answer $= B.\mathit{even}$
$B \rightarrow D\,B_1$	$B.\mathit{even} = B_1.\mathit{even}$
$B \rightarrow D$	$B.\mathit{even} = D.\mathit{even}$
$D \rightarrow \mathtt{0}$	$D.\mathit{even} = \mathit{true}$
$D \rightarrow \mathtt{1}$	$D.\mathit{even} = \mathit{false}$

Here, each nonterminal, B and D, has a synthesized attribute, *even*, which has value *true* if the binary numeral generated by the nonterminal is an even number, *false* otherwise.

Since the grammar is LR(1) and only associated with synthesized attributes, the semantic rules can be computed during LR-parsing, as seen in Figure 5. When a reduce transition occurs, its corresponding semantic rule is computed. The computed result is annotated to its corresponding state, shown as a superscript in our notation. In the example in Figure 5, the computed attribute values are annotated only to the states, s_0 and s_3. For the example binary numeral, `101`, the computed result is *false*, as expected. For this program,

```
x = '0'
while ...
  x = read_D() · x
print x · !
```

$$\begin{array}{l} X0 = \mathtt{0} \\ X1 = X0 \sqcup X2 \\ X2 = D \cdot X1 \\ X3 = X1 \cdot ! \end{array}$$



Shift transitions:

$[\ell \hookrightarrow \mathtt{0} \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_1]$

$[\ell \hookrightarrow \mathtt{1} \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_2]$

$[\ell \hookrightarrow D \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_3]$

$[\mathtt{!} \hookrightarrow D \hookrightarrow s_0] \Rightarrow s_0 :: [\mathtt{!} \hookrightarrow s_5]$

$[\mathtt{!} \hookrightarrow B \hookrightarrow s_0^b] \Rightarrow [s_6^b]$

$[\ell \hookrightarrow \mathtt{0} \hookrightarrow s_3] \Rightarrow s_3 :: [\ell \hookrightarrow s_1]$

$[\ell \hookrightarrow \mathtt{1} \hookrightarrow s_3] \Rightarrow s_3 :: [\ell \hookrightarrow s_2]$

$[\ell \hookrightarrow D \hookrightarrow s_3] \Rightarrow s_3 :: [\ell \hookrightarrow s_3]$

$[\mathtt{!} \hookrightarrow D \hookrightarrow s_3] \Rightarrow s_3 :: [\mathtt{!} \hookrightarrow s_5]$

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

Reduce transitions:

$s :: [\ell \hookrightarrow s_1] \Rightarrow [\ell \hookrightarrow D \hookrightarrow s^{true}]$

$s :: [\ell \hookrightarrow s_2] \Rightarrow [\ell \hookrightarrow D \hookrightarrow s^{false}]$

$s_i :: s_j^b :: [\ell \hookrightarrow s_4] \Rightarrow [\ell \hookrightarrow B \hookrightarrow s_i^b]$

$s_i^b :: [\ell \hookrightarrow s_5] \Rightarrow [\ell \hookrightarrow B \hookrightarrow s_i^b]$

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

parse stack (top lies at right)	input sequence
$[s_0]$	101!
$[\mathtt{1} \hookrightarrow s_0]$	01!
$[\mathtt{0} \hookrightarrow \mathtt{1} \hookrightarrow s_0]$	1! (shift)
$s_0 :: [\mathtt{0} \hookrightarrow s_2]$	1! (reduce $D \to \mathtt{1}$)
$s_0 :: [D \hookrightarrow s_0^{false}]$	1! (shift)
$s_0^{false} :: [\mathtt{0} \hookrightarrow s_3]$	1!
$s_0^{false} :: [\mathtt{1} \hookrightarrow \mathtt{0} \hookrightarrow s_3]$	! (shift)
$s_0^{false} :: s_3 :: [\mathtt{1} \hookrightarrow s_1]$	! (reduce $D \to \mathtt{0}$)
$s_0^{false} :: [\mathtt{1} \hookrightarrow D \hookrightarrow s_3^{true}]$	! (shift)
$s_0^{false} :: s_3^{true} :: [\mathtt{1} \hookrightarrow s_3]$	!
$s_0^{false} :: s_3^{true} :: [\mathtt{!} \hookrightarrow \mathtt{1} \hookrightarrow s_3]$	(shift)
$s_0^{false} :: s_3^{true} :: s_3 :: [\mathtt{!} \hookrightarrow s_2]$	(reduce $D \to \mathtt{1}$)
$s_0^{false} :: s_3^{true} :: [\mathtt{!} \hookrightarrow D \hookrightarrow s_3^{false}]$	(shift)
$s_0^{false} :: s_3^{true} :: s_3^{false} :: [\mathtt{!} \hookrightarrow s_5]$	(reduce $B \to D$)
$s_0^{false} :: s_3^{true} :: [\mathtt{!} \hookrightarrow B \hookrightarrow s_3^{false}]$	(shift)
$s_0^{false} :: s_3^{true} :: s_3^{false} :: [\mathtt{!} \hookrightarrow s_4]$	(reduce $B \to D\,B$)
$s_0^{false} :: [\mathtt{!} \hookrightarrow B \hookrightarrow s_3^{false}]$	(shift)
$s_0^{false} :: s_3^{false} :: [\mathtt{!} \hookrightarrow s_4]$	(reduce $B \to D\,B$)
$[\mathtt{!} \hookrightarrow B \hookrightarrow s_0^{false}]$	(shift)
$[s_6^{false}]$	(finished)

Fig. 5. Syntax-directed semantic processing for LR(1) grammar, $B \to DB \mid D, \quad D \to \mathtt{0} \mid \mathtt{1}$

its abstract parsing with synthesized-attribute computing proceeds as follows:

$$\begin{aligned}
X3[s_0] &= (X1 \cdot \mathtt{!})[s_0] = X1[s_0] \oplus \mathtt{!} \\
X1[s_0] &= X0[s_0] \cup X2[s_0] \\
X0[s_0] &= \mathtt{0}[s_0] = \{[\mathtt{0} \hookrightarrow s_0]\} \\
X2[s_0] &= (D \cdot X1)[s_0] = [D \hookrightarrow s_0] \oplus X1 = X1[D \hookrightarrow s_0] \\
&= X0[D \hookrightarrow s_0] \cup X2[D \hookrightarrow s_0] \\
X0[D \hookrightarrow s_0] &= \{[\mathtt{0} \hookrightarrow D \hookrightarrow s_0]\} \Rightarrow \{s_0 :: [\mathtt{0} \hookrightarrow s_3]\} \\
X2[D \hookrightarrow s_0] &= (D \cdot X1)[D \hookrightarrow s_0] = [D \hookrightarrow D \hookrightarrow s_0] \oplus X1 \\
&\Rightarrow \{s_0 :: [D \hookrightarrow s_3]\} \oplus X1 = \{s_0 :: X1[D \hookrightarrow s_3]\} \\
X1[D \hookrightarrow s_3] &= X0[D \hookrightarrow s_3] \cup X2[D \hookrightarrow s_3] \\
X0[D \hookrightarrow s_3] &= \{[\mathtt{0} \hookrightarrow D \hookrightarrow s_3]\} \Rightarrow \{s_3 :: [\mathtt{0} \hookrightarrow s_3]\} \\
X2[D \hookrightarrow s_3] &= (D \cdot X1)[D \hookrightarrow s_3] = [D \hookrightarrow D \hookrightarrow s_3] \oplus X1 \\
&\Rightarrow \{s_3 :: [D \hookrightarrow s_3]\} \oplus X1 = \{s_3 :: X1[D \hookrightarrow s_3]\}
\end{aligned}$$

Now we have a recursive equation to solve:

$$\begin{aligned}
X1[D \hookrightarrow s_3] &= \{s_3 :: [\mathtt{0} \hookrightarrow s_3]\} \cup \{s_3 :: X1[D \hookrightarrow s_3]\} \\
&= \{s_3^+ :: [\mathtt{0} \hookrightarrow s_3]\}
\end{aligned}$$

Using this result, we obtain:

$$\begin{aligned}
X2[D \hookrightarrow s_0] &= \{s_0 :: s_3^+ :: [\mathtt{0} \hookrightarrow s_3]\} \\
X2[s_0] &= \{s_0 :: [\mathtt{0} \hookrightarrow s_3], s_0 :: s_3^+ :: [\mathtt{0} \hookrightarrow s_3]\} \\
X1[s_0] &= \{[\mathtt{0} \hookrightarrow s_0], s_0 :: [\mathtt{0} \hookrightarrow s_3], s_0 :: s_3^+ :: [\mathtt{0} \hookrightarrow s_3]\} \\
X3[s_0] &= \{[\mathtt{!} \hookrightarrow \mathtt{0} \hookrightarrow s_0], s_0 :: [\mathtt{!} \hookrightarrow \mathtt{0} \hookrightarrow s_3], s_0 :: s_3^+ :: [\mathtt{!} \hookrightarrow \mathtt{0} \hookrightarrow s_3]\} \\
&= \{[s_6^{true}]\}
\end{aligned}$$

$$\begin{aligned}
&\text{since } \{[\mathtt{!} \hookrightarrow \mathtt{0} \hookrightarrow s_0]\} \Rightarrow \{s_0 :: [\mathtt{!} \hookrightarrow s_1]\} \Rightarrow \{[\mathtt{!} \hookrightarrow D \hookrightarrow s_0^{true}]\} \\
&\qquad \Rightarrow \{s_0^{true} :: [\mathtt{!} \hookrightarrow s_5]\} \Rightarrow \{[\mathtt{!} \hookrightarrow B \hookrightarrow s_0^{true}]\} \Rightarrow \{[s_6^{true}]\} \\
&\text{and } \{s_0 :: [\mathtt{!} \hookrightarrow \mathtt{0} \hookrightarrow s_3]\} \Rightarrow \{s_0 :: s_3 :: [\mathtt{!} \hookrightarrow s_1]\} \Rightarrow \{s_0 :: [\mathtt{!} \hookrightarrow D \hookrightarrow s_3^{true}]\} \\
&\qquad \Rightarrow \{s_0 :: s_3^{true} :: [\mathtt{!} \hookrightarrow s_5]\} \Rightarrow \{s_0 :: [\mathtt{!} \hookrightarrow B \hookrightarrow s_3^{true}]\} \\
&\qquad \Rightarrow \{s_0 :: s_3^{true} :: [\mathtt{!} \hookrightarrow s_4]\} \Rightarrow \{[\mathtt{!} \hookrightarrow B \hookrightarrow s_0^{true}]\} \Rightarrow \{[s_6^{true}]\} \\
&\text{and } \{s_0 :: s_3^+ :: [\mathtt{!} \hookrightarrow \mathtt{0} \hookrightarrow s_3]\} \Rightarrow \{s_0 :: s_3^+ :: s_3 :: [\mathtt{!} \hookrightarrow s_1]\} \\
&\qquad \Rightarrow \{s_0 :: s_3^+ :: [\mathtt{!} \hookrightarrow D \hookrightarrow s_3^{true}]\} \Rightarrow \{s_0 :: s_3^+ :: s_3^{true} :: [\mathtt{!} \hookrightarrow s_5]\} \\
&\qquad \Rightarrow \{s_0 :: s_3^+ :: [\mathtt{!} \hookrightarrow B \hookrightarrow s_3^{true}]\} \Rightarrow \{s_0 :: s_3^+ :: s_3^{true} :: [\mathtt{!} \hookrightarrow s_4]\} \\
&\qquad \Rightarrow \{s_0 :: [\mathtt{!} \hookrightarrow B \hookrightarrow s_3^{true}], s_0 :: s_3^+ :: [\mathtt{!} \hookrightarrow B \hookrightarrow s_3^{true}]\} \\
&\qquad\quad \text{(second set element adds nothing to fixed point)} \\
&\qquad \Rightarrow \{s_0 :: s_3^{true} :: [\mathtt{!} \hookrightarrow s_4]\} \Rightarrow \{[\mathtt{!} \hookrightarrow B \hookrightarrow s_0^{true}]\} \Rightarrow \{[s_6^{true}]\}
\end{aligned}$$

This proves that all possible string values of `x` at the end are well-structured B-phrases and even-valued. The approach is well suited to "type checking" XML-like documents; this application is currently under investigation.

10 Conclusion

The worklist algorithm for abstract parsing discussed in this paper has been implemented for PHP applications that dynamically generate HTML documents. A scannerless LALR(1) parsing table for an HTML grammar written up to character level is automatically generated by a parser generater, and a set of flow equations are generated from the PHP application to be analyzed. Our abstract parser then takes the flow equations, the parsing table, and a hot spot and parses the set of all documents dynamically generated at the given hot spot. In addition, our abstract parser builds a set of abstract syntax trees of the documents for the use of further analyses. The current implementation has been applied to a suite of PHP applications publicly available and has successfully identified multiple parse erorrs in a reasonable execution time with a few predictable false positives [7].

The extensions proposed in this paper, such as dealing with destructive string operators, composing scanner with the parser, modular abstract parsing with the existence of unknown string variables, string filtering through conditionals, and semantic processing such as type checking and taint analysis, are currently being implemented or are planned for implementation in the near future. The extensions are expected to remove false positives observed from our initial implementation and to make abstract parsing more practical and useful.

Acknowledgements: We thank Carolyn Talcott for her decades of leadership in programming-languages research and dedicate this paper to her on the occasion of her 60th birthday.

Conversations with Anders Møller, Se-won Kim, and Kwangkuen Yi and his group at Seoul National University have been helpful. We thank GTOne's Soo-Yong Lee for his inspiration and support.

References

1. G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *Proc. Int'l. Conf. Software Maintenance, Oxford*, 1999.
2. C. Brabrand, A. Møller, and M.I. Schwartzbach. The <bigwig> project. *ACM Trans. Internet Technology*, 2, 2002.
3. T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In *Proc. Asian Symp. Prog. Lang. and Systems*, pages 374–388. Springer LNCS 4279, 2006.
4. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Static analysis for dynamic XML. In *Proc. PLAN-X-02*, 2002.
5. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Extending Java for high-level web service construction. *ACM TOPLAS*, 25, 2003.
6. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Precise analysis of string expressions. In *Proc. SAS'03*, 2003.
7. 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.
8. E. Duesterwald, R. Gupta, and M.L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM TOPLAS*, 19:992–1030, 1997.
9. C.F. Goldfarb. *The SGML Handbook*. Oxford Univ. Press, 1991.
10. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engg.*, 1995.
11. N. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4*, pages 527–636. Oxford Univ. Press, 1995.
12. 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.
13. C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Proc. International Symp. Static Analysis*, pages 336–352. Springer LNCS 4134, 2006.

14. 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.
15. Y. Minimide and A. Tozawa. XML validation for context-free grammars. In *Proc. Asian Symp. Prog. Lang. and Systems*, pages 357–373. Springer LNCS 4279, 2006.
16. A. Møller and M. Schwarz. HTML validation of context-free languages. Technical report, Computer Science Dept., Aarhus University, 2010.
17. T. Nishiyama and Y. Minimide. A translation from the HTML DTD into a regular hedge grammar. In *Proc. 13th Int. Conf. on Implementation and Applications of Automata*, pages 122–131. Springer LNCS 5148, 2008.
18. P. Thiemann. Grammar-based analysis of string expressions. In *Proc. ACM workshop Types in languages design and implementation*, pages 59–70, 2005.
19. 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.
20. G. Wassermann and Z. Su. The essence of command injection attacks in web applications. In *Proc. 33d ACM Symp. POPL*, pages 372–382, 2006.
21. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. ACM PLDI*, pages 32–41, 2007.

Appendix: Concrete, collecting, and abstract semantics

A source program computes an output store that maps variables to strings. The *concrete collecting semantics* [11] defines a set of stores for each program point (command line); the collecting semantics is then abstracted in the usual fashion so that it computes, for each program point, a single store that maps each variable to a set of strings. For the example in Figure 1, we have

$$\begin{array}{l}
p_0 == \texttt{x} \mapsto \{\texttt{'a'}\} \\
p_r == \texttt{x} \mapsto \{\texttt{'a'}\},\;\; \texttt{r} \mapsto \{\texttt{']'}\} \\
p_1 == \texttt{x} \mapsto \{\texttt{'['}^i\texttt{'a''']'}^i \mid i \geq 0\},\;\; \texttt{r} \mapsto \{\texttt{']'}\} \\
p_2 == p_1 == p_3
\end{array}$$

The collecting semantics is overapproximated by the *data-flow semantics*, which uses flow equations to define the set of strings denoted by each variable at each program point. In Figure 1, the listed data-flow equations are a shorthand for this fuller form:

$$\begin{array}{ll}
X0 = \texttt{a} & \\
Rr = \texttt{]} & Xr = X0 \\
X1 = Xr \sqcup X2 & R1 = Rr \\
X2 = \texttt{[} \cdot X1 \cdot \texttt{]} & R2 = R1 \\
X3 = X1 & R3 = R1
\end{array}$$

The least-fixed-point solution is computed in the domain of sets of strings. Since the example ignores the loop test, the data-flow semantics computes the same sets as the collecting semantics.

Let Σ name the states in the LR(k)-parser's controller. A parse stack has form, $s_1 :: s_2 :: \cdots :: s_j :: [c]$, $j \geq 0$, where each $s_i \in \Sigma$, and the top, current parse state, $[c] \in \mathit{Configuration}$, has the form, $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, $0 < j < k$, where the ℓ_is are input symbols and $s \in \Sigma$.

Function $\gamma : \mathit{ParseStack} \rightarrow (\Sigma \times \mathcal{P}(\mathit{String}))$ concretizes a parse stack into the start state and the string(s) that generate the stack:

$$\gamma(st) = (s_0, T) \text{ such that } \mathit{LRparse}(t, s_0) = st \text{ and } t \in T$$

The function, $\gamma^* : \mathcal{P}(ParseStack) \rightarrow \mathcal{P}(\Sigma \times String)$, is the induced lift.

The *abstract-parse interpretation*, $\mathcal{X}$, computes the set of parse stacks denoted by each variable at each program point: For flow equation, $Xi = E_i$, the function, $\mathcal{X}_i : Configuration \rightarrow \mathcal{P}(ParseStack)$, is defined $\mathcal{X}_i[s] = \llbracket E_i \rrbracket[c]$, where

$$\llbracket \mathtt{a} \rrbracket[c] = \{rewrite[\mathtt{a} \hookrightarrow c]\}, \text{ where } \mathtt{a} \text{ is a terminal symbol}$$

$$\llbracket E_1 \sqcup E_2 \rrbracket[c] = \llbracket E_1 \rrbracket[c] \cup \llbracket E_2 \rrbracket[c]$$

$$\llbracket Xj \rrbracket[c] = \llbracket E_j \rrbracket[c], \text{ where } Xj = E_j \text{ is the flow equation for } Xj$$

$$\llbracket E_1 \cdot E_2 \rrbracket[c] = \{rewrite(p') \mid p' \in (\llbracket E_1 \rrbracket[c] \oplus \llbracket E_2 \rrbracket)\},$$

$$\quad \text{where } S \oplus g = \{tail(p) :: g(head(p)) \mid p \in S\}, \; S \in \mathcal{P}(ParseStack)$$

where $rewrite(st)$ repeatedly applies the shift/reduce rules, $\Rightarrow$, to st until a normal form is achieved.

Using γ, we can prove the abstract-parse interpretation sound with respect to the concrete collecting semantics.

The abstract-parse interpretation is made finitely convergent by abstracting the domain, $\mathcal{P}(ParseStack)$, into the domain of sets of subgraphs of the LR(k)-parser automaton: Represent a stack, $st = s_1 :: s_2 :: \cdots :: s_j :: [c]$, as the linked path, $pathst = \;\; \leftarrow s_1 \leftarrow s_2 \leftarrow \cdots \leftarrow s_j \leftarrow [c] \leftarrow$. Define $fold : ParseStack \rightarrow ParserSubgraph$ as $fold(pathst) = \;\; \longleftarrow^1 G \longleftarrow^j [c] \leftarrow,$ where G is the smallest subgraph in the parser automaton that covers the path from s_1 to s_j; $\longleftarrow^1$ is an out-arc from node s_1; and $\longleftarrow^j$ is an in-arc into node s_j. The answer graph is computed by folding repeating states in the argument path and preserving all arcs.

$fold$'s definition is easily generalized from folding paths to folding graphs — merge repeating nodes and retain the arcs.

The finitely convergent abstract-parse interpretation is defined in terms of $fold$; we modify the semantics of this one clause of the abstract-parse interpretation:

$$\begin{aligned} &\llbracket Xj \rrbracket[c] = fold^*(\llbracket E_j \rrbracket[c]), \quad \text{where } Xj = E_j \text{ is the flow equation for } Xj \\ &\quad \text{and } fold^*(T) = \{fold(t) \mid t \in T\} \end{aligned}$$

A set of subgraphs might be further abstracted into a single graph by unioning the graphs in the set into one graph — merge the graphs' like-named nodes and preserve the edges.