ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

ROSAEC MEMO
2009-001
June 9, 2009

Abstract Parsing for Two-staged Languages with Concatenation

Soonho Kong, Wontae Cho, Kwangkeun Yi
{soon,wtchoi,kwang}@ropas.snu.ac.kr
Seoul National University

June 9, 2009

Abstract

This article, based on Doh, Kim, and Schmidt's "abstract parsing" technique, presents an abstract interpretation for statically checking the syntax of generated code in two-staged programs. Abstract parsing is a static analysis technique for checking the syntax of generated strings. We adopt this technique for two-staged programming languages and formulate it in the abstract interpretation framework. We parameterize our analysis with the abstract domain so that one can choose the abstract domain as long as it satisfies the condition we provide. We also present an instance of the abstract domain, namely an abstract parse stack and its widening with k-cutting.

1 Introduction

1.1 Motivation

For programs that generate and run programs during execution, statically checking the program safety is a challenge. We need to check the safety of generated programs as well as that of the immediate target program. Checking the safety must include checking the programs resulting from evaluating programs.

The semantic safety of such multi-staged programs can be achieved in part by a static type system as reported in [4, 15, 18, 25]. A sound static type system assures that program as data as well as the immediate target program will not have a type error during their executions.

In such a static type system, syntactic errors in the generated code are not an issue. The considered target language is such that primitive code fragments and their compositions are always syntactically correct.

In reality though (as in most web-programming or scripting languages such as PHP, Python, Ruby, Perl, and Javascript), if code is represented as a string and code composition is achieved by string concatenation, syntactically checking the code string value is the foremost issue in the static safety check of such multi-staged programs.

Recently, Doh, Kim, and Schmidt reported a powerful technique called "abstract parsing" [16] that statically analyzes the string values from programs. In abstract parsing, to statically check the generated string is to simulate the parsing actions for possible string values.

In this paper, we report a formalization of abstract parsing in the abstract interpretation framework [10, 11, 12] for two-staged programming languages. Our contribution is to lay a basis to expose the power and, if any, limitation of abstract parsing as static analysis for multi-staged languages.

`x = 'a'`	$X_0 = \texttt{a}$
`l = '['`	$L = \texttt{[}$
`r = ']'`	$R = \texttt{]}$
`x = l . x`	$X_1 = L.X_0$

Figure 1: Example program (left) and its data-flow equations (right)

1.2 Abstract Parsing

We here review the abstract parsing idea of [16]. Suppose we want to check that strings generated by the program in Figure 1 conform to the following grammar.

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

Abstract parsing derives data-flow equations from the program as in Figure 1. The equation variables are treated as functions that map an input parse state to an output parse stack. They are solved using the goto controller of an LR parser for the grammar, shown in Figure 2.

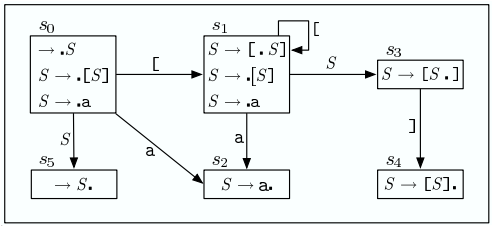


Figure 2: Goto controller of the LR(0) parser for $S \to \texttt{a} \mid \texttt{[}S\texttt{]}$. (from [16])

Suppose we want to check that X_1 will accept strings of the target grammar. The analysis starts with $X_1(s_0)$ where s_0 is an initial parse state. To solve

$$X_1(s_0) = (L.X_0)(s_0),$$

the analysis first computes $L(s_0)$. With the state s_0 and the token "`[`" the goto controller returns $goto(s_0, \texttt{[}) = s_1$. Having $L(s_0) = s_1$, the analysis computes $X_0(s_1)$. After consuming the token "`a`" and moving to the parse state s_2, parser reduces with $S \to \texttt{a}$ and moves the parse state back to s_1. Then $goto(s_1, S) = s_3$ yields $X_0(s_1) = s_3$. Therefore we have

$$X_1(s_0) = s_3 s_1$$

and the analysis concludes that X_1 has a string unacceptable for the grammar because state s_3 is not the accept state.

1.3 Contribution

- We formulate this abstract parsing idea in the abstract interpretation framework for two-staged languages with concatenation. By this formulation we can see what approximations are involved in abstract parsing and what limitations (as a static analysis) to expect from the abstract parsing technique.

  Based on the abstract interpretation framework, we present a concise and elegant perspective on the core idea of abstract parsing. In the original work [16], code is abstracted

into the parse stack and the special operator "$*$" is needed to handle string concatenation. In our formulation, however, we abstract code into a function which maps an input parse stack to an output parse stack. Code concatenation is handled simply by function composition.

- We generalize the abstract-parsing abstract interpretation, as usual, by parameterizing the abstract domain of parse stacks.

  This generalization separates the core idea and its implementation of abstract parsing. By choosing an appropriate abstract domain, one can control the analysis precision and cost.

1.4 Organization

Section 2 introduces the syntax and semantics of our target two-staged language with concatenation. Section 3 presents concrete parsing semantics with LR(k) parsing. Section 4 presents abstract parsing semantics and its parameterized framework. Section 5 presents a concrete example of the abstract domain which can be used to instantiate the framework. Section 6 reviews related work and Section 7 concludes.

2 Two-staged Language

We consider a two-staged language with concatenation. The language is an imaginary, first-order language whose only value is code. The language is minimal, so as not to distract our focus on formalizing the abstract parsing method. For example, loops and conditional jumps are without the condition expression, for which abstract interpretation anyway considers all iterations and all branches.

2.1 Syntax and Semantics

A program is an expression e:

$$e \in \mathit{Exp} ::= x \mid \texttt{let}\ x\ e_1\ e_2 \mid \texttt{or}\ e_1\ e_2 \mid \texttt{re}\ x\ e_1\ e_2\ e_3 \mid \texttt{`}f$$

An expression can contain code fragments f:

$$f \in \mathit{Frag} ::= x \mid \texttt{let} \mid \texttt{or} \mid \texttt{re} \mid \texttt{(} \mid \texttt{)} \mid f_1.f_2 \mid \texttt{,}e$$

Operational semantics of the target language is shown in Figure 3. Expression $\texttt{or}\ e_1\ e_2$ is for branches. It could be the value of e_1 or the value of e_2. Expression $\texttt{re}\ x\ e_1\ e_2\ e_3$ is for loops. Variable x has the value of e_1 as its initial value. Loop body e_2 is iterated ≥ 0 times. The result of each iteration e_2 will be bound to x in e_2 for next iteration or in e_3 for the result of the loop. Backquote form $\texttt{`}f$ is for code fragment $f$. We construct the fragment by using the following tokens: variables, `let`, `or`, `re`, `(`, and `)`. Compound fragment $f_1.f_2$ concatenates two code fragments $f_1$ and $f_2$. Comma fragment $\texttt{,}e$ first evaluates $e$ then substitutes its result code value for itself. Note that the meaning of $\texttt{`}f$ and $\texttt{,}e$ is the same as in LISP's quasi-quotation system.

2.2 Example Program

In our language, it is possible to write a program generating mal-formed code. For instance, the following program generates "`a b`" (after zero iterations), "`or a b`" (after one iteration), "`or or a b`" (after two iterations), and so on.

```
re x `a `(or . ,x ) `(,x . b)
```

$$\sigma \in \mathit{Env} = \mathit{Var} \to \mathit{Code}$$
$$v \in \mathit{Code} = \mathit{Token\ sequence}$$
$$e \in \mathit{Exp}$$
$$f \in \mathit{Frag}$$

$\boxed{\sigma \vdash^0 e \Rightarrow v}$

$$\sigma \vdash^0 x \Rightarrow \sigma(x) \qquad \text{(variable)}$$

$$\frac{\sigma \vdash^0 e_1 \Rightarrow v \qquad \sigma[x \mapsto v] \vdash^0 e_2 \Rightarrow v'}{\sigma \vdash^0 \texttt{let}\ x\ e_1\ e_2 \Rightarrow v'} \qquad \text{(let binding)}$$

$$\frac{\sigma \vdash^0 e_1 \Rightarrow v}{\sigma \vdash^0 \texttt{or}\ e_1\ e_2 \Rightarrow v} \qquad \frac{\sigma \vdash^0 e_2 \Rightarrow v}{\sigma \vdash^0 \texttt{or}\ e_1\ e_2 \Rightarrow v} \qquad \text{(branch)}$$

$$\frac{\sigma \vdash^0 e_1 \Rightarrow v \qquad \sigma[x \mapsto v] \vdash^0 \mathit{loop}\ x\ e_2\ e_3 \Rightarrow v'}{\sigma \vdash^0 \texttt{re}\ x\ e_1\ e_2\ e_3 \Rightarrow v'} \qquad \text{(loop)}$$

$$\frac{\sigma \vdash^0 e_2 \Rightarrow v \qquad \sigma[x \mapsto v] \vdash^0 \mathit{loop}\ x\ e_2\ e_3 \Rightarrow v'}{\sigma \vdash^0 \mathit{loop}\ x\ e_2\ e_3 \Rightarrow v'}$$

$$\frac{\sigma \vdash^0 e_3 \Rightarrow v}{\sigma \vdash^0 \mathit{loop}\ x\ e_2\ e_3 \Rightarrow v}$$

$$\frac{\sigma \vdash^1 f \Rightarrow v}{\sigma \vdash^0 \text{`}f \Rightarrow v} \qquad \text{(back quote)}$$

$\boxed{\sigma \vdash^1 f \Rightarrow v}$

$$\frac{}{\sigma \vdash^1 x \Rightarrow x} \qquad \frac{}{\sigma \vdash^1 \texttt{let} \Rightarrow \texttt{let}} \qquad \text{(token)}$$

$$\frac{}{\sigma \vdash^1 \texttt{or} \Rightarrow \texttt{or}} \qquad \frac{}{\sigma \vdash^1 \texttt{re} \Rightarrow \texttt{re}}$$

$$\frac{}{\sigma \vdash^1 \texttt{(} \Rightarrow \texttt{(}} \qquad \frac{}{\sigma \vdash^1 \texttt{)} \Rightarrow \texttt{)}}$$

$$\frac{\sigma \vdash^1 f_1 \Rightarrow v_1 \qquad \sigma \vdash^1 f_2 \Rightarrow v_2}{\sigma \vdash^1 f_1.f_2 \Rightarrow v_1 v_2} \qquad \text{(concatenation)}$$

$$\frac{\sigma \vdash^0 e \Rightarrow v}{\sigma \vdash^1 \texttt{,}e \Rightarrow v} \qquad \text{(comma)}$$

Figure 3: Operational semantics of the target language.

Only "`or a b`" is correct and the rest of them have a syntax error.

However the following program generates "`a`" (after zero iterations), "`or a b`" (after one iteration), "`or (or a b) b`" (after two iterations), and so on,

```
re x `a `(or . ,x . b) x
```

and all of them are syntactically correct.

2.3 Collecting Semantics and its Abstraction Plan

The collecting semantics of the language is defined as follows. This semantics is the natural set extension for the sets of environments. The *fix* operator is the usual least fixpoint operator to capture all the iteration results from loops.

$$
\begin{aligned}
\mathit{Code} &= \mathit{Token\ sequence} \\
\sigma \in \mathit{Env} &= \mathit{Var} \to \mathit{Code} \\
[\![e]\!]^0 &\in 2^{\mathit{Env}} \to 2^{\mathit{Code}} \\
[\![f]\!]^1 &\in 2^{\mathit{Env}} \to 2^{\mathit{Code}}
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!]^0\Sigma &= \{\sigma(x) \mid \sigma \in \Sigma\} \\
[\![\texttt{let}\ x\ e_1\ e_2]\!]^0\Sigma &= \bigcup_{\sigma\in\Sigma} \bigcup_{c\in[\![e_1]\!]^0\{\sigma\}} [\![e_2]\!]^0\{\sigma[x \mapsto c]\} \\
[\![\texttt{or}\ e_1\ e_2]\!]^0\Sigma &= [\![e_1]\!]^0\Sigma \cup [\![e_2]\!]^0\Sigma \\
[\![\texttt{re}\ x\ e_1\ e_2\ e_3]\!]^0\Sigma &= \bigcup_{\sigma\in\Sigma} [\![e_3]\!]^0\{\sigma[x \mapsto c] \mid c \in \\
&\qquad \mathit{fix}\lambda C.[\![e_1]\!]^0\{\sigma\} \cup [\![e_2]\!]^0\{\sigma[x \mapsto c'] \mid c' \in C\}\} \\
[\![\texttt{`}\,f]\!]^0\Sigma &= [\![f]\!]^1\Sigma \\
[\![x]\!]^1\Sigma &= \{x\} \\
[\![\texttt{let}]\!]^1\Sigma &= \{\texttt{let}\} \\
[\![\texttt{or}]\!]^1\Sigma &= \{\texttt{or}\} \\
[\![\texttt{re}]\!]^1\Sigma &= \{\texttt{re}\} \\
[\![\texttt{(}]\!]^1\Sigma &= \{\texttt{(}\} \\
[\![\texttt{)}]\!]^1\Sigma &= \{\texttt{)}\} \\
[\![f_1.f_2]\!]^1\Sigma &= \bigcup_{\sigma\in\Sigma} \{xy \mid x \in [\![f_1]\!]^1\{\sigma\} \wedge y \in [\![f_2]\!]^1\{\sigma\}\} \\
[\![\texttt{,}e]\!]^1\Sigma &= [\![e]\!]^0\Sigma
\end{aligned}
$$

From the collecting semantics above, we derive a series of abstract semantics. From the collecting semantics' semantic domain

$$2^{\mathit{Var}\to\mathit{Code}} \to 2^{\mathit{Code}},$$

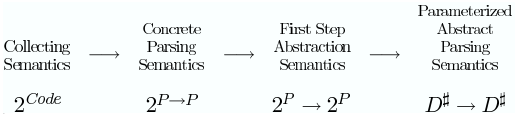
the powerset environment $2^{\mathit{Var}\to\mathit{Code}}$ is abstracted into $\mathit{Var} \to 2^{\mathit{Code}}$, i.e., the semantic domain becomes

$$(\mathit{Var} \to 2^{\mathit{Code}}) \to 2^{\mathit{Code}}.$$

Now the abstraction of 2^{Code} becomes the essential part of the abstract interpretation design. Before we abstract 2^{Code}, we formulate a code fragment as a function that maps a parse stack

to a parse stack. We call this formulation "concrete parsing" (Section 3). That is, 2^{Code} becomes $2^{P \to P}$ (where P is the set of parse stacks). Then we abstract $2^{P \to P}$ into $2^P \to 2^P$ (Section 4). Lastly, we present an abstract-parsing abstract interpretation that parameterizes an abstract domain $D^\sharp$ of 2^P.

In summary, this series of abstraction steps for the value domain in the semantics is:



3 Concrete Parsing

3.1 Analyze-and-parse Strategy

We take the analyze-*and*-parse strategy in abstract parsing [16] into our semantics. The semantics simulates the parsing operations. It is compared to the analyze-*then*-parse strategy which analyzes the program, calculates approximated set of code, then parses them.

Analyze-*and*-parse strategy is more efficient than analyze-*then*-parse strategy as reported in [16]. In analyze-*then*-parse strategy, code is abstracted into a grammar. Then it checks whether the abstracted grammar is included in the reference grammar or not. However, grammar inclusion check is more expensive than parsing. In addition, analyze-*and*-parse directly computes parsing information without approximating the code into the grammar.

We formulate the analyze-*and*-parse strategy in our semantics. The parsing domain is constructed as an abstract domain where code is abstracted into parsing information. We abstract the parsing domain into an abstract parsing domain to control the precision and cost of analysis and to make sure the analysis terminates.

Since our semantics uses an LR(k) parser as a component, it is essential to review its key concepts.

3.2 LR Parsing

The LR(k) parsing technique [1] is an efficient way to determine whether the string conforms to the given grammar or not. An LR parser is a state machine which consists of a parse stack, an action table, and a goto table. The set of parse states $\Sigma = \{s_1, s_2, \ldots, s_n\}$ is defined by parser generator from the given grammar. Parse stack $p \in P = \Sigma^+$ is a sequence of parse states which it has been in. Two special parse stacks p_{init} and p_{acc} are defined. Parsing starts with the initial parse stack p_{init}. Successful parsing should stop at the accept parse stack p_{acc}. Otherwise it indicates that the parsed string does not conform to the given grammar. String representation "$s_{top} \ldots s_{bot}$" denotes a parse stack whose top state is s_{top} and bottom state is s_{bot}. The action table decides which operation (shift/reduce) to perform from the current state and current token. The goto table determines the state to push after we pop states in the reduce operation.

The process of parsing is a composition of the atomic function $parse\_action : P \to Token \to P$ which is described in Algorithm 1. It returns the parse stack from the given parse stack p and input token t.

The parsing process $parse : P \to Token\,sequence \to P$ is a composition of the $parse\_action$.

$$\begin{aligned} &parse(p, t_1 \ldots t_n) \\ &= parse\_action(\ldots(parse\_action(p, t_1)), \ldots, t_n) \end{aligned}$$

A parser gets the input code c and returns the parse stack $parse(p_{init}, c)$.

Algorithm 1 *parse_action* algorithm

```
1: procedure parse_action(p, t)
2:     s_top ← the state on top of stack p
3:     if ACTION[s_top, t] = shift s then
4:         push s onto the stack p
5:         return p
6:     else if ACTION[s_top, t] = reduce A → β then
7:         pop |β| symbol off the stack p
8:         s_top ← the state on top of stack p
9:         push GOTO[s_top, A] onto the stack p
10:        return parse_action(p, t)
11:    end if
12: end procedure
```

3.3 Concrete Parsing Domain : Value $V_P = 2^{P \to P}$

We define the concrete parsing domain. It is "concrete" in that we use the same parse stack defined in LR(k) parsing.

Because LR(k) parsing computes a parse stack, it is tempting to choose the parse stack $parse(p_{init}, c)$ as an abstraction of the code c. However this setting causes a problem when we handle the concatenation $x.y$. Let $p_x = parse(p_{init}, x)$ and $p_y = parse(p_{init}, y)$ be the resulting parse stacks for the code fragments x and y. The parse stack p for the concatenation $x.y$ is computed as

$$\begin{aligned} p &= parse(p_{init}, x.y) \\ &= parse(parse(p_{init}, x), y) \\ &= parse(p_x, y). \end{aligned}$$

However, what we have is $p_y = parse(p_{init}, y)$ not $parse(p_x, y)$. The $parse(p_{init}, y)$ is the parse stack after parsing y from the initial stack not from the p_x stack. We cannot directly compute p from p_x and p_y.

Abstracting the code c into the parse stack transition function $\lambda p.parse(p, c)$ solves the above concatenation problem elegantly. Let f_x and f_y be the parse stack transition functions for code fragments x and y respectively. Then we have

$$\begin{aligned} f_x &= \lambda p.parse(p, x) \\ f_y &= \lambda p.parse(p, y). \end{aligned}$$

With the two parse stack transition functions f_x and f_y, we construct the parse stack transition function $f_{x.y}$ as follows.

$$\begin{aligned} f_{x.y} &= \lambda p.parse(p, x.y) \\ &= \lambda p.parse(parse(p, x), y) \\ &= (\lambda p.parse(p, y)) \circ (\lambda p.parse(p, x)) \\ &= f_y \circ f_x \end{aligned}$$

3.4 Concrete Parsing Semantics

Using the abstraction from *Code* to $P \to P$, the Galois connection $2^{Code} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} V_P = 2^{P \to P}$ is established as follows.

$$\alpha = \lambda C.\{\lambda p.parse(p, c) \mid c \in C\}$$

$$\gamma = \lambda F. \bigcup_{f \in F} \{c \mid \forall p \in P. parse(p, c) = f(p)\}$$

Concrete parsing semantics is derived from the collecting semantics as follows.

$$\begin{aligned} P &= \text{the set of parse stacks} \\ V_P &= 2^{P \to P} \\ \sigma \in Env_P &= Var \to V_P \\ [\![e]\!]^0_P &\in Env_P \to V_P \\ [\![f]\!]^1_P &\in Env_P \to V_P \end{aligned}$$

$$\begin{aligned} [\![x]\!]^0_P\sigma &= \sigma(x) \\ [\![\texttt{let } x\ e_1\ e_2]\!]^0_P\sigma &= [\![e_2]\!]^0_P(\sigma[x \mapsto [\![e_1]\!]^0_P\sigma]) \\ [\![\texttt{or } e_1\ e_2]\!]^0_P\sigma &= [\![e_1]\!]^0_P\sigma \cup [\![e_2]\!]^0_P\sigma \\ [\![\texttt{re } x\ e_1\ e_2\ e_3]\!]^0_P\sigma &= [\![e_3]\!]^0_P(\sigma[x \mapsto \\ &\qquad fix\ \lambda k.[\![e_1]\!]^0_P\sigma \cup [\![e_2]\!]^0_P(\sigma[x \mapsto k])]) \\ [\![\text{`} f]\!]^0_P\sigma &= [\![f]\!]^1_P\sigma \\ [\![t]\!]^1_P\sigma &= \{\lambda p. parse\_action(p, t)\} \\ [\![f_1.f_2]\!]^1_P\sigma &= \{p_2 \circ p_1 \mid p_1 \in [\![f_1]\!]^1_P\sigma \wedge p_2 \in [\![f_2]\!]^1_P\sigma\} \\ [\![,e]\!]^1_P\sigma &= [\![e]\!]^0_P\sigma \end{aligned}$$

4 Abstract Parsing

4.1 First Step Abstraction : Value $V_{\hat{P}} = 2^P \to 2^P$

First, we abstract the concrete parsing domain $V_P = 2^{P \to P}$ to $V_{\hat{P}} = 2^P \to 2^P$ by establishing the Galois connection $V_P \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} V_{\hat{P}}$ where

$$\begin{aligned} \alpha &= \lambda F. \lambda P. \bigcup_{p \in P} \{f(p) \mid f \in F\} \\ \gamma &= \lambda F. \bigcup \{S \mid \alpha(S) \subseteq f\}. \end{aligned}$$

Then we derive the abstract semantics for this domain as follows.

$$\begin{aligned} \sigma \in Env_{\hat{P}} &= Var \to V_{\hat{P}} \\ [\![e]\!]^0_{\hat{P}} &\in Env_{\hat{P}} \to V_{\hat{P}} \\ [\![f]\!]^1_{\hat{P}} &\in Env_{\hat{P}} \to V_{\hat{P}} \end{aligned}$$

$$\begin{aligned} [\![x]\!]^0_{\hat{P}} &= \sigma(x) \\ [\![\texttt{let } x\ e_1\ e_2]\!]^0_{\hat{P}}\sigma &= [\![e_2]\!]^0_{\hat{P}}(\sigma[x \mapsto [\![e_1]\!]^0_{\hat{P}}\sigma]) \\ [\![\texttt{or } e_1\ e_2]\!]^0_{\hat{P}}\sigma &= [\![e_1]\!]^0_{\hat{P}}\sigma \cup [\![e_2]\!]^0_{\hat{P}}\sigma \\ [\![\texttt{re } x\ e_1\ e_2\ e_3]\!]^0_{\hat{P}}\sigma &= [\![e_3]\!]^0_{\hat{P}}(\sigma[x \mapsto \\ &\qquad fix\ \lambda k.[\![e_1]\!]^0_{\hat{P}}\sigma \cup [\![e_2]\!]^0_{\hat{P}}(\sigma[x \mapsto k])]) \\ [\![\text{`} f]\!]^0_{\hat{P}}\sigma &= [\![f]\!]^1_{\hat{P}}\sigma \end{aligned}$$

$$\begin{aligned}
[\![t]\!]^1_{\hat{P}}\sigma &= \lambda P.Parse_action(P,t)\\
[\![f_1.f_2]\!]^1_{\hat{P}}\sigma &= [\![f_2]\!]^1_{\hat{P}}\sigma \circ [\![f_1]\!]^1_{\hat{P}}\sigma\\
[\![,e]\!]^1_{\hat{P}}\sigma &= [\![e]\!]^0_{\hat{P}}\sigma
\end{aligned}$$

where $Parse\_action : 2^P \to Token \to 2^P$ is the natural set extension of *parse_action*:

$$Parse\_action = \lambda P.\lambda t.\{parse\_action(p,t) \mid p \in P\}.$$

The abstract semantic function $[\![\cdot]\!]^0_{\hat{P}}$ is used to check whether generated code conforms to the grammar. For the given program e, we compute

$$S = [\![e]\!]^0_{\hat{P}}\sigma_0\{p_{init}\} : 2^P$$

where $\sigma_0 \in Env_{\hat{P}}$ is an empty environment. Then we compare S with $\{p_{acc}\}$. If they are equal, we conclude that the generated code in the given program conforms to the grammar. Otherwise, the analysis concludes that the program may generate syntactically incorrect code.

4.2 Parameterized Framework

We generalize the analysis by parameterizing abstract domain. Instead of abstracting 2^P (the powerset domain of parse stacks) into a particular domain, we provide conditions which the abstract domain for 2^P should satisfy. Then we define the semantic function on the abstract parsing domain.

Definition 1 (Abstract Parsing Domain)**.** *$V^\sharp = D^\sharp \to D^\sharp$ is an abstract parsing domain if an abstract domain $D^\sharp$ satisfies the following conditions [10, 11, 12].*

1. *$\langle D^\sharp, \sqsubseteq, \sqcup, \bot_{D^\sharp}\rangle$ is a CPO (Complete Partial Order).*
2. *Powerset domain of concrete parse stack 2^P and its abstract domain $D^\sharp$ are Galois connected via $\alpha_{2^P \to D^\sharp}$ and $\gamma_{D^\sharp \to 2^P}$.*
3. *$Parse\_action^\sharp : D^\sharp \to Token \to D^\sharp$ is a sound approximation of $Parse\_action$. That is, we have*

$$\begin{aligned}
&\forall P \in 2^P.\forall t \in Token.\\
&\alpha_{2^P \to D^\sharp}(Parse_action(P,t)) \sqsubseteq Parse_action^\sharp(\alpha_{2^P \to D^\sharp}(P), t)
\end{aligned}$$

The partial order $\sqsubseteq_{V^\sharp}$ and join operator $\sqcup_{V^\sharp}$ are defined pointwisely.

Definition 2. *Semantic function $[\![\cdot]\!]_{D^\sharp}$ on the abstract parsing domain $V^\sharp$ is defined as follows.*

$$\begin{aligned}
\sigma \in Env_{D^\sharp} &= Var \to V^\sharp\\
[\![e]\!]^0_{D^\sharp} &\in Env_{D^\sharp} \to V^\sharp\\
[\![f]\!]^1_{D^\sharp} &\in Env_{D^\sharp} \to V^\sharp
\end{aligned}$$

$$\begin{aligned}
[\![x]\!]^0_{D^\sharp}\sigma &= \sigma(x)\\
[\![\texttt{let}\ x\ e_1\ e_2]\!]^0_{D^\sharp}\sigma &= [\![e_2]\!]^0_{D^\sharp}(\sigma[x \mapsto [\![e_1]\!]^0_{D^\sharp}\sigma])\\
[\![\texttt{or}\ e_1\ e_2]\!]^0_{D^\sharp}\sigma &= [\![e_1]\!]^0_{D^\sharp}\sigma \sqcup [\![e_2]\!]^0_{D^\sharp}\sigma\\
[\![\texttt{re}\ x\ e_1\ e_2\ e_3]\!]^0_{D^\sharp}\sigma &= [\![e_3]\!]^0_{D^\sharp}(\sigma[x \mapsto\\
&\qquad fix\ \lambda k.[\![e_1]\!]^0_{D^\sharp}\sigma \sqcup [\![e_2]\!]^0_{D^\sharp}(\sigma[x \mapsto k])])
\end{aligned}$$

$$\begin{aligned}
[\![`f]\!]^0_{D^\sharp}\sigma &= [\![f]\!]^1_{D^\sharp}\sigma \\
[\![t]\!]^1_{D^\sharp}\sigma &= \lambda D.Parse_action^\sharp(D,t) \\
[\![f_1.f_2]\!]^1_{D^\sharp}\sigma &= [\![f_2]\!]^1_{D^\sharp}\sigma \circ [\![f_1]\!]^1_{D^\sharp}\sigma \\
[\![,e]\!]^1_{D^\sharp}\sigma &= [\![e]\!]^0_{D^\sharp}\sigma
\end{aligned}$$

Theorem 1 shows that $[\![\cdot]\!]_{D^\sharp}$ is a sound approximation of $[\![\cdot]\!]_{\hat{P}}$.

Theorem 1. *Semantic function $[\![\cdot]\!]_{D^\sharp}$ on the abstract parsing domain $V^\sharp$ is a sound approximation of $[\![\cdot]\!]_{\hat{P}}$. That is, we have*

$$\begin{aligned}
&\forall e \in Exp.\forall \sigma \in Env_{\hat{P}}. \\
&\alpha_{V_{\hat{P}}\to V^\sharp}([\![e]\!]_{\hat{P}}\sigma) \sqsubseteq [\![e]\!]_{D^\sharp}(\alpha_{Env_{\hat{P}}\to Env_{D^\sharp}}(\sigma))
\end{aligned}$$

where

$$\begin{aligned}
\alpha_{V_{\hat{P}}\to V^\sharp} &= \lambda F.\lambda D.\alpha_{2^P\to D^\sharp}(F(\gamma_{D^\sharp\to 2^P}(D))) \\
\alpha_{Env_{\hat{P}}\to Env_{D^\sharp}} &= \lambda\sigma.\lambda x.\alpha_{V_{\hat{P}}\to V^\sharp}(\sigma(x)).
\end{aligned}$$

Proof. By structural induction on e with the conditions that the abstract parsing domain $D^\sharp$ should satisfy. □

5 Instantiation : Powerset Domain of Abstract Parse Stack $\hat{P}$ with k-cutting

In this section, we introduce an instance of an abstract domain $D^\sharp$, which is $\hat{D}$, a powerset domain of abstract parse stack $\hat{P}$ and its widening with k-cutting. Thus the abstract parsing domain is

$$\hat{V} = \hat{D} \to \hat{D}.$$

First, we define abstract parse stack $\hat{P}$ which is an abstraction of concrete parse stack P. The abstract domain $\hat{D}$ is constructed with abstract parse stack $\hat{P}$. By establishing the Galois connection between 2^P and $\hat{D}$ and defining $Pars\widehat{e\_action}$, we show that $\hat{D}$ is an instance of $D^\sharp$. Finally, a widening operator is defined using k-cutting to guarantee the termination of analysis.

5.1 Abstract Parse Stack $\hat{P}$

Cut parse stack $\bar{P}$ is introduced to represent a parse stack which has been cut and only maintains top n states. Special state '–' $\notin \Sigma$ at the bottom of cut parse stack $\bar{p}$ indicates that it has been cut.

$$\bar{P} = \{p \cdot - \mid p \in \Sigma^*\}$$

Abstract parse stack $\hat{P}$ is defined as a union of concrete parse stack P and cut parse stack $\bar{P}$.

$$\hat{P} = P \cup \bar{P}$$

Given an abstract parse stack ρ, the function $prefix : \hat{P} \to \Sigma^*$ yields the longest prefix of ρ which does not contain special state '–'.

$$prefix(\rho) = \begin{cases} \rho & \text{if } \rho \in P \\ s_1 \dots s_n & \text{if } \rho = s_1 \dots s_n - \\ \epsilon \text{ (empty string)} & \rho = - \end{cases}$$

Then we define partial order $\sqsubseteq$ on $\hat{P}$ as follows.

$$\rho_1 \sqsubseteq_{\hat{P}} \rho_2 \stackrel{\text{def}}{=} prefix(\rho_1) \text{ starts with } prefix(\rho_2)$$

5.2 Abstract Domain : $\hat{D}$

Abstract domain $\hat{D}$ with its partial order $\sqsubseteq$ and join $\sqcup$ is defined as follows.

$$\begin{aligned}\hat{D} &= \{norm(\hat{d}) \mid \hat{d} \in 2^{\hat{P}}\} \\ \hat{d}_1 \sqsubseteq \hat{d}_2 &\stackrel{\text{def}}{=} \forall \rho_1 \in \hat{d}_1 . \exists \rho_2 \in \hat{d}_2 . \rho_1 \sqsubseteq_{\hat{P}} \rho_2 \\ \hat{d}_1 \sqcup \hat{d}_2 &\stackrel{\text{def}}{=} norm(\hat{d}_1 \cup \hat{d}_2)\end{aligned}$$

Normalization function *norm* is defined by

$$norm(\hat{d}) = \{\rho \in \hat{d} \mid \forall \rho' \in \hat{d} . \rho \not\sqsubset_{\hat{P}} \rho'\}$$

to ensure that elements in $norm(\hat{d})$ are the maximal elements of $\hat{d}$. It is necessary to eliminate non-maximal elements for binary relation $\sqsubseteq$ to be anti-symmetric and to be a partial order. For instance, $\{-, s_1 s_0 -\}$ contains non-maximal element $\{s_1 s_0 -\}$ since we have $s_1 s_0 - \sqsubseteq -$. If we allow $\{s_1 s_0 -, -\}$ in $\hat{D}$ without normalizing it into $\{-\}$, we have

$$\begin{aligned}(\{s_1 s_0 -, -\} \sqsubseteq \{-\}) \wedge (\{-\} \sqsubseteq \{s_1 s_0 -, -\}) \\ \not\Rightarrow \{s_1 s_0 -, -\} = \{-\}.\end{aligned}$$

Then partial order $\sqsubseteq$ on $\hat{D}$ becomes preorder since $\sqsubseteq$ is not anti-symmetric anymore.

5.3 Abstract Domain $\hat{D}$ as an Instance of $D^\sharp$

Abstract domain $\hat{D}$ is an instance of $D^\sharp$. To verify this, we need to show that $\hat{D}$ satisfies the conditions in Definition 1.

1. $\langle \hat{D}, \sqsubseteq, \sqcup, \phi \rangle$ is a CPO by definition of $\hat{D}$.
2. To establish a Galois connection between 2^P and $\hat{D}$, we first define the function *expand* : $\hat{P} \to 2^P$ as follows.

$$expand(\rho) = \begin{cases} \{\rho\} & \text{if } \rho \in P \\ \{prefix(\rho) \cdot p \mid p \in P\} & \text{if } \rho \in \bar{P} \end{cases}$$

   It expands an abstract parse stack ρ to all the concrete parse stacks which ρ can represent. For instance, $expand(s_1 -) = \{s_1 s_0, s_1 s_2, \dots\}$. Note that we have $\forall p \in expand(\rho) . p \sqsubseteq_{\hat{P}} \rho$ and therefore we get $expand(\rho) \sqsubseteq_{\hat{D}} \{\rho\}$.

   The function $Expand : \hat{D} \to 2^P$ is also defined as the natural set extension of *expand* function by

$$Expand(\hat{d}) = \bigcup_{\rho \in \hat{d}} expand(\rho).$$

   From the property of *expand*, it is clear that $Expand(\hat{d}) \sqsubseteq_{\hat{D}} \hat{d}$.

   Using the *Expand* function, we define the Galois connection $2^P \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{D}$ where

$$\begin{aligned}\alpha &= id \\ \gamma &= \lambda \hat{d} . Expand(\hat{d})\end{aligned}$$

   Proof. To prove α and γ constitute a Galois connection, it is sufficient to show that the following properties hold.

(a) Trivially, $\alpha = id$ is monotone.

(b) Monotonicity of γ is immediate from the monotonicity of *expand*.

(c) We show that $\forall \hat{d} \in \hat{D}.\alpha \circ \gamma(\hat{d}) \sqsubseteq \hat{d}$.

$$\begin{aligned}\alpha \circ \gamma(\hat{d}) &= \gamma(\hat{d}) \\ &= \mathit{Expand}(\hat{d}) \\ &\sqsubseteq \hat{d}\end{aligned}$$

(d) We show that $\forall P \in 2^P.P \subseteq \gamma \circ \alpha(P)$.

$$\begin{aligned}\gamma \circ \alpha(P) &= \gamma(P) \\ &= \mathit{Expand}(P) \\ &= \bigcup_{p \in P} \mathit{expand}(p) \\ &= \bigcup_{p \in P} p \\ &= P\end{aligned}$$

□

3. We first define $\widehat{\mathit{pars\_action}} : \hat{P} \to \mathit{Token} \to \hat{P}$ as in Algorithm 2. It is a modified version of the *parse_action* algorithm. The only modifications are adding lines 2 – 4 and lines 12 – 14 to handle $\bar{p} \in \bar{P}$ because the action and goto tables do not have an entry for the special state '–'. Note that $\forall p \in P.\forall t \in \mathit{Token}.\mathit{parse\_action}(p,t) = \widehat{\mathit{pars\_action}}(p,t)$.

Algorithm 2 $\widehat{\mathit{pars\_action}}$ algorithm

```
 1: procedure pars_action(ρ, t)
 2:     if ρ = – then
 3:         return ρ
 4:     end if
 5:     s_top ← the state on top of stack ρ
 6:     if ACTION[s_top, t] = shift s then
 7:         push s onto the stack ρ
 8:         return ρ
 9:     else if ACTION[s_top, t] = reduce A → β then
10:         pop |β| symbol off the stack ρ
11:         s_top ← the state on top of stack ρ
12:         if s_top = – then
13:             return ρ
14:         end if
15:         push GOTO[s_top, A] onto the stack ρ
16:         return pars_action(ρ, t)
17:     end if
18: end procedure
```

$\widehat{\mathit{Pars\_action}} : \hat{D} \to \mathit{Token} \to \hat{D}$ is defined as the natural set extension of $\widehat{\mathit{pars\_action}}$ as follows.

$$\widehat{\mathit{Pars\_action}} = \lambda \hat{d}.\lambda t.\{\widehat{\mathit{pars\_action}}(\rho, t) \mid \rho \in \hat{d}\}$$

$\widehat{\mathit{Pars\_action}}$ is a sound approximation of *Parse_action*.

Proof. For all $P \in 2^P$ and $t \in \mathit{Token}$, we have

$$\begin{aligned}\alpha_{2^P \to \hat{D}}(\mathit{Parse\_action}(P,t)) &= \mathit{Parse\_action}(P,t)\\ &= \{\mathit{parse\_action}(p,t) \mid p \in P\}\\ &= \{\mathit{pars\widehat{e\_a}ction}(p,t) \mid p \in P\}\\ &= \mathit{Pars\widehat{e\_a}ction}(P,t)\\ &= \mathit{Pars\widehat{e\_a}ction}(\alpha_{2^P \to \hat{D}}(P),t).\end{aligned}$$

□

5.4 Widening Operator on $\hat{D}$

The termination of the analysis parameterized with $\hat{D}$ is not guaranteed because $\hat{D}$ has infinite height. Instead of limiting the height of the domain, we use the widening method to achieve termination.

We define an operator $\nabla_{\hat{D}} : \hat{D} \times \hat{D} \to \hat{D}$ such that

$$A \nabla_{\hat{D}} B = \{\mathit{norm}(\mathit{cut}_k(\rho)) \mid \rho \in A \cup B\}$$

where cut_k is defined by

$$\mathit{cut}_k(\rho) = \begin{cases} \rho & \text{if } |\rho| \le k \\ s_1 \dots s_{k-1}{}^{-} & \text{if } \rho = s_1 \dots s_{k-1} s_k \dots s_n. \end{cases}$$

Theorem 2 shows that $\nabla_{\hat{D}}$ is a widening operator on $\hat{D}$.

Theorem 2. *$\nabla_{\hat{D}} : \hat{D} \times \hat{D} \to \hat{D}$ is a widening operator.*

Proof. We have to check if $\nabla_{\hat{D}}$ operator satisfies the widening conditions [10, 11, 12]:

(i) $\forall x, y \in \hat{D}.(x \sqsubseteq x \nabla_{\hat{D}} y) \wedge (y \sqsubseteq x \nabla_{\hat{D}} y)$.

(ii) for all increasing chains $x_0 \sqsubseteq x_1 \dots$, the increasing chain defined by $y_0 = x_0, \dots, y_{i+1} = y_i \nabla_{\hat{D}} x_{i+1}, \dots$ is not strictly increasing.

To prove (i), we observe that $\rho \sqsubseteq_{\hat{P}} \mathit{cut}_k(\rho)$ by definition of $\sqsubseteq_{\hat{P}}$ and cut_k. Using this we get

$$\begin{aligned} x &= \{\rho \mid \rho \in x\}\\ &\sqsubseteq \{\mathit{cut}_k(p) \mid p \in x\}\\ &\sqsubseteq \{\mathit{cut}_k(p) \mid p \in x \cup y\}\\ &= x \nabla_{\hat{D}} y. \end{aligned}$$

Proof for $y \sqsubseteq x \nabla_{\hat{D}} y$ is analogous.

To prove (ii), we observe that the range of $\nabla_{\hat{D}}$ operator is the finite set $\hat{P}' = \{\rho \in \hat{P} \mid |\rho| \le k\}$. For all $i \ge 1$, we have $y_i \in \hat{P}'$. Therefore the increasing chain $y_0, \dots, y_n, \dots$ is not strictly increasing. □

Using the widening operator $\nabla_{\hat{D}}$, we define a widening operator for $\hat{V} = \hat{D} \to \hat{D}$, such that

$$f \nabla_{\hat{V}} g = \lambda \hat{d}. \begin{cases} f(\hat{d}) \nabla_{\hat{D}} g(\hat{d}) & \text{if } \forall \rho \in \hat{d}.|\rho| \le l \\ \{-\} & \text{otherwise.} \end{cases}$$

where l is a constant to bound the set of meaningful entries finitely fixed in the resulting function. For those finite meaningful entries, we widen their images using the widening $\nabla_{\hat{D}}$. This $\nabla_{\hat{V}}$ operator is a widening operator and the analysis using this widening always terminates.

The analysis using the widening gives better precision than the one using the domain with finite height, for instance, $\hat{D}' = \{\hat{d} \in \hat{D} \mid |\hat{d}| \leq k\}$. The widening approach only restricts the length of the parse stack at the loop head. In the loop body, it allows arbitrary length of parse stacks. Let's consider the following program.

```
re x `y
   `(let . z . ,x
         . or . z . z)
   x
```

Using widening approach, it is sufficient to have cut threshold $k = 2$ to analyze the program without precision loss. In the loop body, the length of parse stack increases from one to seven. At the end of the loop body, however, the parser reduces the parse stack and its length becomes two. Since the maximum length of the parse stack at the loop head is two and the widening only occurs at the loop head, cut threshold $k = 2$ is enough.

However if we restrict the domain and use $\hat{D}$ with $k = 2$, we will lose precision while analyzing the loop body. A parse stack with special state '–' at the bottom will be introduced. To be as precise as the one using the widening approach, we need to increase k to seven.

6 Related Work

In this section, we discuss several areas of related work: multi-staged languages, string analysis, string verification, and code generation.

Multi-staged Languages According to Sheard [24], there are three representations for code in multi-staged languages – quasi-quote, algebraic data type, and strings.

Quasi-quote and algebraic data type representation preserve internal structure of the code and syntactic safety can be guaranteed.

Regarding semantic safety check, various static type systems for multi-staged languages are reported [2, 4, 5, 6, 13, 14, 15, 21, 22, 23, 25]. The static type system has been matured [18] to support almost all of the Lisp-like multi-staged programming practices.

However, using a string representation loses track of internal structure of the code and opens the syntactic safety problem. Even if string representation is strongly discouraged in the research community [24], it is used in most web-programming or scripting languages such as PHP, Python, Ruby, Perl, and Javascript.

String Analysis Before Doh, Kim, and Schmidt's abstract parsing technique [16], string analysis works [8, 19, 7] were all "analyze-*then*-parse" techniques.

Christensen et al. [8] presents an approach which abstracts the set of generated strings in a program into a regular grammar and performs a grammar inclusion check between the regular grammar and the reference grammar, which is a context-free grammar (CFG). Precision loss occurs when the generated strings are abstracted into the regular grammar.

Minamide [19] takes a slightly different approach. The context-free reference grammar is abstracted into a regular grammar by restricting the nesting depth of generated strings. The set of generated strings is abstracted into a context-free grammar. Then a grammar inclusion check is performed between the regular grammar and the context-free grammar. This approach is practical for HTML/XML document analysis because their nesting depth is usually restricted. However, it is not applicable in general.

Choi et al. [7] presents abstract interpretation based string analysis which uses a heuristic widening method and overcomes the difficulties of handling heap variables and context sensitivity. However, using a regular grammar as an abstraction results in precision loss.

String Verification String analysis techniques are used as a basis in string verification. Christodorescu et al. [9] and Wassermann et al. [26] present string verifiers based on the Christensen et al.'s string analysis [8]. Wassermann and Su [27] and Minamide and Tozawa [20] present string verifiers based on the Minamide's string analysis [19].

Code Generation There are many researches to achieve syntax safety of generated code other than multi-staged languages area.

Repleo [3] shows a template engine which generates syntax safe code. Having the grammar for the template, it statically checks the template and its sub-templates to detect possible syntactic errors in the generated code. However, it still requires evaluating the template with the model to guarantee the generated code is syntax safe.

Engler et al. presents 'C [17], an extension of ANSI C which allows dynamic code generation. They employ a version of quasi-quote system and type annotation to achieve a type safety in the generated code. This setting does not allow the generated code to have a syntactic error.

7 Conclusion

We have presented a static analysis technique for checking the syntax of generated code in two-staged languages with concatenation. Formulating abstract parsing in the abstract interpretation framework, we derive abstract semantics which is composed by atomic parse stack transition functions defined in LR(k) parser. This formulation not only gives us a more concise and elegant explanation of the original idea but also decouples the core idea of abstract parsing from its implementation. The provided framework allows us to choose the abstract domain of abstract parsing and control the precision and cost of analysis.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] D. Ancona and E. Moggi. A fresh calculus for name management. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, October 2004.

[3] Jeroen Arnoldus, Jeanot Bijpost, and Mark van den Brand. Repleo: a syntax-safe template engine. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 25–32, New York, NY, USA, 2007. ACM.

[4] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(3), 2003.

[5] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. ML-like inference for classifiers. In *Proceedings of the European Symposium on Programming*, pages 79–93. Springer, 2004.

[6] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *ACM International Conference on Functional Programming*, pages 275–286. ACM, August 2003.

[7] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A practical string analyzer by the widening approach. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, volume 4729 of *Lecture Notes in Computer Science*, pages 374–388, Sydney, Austrailia, November 2006. Springer-Verlag.

[8] Aske Simon Christensen, Anders Mller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the Static Analysis Symposium*, pages 1–18. Springer-Verlag, 2003.

[9] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. String analysis for x86 binaries. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 88–95, New York, NY, USA, 2005. ACM.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[12] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[13] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of the Symposium on Logic in Computer Science*, pages 184–195. IEEE Computer Society Press, 1996.

[14] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 258–270. ACM, 1996.

[15] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[16] Kyung-Goo Doh, Hyunha Kim, and David Schmidt. Abstract parsing: static analysis of dynamically generated string output using lr-parsing technology. In *Proceeeding of the International Static Analysis Symposium*, 2009. Available from `http://santos.cis.ksu.edu/schmidt/dohsas09.pdf`.

[17] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. C: a language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, New York, NY, USA, 1996. ACM.

[18] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 257–269, 2006.

[19] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International Conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005. ACM.

[20] Yasuhiko Minamide and Akihiko Tozawa. Xml validation for context-free grammars. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, pages 357–373. Springer-Verlag, 2006.

[21] Aleksandar Nanevski. Meta-programming with names and necessity. In *ACM International Conference on Functional Programming*, pages 206–217. ACM, October 2002.

[22] Aleksandar Nanevski and Frank Pfenning. Staged computation with names and necessity. *Journal of Functional Programming*, 15(6):893–939, 2005.

[23] Morten Rhiger. First-class open and closed code fragments. In *Proceedings of the Symposium on Trends in Functional Programming*, September 2005.

[24] Tim Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.

[25] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2003.

[26] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. *ACM Transactions on Software Engineering and Methodology*, 16(4):14, 2007.

[27] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, New York, NY, USA, 2007. ACM.