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.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords Multi-staged languages, Program analysis, Abstract Interpretation, Parsing

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 syntactically correct.

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.

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.

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.

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 \( goto(s_0, [) = s_1 \). 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 \( S \rightarrow a \) and moves the parse state back to \( s_1 \). Then \( goto(s_1, S) = s_3 \) yields

\[
\begin{align*}
x &= 'a' & X_0 &= a \\
1 &= '[' & L &= [ \\
x &= ']' & R &= ] \\
x &= 1 . x & X_1 &= L.X_0
\end{align*}
\]

Figure 1. Example program (left) and its data-flow equations (right)
Figure 2. Goto controller of the LR(0) parser for \( S \rightarrow a \mid [S] \).
(from [16])

\[
X_0(s_1) = s_3, \text{ Therefore we have } X_1(s_0) = s_3 s_1
\]

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(1) parsing. Section 4 presents abstract parsing semantics with LR(0) parsing. 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 \( e \):

\[
e \in \text{Exp} ::= x \mid \text{let } x e_1 e_2 \mid \text{or } e_1 e_2 \mid \text{re } x e_1 e_2 e_3 \mid \text{f}
\]

An expression can contain code fragments \( f \):

\[
f \in \text{Frag} ::= x \mid \text{let } | \text{or } | \text{re } | ( ) \mid f_1, f_2 \mid \epsilon
\]

Operational semantics of the target language is shown in Figure 3.

Expression \( \text{or } e_1 e_2 \) is for branches. It could be the value of \( e_1 \) or the value of \( e_2 \). Expression \( \text{re } x e_1 e_2 e_3 \) is for loops. Variable \( x \) has the value of \( e_1 \) as its initial value. Loop body \( e_2 \) is iterated \( \geq 0 \) times.

The result of each iteration \( e_2 \) will be bound to \( x \) in \( e_2 \) for next iteration. Backquote form \( \text{f} \) is for code fragment \( f \). We construct the fragment by using the following tokens: variables, \( \text{let} \), \text{or}, \text{re}, ( ), and ). Compound fragment \( f_1, f_2 \) concatenates two code fragments \( f_1 \) and \( f_2 \). Comma fragment \( \epsilon \) first evaluates \( e \) then substitutes its result code value for itself.
that the meaning of \( f \) and \( e \) is the same as in LISP’s quasiquotation 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.

\[
\text{re } \lambda x (\text{or } . x) \text{ (} \text{or } \lambda x (\text{or } . x) \text{ b) x}
\]

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,

\[
\text{re } \lambda x (\text{or } . x) \text{ (} \text{or } \lambda x (\text{or } . x) \text{ b) x}
\]

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.

\[
\text{Collecting Semantics} \xrightarrow{} \text{Concrete Parsing Semantics} \xrightarrow{} \text{First Step Abstraction Semantics} \xrightarrow{} \text{Parameterized Abstract Parsing Semantics}
\]

#### 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 \( p_{\text{init}} \) and \( p_{\text{acc}} \) are defined. Parsing starts with the initial parse stack \( p_{\text{init}} \). Successful parsing should stop at the accept parse stack \( p_{\text{acc}} \). Otherwise it indicates that the parsed string does not conform to the given grammar. String representation “\( s_{\text{stop}} \ldots s_{\text{bot}} \)” denotes a parse stack whose top state is \( s_{\text{stop}} \) and bottom state is \( s_{\text{bot}} \). 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 \( \text{parse_action} : P \rightarrow \text{Token sequence} \rightarrow P \) which is described in Algorithm 1. It returns the parse stack from the given parse stack \( p \) and input token \( f \).

The parsing process \( \text{parse} : P \rightarrow \text{Token sequence} \rightarrow P \) is a composition of the \( \text{parse_action} \).

\[
\text{parse}(p, t_1 \ldots t_n) = \text{parse_action}(\ldots(\text{parse_action}(p, t_1)), \ldots, t_n)
\]

A parser gets the input code \( c \) and returns the parse stack \( \text{parse}(p_{\text{init}}, c) \).
Algorithm 1 parse_action algorithm
1: procedure parse_action(p, t)
2:   \(s_{top} \rightarrow \) 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 \(\rightarrow\) \(\beta\) then
7:     pop \(\beta\) symbol off the stack p
8:     \(s_{top} \rightarrow\) 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 \(V_P = 2^{P \rightarrow P}\)
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 \(\text{parse}(\text{pout}, c)\) as an abstraction of the code \(c\). However this setting causes a problem when we handle the concatenation \(x.y\). Let \(p_x = \text{parse}(\text{pout}, x)\) and \(p_y = \text{parse}(\text{pout}, y)\) be the resulting parse stacks for the code fragments \(x\) and \(y\). The parse stack \(p\) for the concatenation \(x.y\) is computed as
\[
p = \text{parse}(\text{pout}, x.y) = \text{parse}(\text{parse}(\text{pout}, x), y) = \text{parse}(p_x, y).
\]

However, what we have is \(p_y = \text{parse}(\text{pout}, y)\) not \(\text{parse}(p_x, y)\). The \(\text{parse}(\text{pout}, y)\) 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 \(\lambda p.\text{parse}(p, c)\) 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
\[
\begin{align*}
  f_x &= \lambda p.\text{parse}(p, x) \\
  f_y &= \lambda p.\text{parse}(p, y).
\end{align*}
\]

With the two parse stack transition functions \(f_x\) and \(f_y\), we can construct the parse stack transition function \(f_{x.y}\) as follows.
\[
\begin{align*}
  f_{x.y} &= \lambda p.\text{parse}(p, x.y) \\
         &= \lambda p.\text{parse}(p, x), y) \\
         &= (\lambda p.\text{parse}(p, y)) \circ (\lambda p.\text{parse}(p, x)) \\
         &= f_y \circ f_x.
\end{align*}
\]

3.4 Concrete Parsing Semantics
Using the abstraction from Code to \(P \rightarrow P\), the Galois connection \(2^{\text{Code}} \bowtie_{\alpha} V_P = 2^{P \rightarrow P}\) is established as follows.
\[
\begin{align*}
  \alpha &= \lambda C. \{ \lambda p.\text{parse}(p, c) \mid c \in C\} \\
  \gamma &= \lambda F. \bigcup_{p \in P} \{ c \mid \forall p \in P.\text{parse}(p, c) = f(p)\}.
\end{align*}
\]

Concrete parsing semantics is derived from the collecting semantics as follows.
\[
\begin{align*}
P &= \text{the set of parse stacks} \\
V_P &= 2^{P \rightarrow P} \\
\sigma &\in \text{Env}_P \rightarrow V_P \\
\left[ e \right]_P^0 &\in \text{Env}_P \rightarrow V_P.
\end{align*}
\]

4. Abstract Parsing
4.1 First Step Abstraction : Value \(V_P = 2^{P \rightarrow P}\)
First, we abstract the concrete parsing domain \(V_P = 2^{P \rightarrow P}\) to \(V_{P} = 2^{P} \rightarrow 2^{P}\) by establishing the Galois connection \(V_P \rightarrow V_{P}\) where
\[
\alpha = \lambda F.\lambda P. \bigcup_{p \in P} \{ f(p) \mid f \in F\}
\]
\[
\gamma = \lambda F. \bigcup_{\sigma \in V} \{ S \mid \alpha(S) \subseteq \{ f \} \}.
\]

Then we derive the abstract semantics for this domain as follows.
\[
\begin{align*}
\sigma &\in \text{Env}_P \rightarrow V_P \\
\left[ e \right]_P^0 &\in \text{Env}_P \rightarrow V_P \\
\left[ f \right]_P^1 &\in \text{Env}_P \rightarrow V_P \\
\left[ x \right]_P^0 &= \sigma(x) \\
\left[ \text{let } x \ e_1 \ e_2 \right]_P^0 \sigma &= \left[ e_2 \right]_P^0 \sigma \cup \left[ e_1 \right]_P^0 \sigma \\
\left[ \text{or } e_1 \ e_2 \right]_P^0 \sigma &= \left[ e_2 \right]_P^0 \sigma \cup \left[ e_1 \right]_P^0 \sigma \\
\left[ \text{re } x \ e_1 \ e_2 \right]_P^0 \sigma &= \left[ e_2 \right]_P^0 \sigma \cup \left[ e_1 \right]_P^0 \sigma \\
\left[ f \right]_P^1 &= \left[ f \right]_P^0 \sigma \\
\left[ \text{let } x \ e_1 \ e_2 \right]_P^0 \sigma &= \left[ e_2 \right]_P^0 \sigma \cup \left[ e_1 \right]_P^0 \sigma \\
\left[ f \right]_P^1 &= \left[ f \right]_P^0 \sigma \\
\left[ \text{let } x \ e_1 \ e_2 \right]_P^0 \sigma &= \left[ e_2 \right]_P^0 \sigma \cup \left[ e_1 \right]_P^0 \sigma \\
\left[ f \right]_P^1 &= \left[ f \right]_P^0 \sigma
\end{align*}
\]

where \(\text{Parse}_\text{action} : \ 2^P \rightarrow \text{Token} \rightarrow 2^P\) is the natural set extension of \(\text{parse}_\text{action}\):
\[
\text{Parse}_\text{action} = \lambda P.\lambda t. \{ \text{parse}_\text{action}(p, t) \mid p \in P\}.
\]

The abstract semantic function \([ f ]_P^1\) is used to check whether generated code conforms to the grammar. For the given program \(e\), we compute
\[
S = \left[ e \right]_P^0 \sigma_0 \{ \text{pout} \} : 2^P
\]
where \(\sigma_0 \in \text{Env}_P\) is an empty environment. Then we compare \(S\) with \(\{ \text{puse} \}\), the accept parse state. 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). $V^t = D^t \rightarrow D^t$ is an abstract parsing domain if an abstract domain $D^t$ satisfies the following conditions [10, 11, 12].

1. $(D^t, \subseteq, \sqcup, \perp_{D^t})$ is a CPO (Complete Partial Order).
2. Powerset domain of concrete parse stack $2^P$ and its abstract domain $D^t$ are Galois connected via $\alpha_{2^P \rightarrow D^t}$ and $\gamma_{D^t \rightarrow 2^P}$.
3. Parse-action$^2 : D^t \rightarrow Token \rightarrow D^t$ is a sound approximation of Parse-action. That is, we have

   $$\forall P \in 2^P, \forall t \in Token.$$

   $$\alpha_{2^P \rightarrow D^t}(\text{Parse-action}(P, t)) \subseteq \text{Parse-action}^2(\alpha_{2^P \rightarrow D^t}(P), t)$$

   The partial order $\subseteq_{V^t}$ and join operator $\sqcup_{V^t}$ are defined point-wisely.

**Definition 2.** Semantic function $\llbracket \cdot \rrbracket_{D^t}$ on the abstract parsing domain $V^t$ is defined as follows:

- $\sigma \in Env_{D^t}$ is $Var \rightarrow V^t$
- $\llbracket e \rrbracket_{D^t} \in Env_{D^t} \rightarrow V^t$
- $\llbracket f \rrbracket_{D^t} \in Env_{D^t} \rightarrow V^t$

- $\llbracket x \rrbracket_{D^t} = \sigma(x)$
- $\llbracket e_1 \ u_1 \ e_2 \ u_2 \ \ldots \ u_{n-1} \ e_n \rrbracket_{D^t} = \llbracket e_1 \rrbracket_{D^t} \sqcup \cdots \sqcup \llbracket e_n \rrbracket_{D^t}$

Theorem 1 shows that $\llbracket P \rrbracket_{D^t}$ is a sound approximation of $\llbracket P \rrbracket_\rho$.

**Theorem 1.** Semantic function $\llbracket \cdot \rrbracket_{D^t}$ on the abstract parsing domain $V^t$ is a sound approximation of $\llbracket \cdot \rrbracket_\rho$. That is, we have

$$\forall e \in \text{Expr}_\rho, \forall \rho \in Env_\rho, \alpha_{V^t \rightarrow V^t} \left( \llbracket e \rrbracket_\rho \right) \subseteq \llbracket e \rrbracket_{D^t} \left( \alpha_{Env_\rho \rightarrow Env_{D^t}}(\sigma) \right)$$

where

$$\alpha_{V^t \rightarrow V^t} = \lambda f. \lambda D. \alpha_{2^P \rightarrow D^t}(\gamma_{D^t \rightarrow 2^P}(D))$$

and

$$\alpha_{Env_\rho \rightarrow Env_{D^t}} = \lambda \rho. \lambda D. \alpha_{2^P \rightarrow D^t}(\rho)$$

**Proof.** By structural induction on $e$ with the conditions that the abstract parsing domain $D^t$ 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 $D^t$, 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

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

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 $\text{Parse-action}$, we show that $\hat{D}$ is an instance of $D^t$. 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 $\hat{P}$ is introduced to represent a parse stack which has been cut and only maintains top $n$ states. Special state '\-' at the bottom of cut parse stack $\hat{P}$ indicates that it has been cut.

$$\hat{P} = \{ p \in \Sigma^* \}$$

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

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

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

$$\text{prefix}(\rho) = \begin{cases} \rho & \text{if } \rho \in P \\ s_1 \ldots s_{n-1} & \text{if } \rho = s_1 \ldots s_{n-1} \\ \epsilon \text{ (empty string)} & \rho = - \end{cases}$$

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

$$\rho_1 \subseteq_{\hat{P}} \rho_2 \overset{def}{=} \text{prefix}(\rho_1) \text{ starts with prefix}(\rho_2) \land \rho_2 \in \hat{P}$$

#### 5.2 Abstract Domain : $\hat{D}$

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

$$\hat{D} = \{ \text{norm}(d) \mid d \in 2^P \}$$

$$\hat{d}_1 \sqsubseteq \hat{d}_2 \overset{def}{=} \forall \rho_1 \in \hat{d}_1, \exists \rho_2 \in \hat{d}_2, \rho_1 \sqsubseteq \rho_2$$

$$\hat{d}_1 \sqcup \hat{d}_2 \overset{def}{=} \text{norm}(\hat{d}_1 \cup \hat{d}_2)$$

Normalization function $\text{norm}$ is defined by

$$\text{norm}(\hat{d}) = \{ \rho \in \hat{d} \mid \forall \rho' \in \hat{d}, \rho \not\sqsubseteq \rho' \}$$

to ensure that elements in $\text{norm}(\hat{d})$ are the maximal elements of $\hat{d}$. It is necessary to eliminate non-maximal elements for binary relation $\subseteq$ 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 $s_1 s_0 - \subseteq -$. If we allow $\{ s_1 s_0 -, - \}$ in $\hat{D}$ without normalizing it to $\{ -, - \}$, we have

$$\{ s_1 s_0 -, - \} \wedge (-) \subseteq \{ s_1 s_0 -, - \} \not\in \{ s_1 s_0 -, - \}$$

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

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

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

1. $(\hat{D}, \subseteq, \sqcup, \phi)$ 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 $\text{expand} : P \rightarrow 2^P$ as follows.

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

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

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

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

From the property of $\text{expand}$, it is clear that $\text{Expand}(\hat{d}) \subseteq_{D} \hat{d}$. Using the $\text{Expand}$ function, we define the Galois connection $2^P \rightarrow_{\alpha} D$ where

$$\alpha = \text{id}$$
$$\gamma = \lambda \hat{d}. \text{Expand}(\hat{d})$$

**Proof.** To prove $\alpha$ and $\gamma$ constitute a Galois connection, it is sufficient to show that the following properties hold.

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

(b) Monotonicity of $\gamma$ is immediate from the monotonicity of $\text{expand}$.

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

$$\alpha \circ \gamma(\hat{d}) = \gamma(\hat{d}) = \text{Expand}(\hat{d}) \subseteq \hat{d}$$

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

$$\gamma \circ \alpha(P) = \gamma(P) = \text{Expand}(P) = \bigcup_{\rho \in P} \text{expand}(\rho) = P.$$  

\( \square \)

3. We first define $\text{parse} \_\text{action} : \hat{P} \rightarrow \text{Token} \rightarrow \hat{P}$ as in Algorithm 2. It is a modified version of the $\text{parse} \_\text{action}$ function. The only modifications are adding lines 2 – 4 and lines 12 – 14 to handle $\hat{p} \in \hat{P}$ because the action and goto tables do not have an entry for the special state "-". Note that $\forall P \in P, \forall t \in \text{Token}. \text{parse} \_\text{action}(p, t) = \text{parse} \_\text{action}(p, t)$.

$\text{Parse} \_\text{action} : \hat{D} \rightarrow \text{Token} \rightarrow \hat{D}$ is defined as the natural set extension of $\text{parse} \_\text{action}$ as follows.

$\text{Parse} \_\text{action} = \lambda \hat{d}. \lambda t. \{ \text{parse} \_\text{action}(p, t) | \rho \in \hat{d} \}$

$\text{Parse} \_\text{action}$ is a sound approximation of $\text{Parse} \_\text{action}$.

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

$$\alpha_{2^P \rightarrow \hat{D}}(\text{Parse} \_\text{action}(P, t)) = \text{Parse} \_\text{action}(P, t)$$
$$= \{ \text{parse} \_\text{action}(p, t) | p \in P \}$$
$$= \text{parse} \_\text{action}(P, t)$$
$$= \text{Parse} \_\text{action}(P, t)$$
$$= \text{Parse} \_\text{action}(\alpha_{2^P \rightarrow \hat{D}}(P), t).$$  

\( \square \)

5.4 Widening Operator on $D$

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 $\bigtriangledown_D : \hat{D} \times \hat{D} \rightarrow \hat{D}$ such that

$$A \bigtriangledown_D B = \{ \text{norm}(\text{cut}_k(\rho)) | \rho \in A \cup B \}$$

where $\text{cut}_k$ is defined by

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

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

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

**Proof.** We have to check if $\bigtriangledown_D$ operator satisfies the widening conditions [10, 11, 12]:

(i) $\forall x, y \in \hat{D}. (x \subseteq x \bigtriangledown_D y) \land (y \subseteq x \bigtriangledown_D y)$.

(ii) for all increasing chains $x_0 \subseteq x_1 \ldots$, the increasing chain defined by $y_0 = x_0, y_1 = x_1, \ldots$ is not strictly increasing.

To prove (i), we observe that $\rho \subseteq \rho, \text{cut}_k(\rho)$ by definition of $\subseteq \rho$ and $\text{cut}_k$. Using this we get

$$x = \{ \rho | \rho \in x \} \subseteq \text{cut}_k(\rho)$$
$$\subseteq \{ \text{cut}_k(\rho) | p \in x \}$$
$$= x \bigtriangledown_D y.$$  

Proof for $y \subseteq x \bigtriangledown_D y$ is analogous.

To prove (ii), we observe that the range of $\bigtriangledown_D$ operator is the finite set $\rho' = \{ \rho \in P | |\rho| \leq k \}$. For all $i \geq 1$, we have $y_i \in P'$. Therefore the increasing chain $y_0, y_1, \ldots$ is not strictly increasing.  

\( \square \)

Using the widening operator $\bigtriangledown_D$, we define a widening operator for $\forall = \hat{D} \rightarrow \hat{D}$, such that

$$f \bigtriangledown_D g = \lambda \hat{d}. \begin{cases} f(\hat{d}) \bigtriangledown_D g(\hat{d}) & \text{if } \forall \rho \in \hat{d}, |\rho| \leq l \\ \{-\} & \text{otherwise}. \end{cases}$$
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 $\nabla_{\mathcal{D}}$. This $\nabla_{\mathcal{D}}$ 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, $\mathcal{D} = \{d \in \mathcal{D} | |d| \leq k\}$. 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.

```plaintext
(\texttt{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 $\mathcal{D}$ 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. However, 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.

Cho 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.

Acknowledgments

We thank Kyung-Goo Doh, Hyunha Kim, and David Schmidt for their willingness to let us build upon their unpublished work. We thank Yungbum Jung, Hakjoo Oh, Deokhwan Kim, Will Klieber, and the anonymous referees for their helpful comments and suggestions.

References


