String Analysis as an Abstract Interpretation

Se-Won Kim and Kwang-Moo Choe

Korea Advanced Institute of Science and Technology

Abstract. We formalize a string analysis within abstract interpretation framework. The abstraction of strings is given as a conjunction of predicates that describes the common configuration changes on the reference pushdown automaton while processing the strings. We also present a family of pushdown automata called ϵ bounded pushdown automata. This family covers all context-free languages, and by using this family of pushdown automata, we can prevent abstract values from becoming infinite conjunctions and guarantee that the operations required in the analyzer are computable.

1 Introduction

The baseline of the correctness of a document-generating program is that all documents generated by the program should be syntactically correct. The syntax is usually defined by a context-free grammar or equivalently a pushdown automaton, which we call the *reference* of the syntax.

The current verification techniques for this property use resolution of constraints, which over-approximate the output of the program. This approach was first introduced in [5], and was adopted in [9,21,15] and other semantic analyses based on them. Though this approach has been quite useful in many applications, it has some drawbacks.

– It is hard to recover the precision lost during the constraint generation. In [9], the ignored branch conditions become the main cause of false alarms.
– Integration with other analysis is not easy because they use constraints as the target of verification.
– The constraints usually trace the flow of strings only for the shallow variables like the local and global variables[4]. If the program heavily involves heap or structured data, they often fail to capture the precise flow within the constraints.
– It may require manually provided *hot-spots*. Each hot-spot is composed of one program point and a variable whose possible string values should be verified.
– It rejects the constraints that generate valid substrings of the reference, and implicitly enforces whole and complete program for the verification.

We alleviate these problems by designing a string analysis within abstract interpretation framework[6,7,8], and let it benefit from various abstract interpretation techniques. The contributions of this paper are:

- We suggest a novel abstraction of strings. These abstract values express the key characteristics of strings that compose the output documents.
- Our string analysis can be used for all context-free references.
- Practically, our abstract domain and abstract operations can be plugged into a conventional analysis framework. This makes it easy to compose string analysis with other analysis and to benefit from the various techniques devoloped for abstract interpretation.

We outline the remainder of this paper. We first define notations and terminologies and see one motivating example that cannot be verified by the previous constraint generation approach. After that, we thoroughly study the meaning of one string from the view of the reference automaton and apply this knowledge to the design of a new abstract domain for strings. We compare our work with previous approaches and conclude.

2 Preliminary

We assume the reference is given as a pushdown automaton (henceforth PDA).

Definition 1 (PDA). *A PDA* $\mathbf{A}$ *is a tuple defined as* $\mathbf{A} = (Q, \Sigma, \Gamma, \Delta, q_0, Z_0, Q_F)$

- Q *is the finite set of states.*
- Σ *is the finite set of input symbols.*
- Γ *is the finite set of stack symbols.*
- $\Delta \subseteq Q \times \Gamma^* \times (\Sigma \cup \{\epsilon\}) \times Q \times \Gamma^*$ *is the finite set of actions. An action* $(q_1, \gamma_1, a, q_2, \gamma_2) \in \Delta$ *is written as* $q_1\gamma_1 \overset{a}{\mapsto} q_2\gamma_2$.
- $q_0 \in Q$ *is the initial state.*
- $Z_0 \in \Gamma$ *is the initial stack content.*
- $Q_F \subseteq Q$ *is the set of final states.*

A configuration of PDA consists of a state and stack content. If the current state is q and the stack content is $Z_1, Z_2 \ldots, Z_n$ from the top, we denote it as $qZ_1Z_2 \cdots Z_n$. Reading a from the input, the PDA configuration changes from $q_1\gamma_1\gamma$ to $q_2\gamma_2\gamma$ if $q_1\gamma_1 \overset{a}{\mapsto} q_2\gamma_2 \in \Delta$. This PDA transition is written as $q_1\gamma_1\gamma \overset{a}{\mapsto} q_2\gamma_2\gamma$. If the PDA configuration changes from $q_1\gamma_1$ to $q_2\gamma_2$ while reading $w \in \Sigma^*$, we denote it as $q_1\gamma_1 \overset{w}{\mapsto} q_2\gamma_2$. If PDA configuration can be changed from $q_1\gamma_1$ to $q_2\gamma_2$, we denote it as $q_1\gamma_1 \overset{*}{\mapsto} q_2\gamma_2$. The language that the automaton $\mathbf{A}$ accepts is defined as

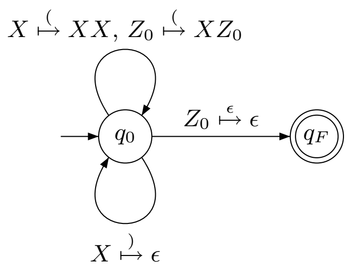
$$L(\mathbf{A}) = \{w \in \Sigma^* \mid q_0Z_0 \overset{w}{\mapsto} q_F,\ q_F \in Q_F\}.$$

We denote state by p and q, stack symbol by X, Y, and Z, input by w. Note that we can transform this stateful PDA to a stateless PDA if we remove the set Q from the tuple and let the set of stack symbol be the disjoint union of Q and Γ, and ensure the state is always at the stack top. In this vein, we use η, γ, δ and κ to represent a PDA configuration or a part of it, which may contain state symbol at the first position. For a configuration $q\gamma$, if $q\gamma = \gamma_1\gamma_2$, we say γ_1 (resp. γ_2) is a *prefix* (resp. *suffix*) of configuration $q\gamma$. We resort to the stateful PDA definitions for three reasons.

– The stateful PDAs are practically more convenient for describing PDAs.
– The distinguished first symbols of configurations help us identify the start of a configuration in a graph structure representing set of configurations. This graph structure is introduced in Section 4.
– The state can be an extra space for integration of other facilities. For example, input preprocessors based on finite state transducers can be integrated into a stateless parser as additional state actions.

3 Motivating Example

In this section, we demonstrate the weakness of constraint generation approach by one simple scenario. Let us assume that we are developing a program that should generate document described by the PDA in Fig. 1a. In the middle of the development, we have the program[1] in Fig. 1b, which constructs valid substring of the document.


$X \overset{(}{\mapsto} XX$, $Z_0 \overset{(}{\mapsto} XZ_0$
q_0
$Z_0 \overset{\epsilon}{\mapsto} \epsilon$
q_F
$X \overset{)}{\mapsto} \epsilon$

(a) A PDA for matching parentheses. State transition is represented by the start and end node of the edge, and stack behavior and input symbol is shown on edges. q_F is the final state.

```
1  $x$ := ‘’;
2  $n$ := (some positive integer);
3  $i$ := 0;
4  while $i < n$ do
5      if $i = 0$ then
6          $x$ := ‘((’
7      else
8          $x$ := $x$ + ‘)(’
9      end
10     $i$ := $i$ + 1;
11 end
12 $x$ := $x$ + ‘)))’;
```

(b) A program in the middle of development. It generates a valid substring of matching parentheses.

Fig. 1. A reference PDA and an incomplete target program

When we apply constraint generation approach in [5] to verify this program, we get the following constraints that track the flow of string values.

$$X_1 \to \epsilon \qquad X_4 \to X_1 \mid X_9 \qquad X_6 \to ((\qquad X_8 \to X_4)($$
$$X_9 \to X_6 \mid X_8 \qquad X_{11} \to X_1 \mid X_9 \qquad X_{12} \to X_{11})))$$

Each X_i represents the possible set of strings that the variable x can have after executing the ith line. Since the program constructs only part of its output

[1] The loop sturucture of this snippet is from a web application written in PHP, School-Mate v1.5.4.

document, the resolution based on the language inclusion within the reference simply fails. However, we assume that the programmer supplies the left and right context of X_{12} by manually adding the rule $S \rightarrow (X_{12}$ to the constraints and uses S as the hot-spot for the verification.

Nevertheless, the constraints suggest that the program might generate '()))' by the derivation $S \Rightarrow (X_{12} \Rightarrow (X_{11}))) \Rightarrow (X_1))) \Rightarrow ()))$, which leads to a false alarm. This false alarm is caused by ignoring the value of n and the control flow that depends on it. There can be two ways to remove this false alarm:

- Run the sign and flow analysis before constraint generator and use the analysis result to transform the program before the constraint generation, or let the constraint generator take into account the analysis result during the constraint generation.
- Run the string analysis simultaneously with sign and flow analysis in one framework.

In this paper, we develop the theoretical facilities for the second approach. The second approach is conceptually simpler in that it requires only abstract domain and abstract operations for strings, and has strictly more opportunity to correctly verify programs. It can also support modular string analysis: we can verify incomplete programs and help find bugs early during the development.

4 PDA Configuration Changes Caused by One String

In this section, we study configuration changes caused on the reference PDA while reading a string. We devise an algorithm which builds a finite representation of all possible PDA configuration changes caused by the given string. We also study a condition for PDAs which is closely related to the complexity of the representation. Throughout this section, we assume the reference PDA is given as $\mathbf{A} = (Q, \Sigma, \Gamma, \Delta, q_0, Z_0, Q_F)$.

4.1 The Need

If the output document is to be correctly processed by the reference PDA, each fragment, which we assume is one of the string literals in the program, successively transforms the PDA configuration from the initial configuration to the accepting. The first fragment transforms the initial configuration to another, and the second fragment transforms the ending configuration of the first fragment to another. This process continues to the last fragment, which must transform the configuration to the accepting at the end. This observation leads us to consider the PDA configuration changes caused by string literals in the program. The same idea was used in [17] for deterministic $LR(k)$ languages. They regard parse actions as *black box* mappings from one parse stack to another. On the other hand, we look into the detailed inner workings of configuration transitions on a string in more general non-deterministic PDAs.

4.2 Configuration Change

For each string literal, we only need to know all pairs of the initial configuration before processing the string and the last configuration after processing it.

Definition 2 (configuration change). *If for a string $w \in \Sigma^*$, $q_0 Z_0 \overset{*}{\mapsto} \delta\eta \overset{w}{\mapsto} \delta'\eta$ and the suffix η is intact (never popped nor read) while processing w, we say w has a configuration change $\delta \rightsquigarrow \delta'$.*

We deliberately omitted the suffix η for the configuration change. It is natural since the bottom η is not relevant as a change caused by the given string. Note that the most conservative choice for η is ϵ. If the bottom η is chosen aggressively to be maximal, we say the resulting configuration change is *minimal*.

4.3 Algorithm for Computing Minimal Configuration Changes

We present the algorithm for finding all minimal configuration changes. This is enough since each configuration change can be obtained by appending valid fragment η to one of the minimal configuration changes. We assume that the input string is $w = a_1 a_2 \cdots a_n$, and n denotes the length of the input.

The algorithm has two phases:

1. It computes all reachable configurations before processing w. The configurations are represented as a finite graph(actually, a relaion '$\succ$' on a finite set $\mathcal{I}$), which we refer as *the initial closure*. The closure represents reachable configurations after processing possible left context of w.
2. Based on the initial closure, it extends the graph while reading the actual input. A valid path in the graph extension corresponds to δ' of a minimal configuration change $\delta \rightsquigarrow \delta'$ of w. It also simultaneously constructs a grammar whose sentence corresponds to δ of $\delta \rightsquigarrow \delta'$.

Our algorithm is based on the algorithm of [18], which applies tabular parsing to pushdown transducers. This technique also has been applied to parsing sentence with wild cards[19], parsing sentence of ambiguous grammar[3] and linear-time suffix parsing of deterministic pushdown languages[23]. Our algorithm constructs the relation '$\succ$' and a context-free grammar. The relation represents a set of reachable configurations, which is similar to [18,23,19,3]. The difference is on the constructed grammar. The constructed grammar in [18,23,19,3] represents the output of pushdown transducers, that is, all the possible parse of the given string. On the other hand, the constructed grammar of our algorithm represents the prefixes of configurations which had been read from the initial closure while processing the input.

The Relation '$\succ$'. The binary relation '$\succ$' is on the finite set of *items*,

$$\begin{aligned} \mathcal{I} = &\{[q\gamma]_* \mid q\gamma \text{ is a prefix of } q_0 Z_0 \dashv\} \\ &\cup \{[q\gamma]_i \mid i = * \vee (i \in \mathbb{Z} \wedge 0 \le i \le n),\ \exists \gamma_1 \overset{a}{\mapsto} \gamma_2 \in \Delta.\ q\gamma \text{ is a prefix of } \gamma_2\}, \end{aligned}$$

Initial relation: $[q_0]_* \succ [q_0 Z_0]_* \succ [q_0 Z_0 \dashv]_*$.

If

$$pX_1 \cdots X_m \overset{a}{\mapsto} qY_1 \cdots Y_l \in \Delta$$

$$[p]_* \succ [\gamma_1 X_1]_* \succ \cdots \succ [\gamma_m X_m]_* \succ [\gamma]_*$$

then

$$[q]_* \succ [qY_1]_* \succ \cdots \succ [qY_1 \cdots Y_l]_* \succ [\gamma]_*$$

Fig. 2. The condition for the initial closure

where the fresh stack symbol '$\dashv$' denotes the end of configurations. Intuitively, each $[q\gamma X]_i$ (resp. $[q]_i$) represents one stack symbol X(resp. a state symbol q) and the relation '$\succ$' represents 'is on top of' relation between symbols. Let us call $[\gamma_1]_{i_1} \succ [\gamma_2]_{i_2}$ an *edge*. The relation as a whole represents a set of configurations. The non-negative integer index i of an item represents the length of the input read when the relation is extended. If the edge is constructed as the initial closure, where the arbitrary input is used, we use '$*$' for the index.

For convenience, we abbreviate $[\gamma_1]_{i_1} \succ [\gamma_2]_{i_2}$, $[\gamma_2]_{i_2} \succ [\gamma_3]_{i_3}$, …, $[\gamma_{k-1}]_{i_{k-1}} \succ [\gamma_k]_{i_k}$ as one chain: $[\gamma_1]_{i_1} \succ [\gamma_2]_{i_2} \succ \cdots \succ [\gamma_k]_{i_k}$.

The Initial Closure. The *initial closure* is the minimal relation that satisfies the condition in Fig. 2. Its construction starts with the relation that represents the initial configuration, and extends the relation by simulating the PDA actions on the relation. Note that PDA actions on the relation are *additive*: they do not destruct edges for popping symbols.

The initial closure describes all possible PDA configurations after reading some input.

Theorem 1. $\exists \gamma_1, \ldots, \gamma_m.\ [p]_* \succ [\gamma_1 Z_1]_* \succ \cdots \succ [\gamma_m Z_m]_* \succ [q_0 Z_0 \dashv]_*$, *if and only if* $q_0 Z_0 \overset{*}{\mapsto} pZ_1 \cdots Z_m$.

The initial closure can also be used for checking the validity of a prefix.

Corollary 1. $\exists \gamma_1, \ldots, \gamma_m, \gamma, Z.\ [p]_* \succ [\gamma_1 Z_1]_* \succ \cdots \succ [\gamma_m Z_m]_* \succ [\gamma Z]_*$, *if and only if* $\exists \eta.\ q_0 Z_0 \overset{*}{\mapsto} pZ_1 \cdots Z_m \eta$.

All the proof of the propositions in this paper can be found in [14].

Processing the Actual Input. The minimal configuration changes of the given input w can be obtained from the smallest extension of the initial closure and the smallest set of production rules that satisfy the rule in Fig. 3. Their construction starts with the initial closure and empty set of production rules, and repeats the rule in Fig. 3 until there can be no additional edges and production rules.

If

$$pX_1 \cdots X_m \overset{a}{\mapsto} qY_1 \cdots Y_l \in \Delta$$

$$[p]_{i_0} \succ [\gamma_1 X_1]_{i_1} \succ \cdots \succ [\gamma_m X_m]_{i_m} \succ [\gamma]_{i_{m+1}}$$

and if

1. $i_0 = * \wedge a \in \{\epsilon, a_1\}$, then $k = |a|$ and
   - Let $\Upsilon = ([qY_1 \cdots Y_l]_k \succ [\gamma]_{i_{m+1}})$
   - Add $\Upsilon \rightarrow pX_1 \cdots X_m$
2. $i_j \in \mathbb{Z} \wedge i_{j+1} = * \wedge a \in \{\epsilon, a_{i_0+1}\}$, then $k = i_0 + |a|$ and
   - Let $\Upsilon = ([qY_1 \cdots Y_l]_k \succ [\gamma]_{i_{m+1}})$
   - Two cases:
     - If $j = m$, then add $\Upsilon \rightarrow ([\gamma_m X_m]_{i_m} \succ [\gamma]_{i_{m+1}})$
     - If $j < m$, then add $\Upsilon \rightarrow ([\gamma_j X_j]_{i_j} \succ [\gamma_{j+1} X_{j+1}]_{i_{j+1}}) X_{j+1} \cdots X_m$
3. $i_{m+1} \in \mathbb{Z} \wedge a \in \{\epsilon, a_{i_0+1}\}$, then $k = i_0 + |a|$.

If one of the above three conditions is satisfied,

$$[q]_k \succ [qY_1]_k \succ \cdots \succ [qY_1 \cdots Y_l]_k \succ [\gamma]_{i_{m+1}}$$

Fig. 3. Processing the actual input $w = a_1 a_2 \cdots a_n$. The construction starts with the initial closure.

Essentially, the rule in Fig. 3 simulates PDA actions on the relation as the rule in Fig. 2. The increased complexity comes from its additional workload:

- It reads input symbols from left to right and uses the variable k to denote the length of the input read. It uses k for the index of the items that represent the symbols pushed by the current action.
- It transcribes the *deficiency* of configuration into a production rule. The deficiency occurs when we directly or indirectly use the edges in the initial closure for popping symbols.

Before the details of Fig. 3, we show some properties first, which will help us understand the algorithm.

Lemma 1. *If* $[\gamma_1]_{i_1} \succ [\gamma_2]_{i_2}$ *then,* $(i_1, i_2 \in \mathbb{Z} \wedge i_1 \geq i_2)$ *or* $(i_1 \in \mathbb{Z} \wedge i_2 = *)$ *or* $(i_1 = i_2 = *)$.

The constructed context-free grammar uses $Q \cup \Gamma$ as the set of terminal, and $([\gamma_1]_{i_1} \succ [\gamma_2]_{i_2})$'s for non-terminals, whose language represents the symbols read from the initial closure to construct the edge. From the algorithm, we can verify that the referred non-terminals have more restricted form.

Lemma 2. *If* $([\gamma_1]_{i_1} \succ [\gamma_2]_{i_2})$ *appears in a production rule added by the rule in Fig. 3, then* $i_1 \in \mathbb{Z}$ *and* $i_2 = *$.

In other words, only the edges that cross the boundary of the initial closure are required for the grammar.

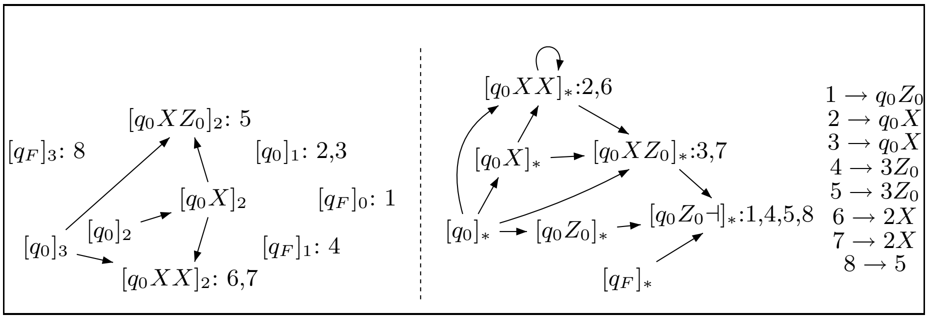


Fig. 4. The result of computing minimal configuration changes on the input ')()' with the PDA in Fig. 1a

Understanding the Rule in Fig. 3. When the algorithm discovers an action,

– whose pop symbols match a sequence of items in '$\succ$',
– and whose input symbol a is at the current position of the input,

it adds the sequence of edges for the push. The indexes of the pushed items are the length of the input read so far. The reasoning behind the production rule construction for each cases are:

1. Case $i_0 = *$: By Lemma 1, $i_0 = i_1 = \cdots = i_{m+1} = *$. In this case, all pop symbols are from the initial closure. We transcribe the deficient fragment $pX_1 \cdots X_m$ from the sequence of the popped items, and remember this fragment as a possible deficiency required for constructing the edge $[qY_1 \cdots Y_l]_k \succ [\gamma]_*$.
2. Case $i_j \in \mathbb{Z} \wedge i_{j+1} = *$: By Lemma 1, $i_0, \ldots, i_j \in \mathbb{Z} \wedge i_{j+1} = \cdots = i_{m+1} = *$.
   – Case $j = m$: No pop symbols are directly from the initial closure. However, the last edge is crossing the boundary of the initial closure, and we are using this edge for popping X_m. In this case, we have only the indirect deficiency for using the last edge, and need to refer it as a possible deficiency required for constructing the edge $[qY_1 \cdots Y_l]_k \succ [\gamma]_*$.
   – Case $j < m$: In this case, we have both kinds of deficiency. We need to append the indirect deficiency, which is referred by the non-terminal, and the direct deficiency transcribed from the items of the initial closure.
3. Case $i_{m+1} \in \mathbb{Z}$: By Lemma 1, $i_0, \ldots, i_{m+1} \in \mathbb{Z}$. Since there is neither direct nor indirect deficiency, it simply add no production rule.

In Fig. 4, we show the result for the input ')()' using the PDA in Fig. 1a as the reference. The relation '$\succ$' is represented as arrows. The initial closure lies on the right of the dotted line, and the edges from processing the actual input lies on the left of the dotted line. For conciseness, those edges that should have crossed the dotted line are represented as the additional number labels on items. For example, the label 1 on $[q_F]_0$ and $[q_0 Z_0 \dashv]_*$ represents the edge from $[q_F]_0$ to $[q_0 Z_0 \dashv]_*$. These number labels are also used in the constucted grammar, which is on the right of the initial closure.

Interpretation of the Result

Theorem 2. *For the input w, there exists $[q]_{i_0} \succ [\gamma_1 Z_1]_{i_1} \succ \cdots \succ [\gamma_m Z_m]_{i_m} \succ [\gamma Z]_*$ where $i_0 = |w| \wedge i_m \in \mathbb{Z}$ and $\eta \in L([\gamma_m Z_m]_{i_m} \succ [\gamma Z]_*)^2$, if and only if w has the minimal configuration change $\eta \rightsquigarrow q Z_1 \cdots Z_m$.*

By Theorem 2, the minimal configuration changes from Fig. 4 are $q_0 X Z_0 \rightsquigarrow q_0 Z_0$, $q_0 X X \rightsquigarrow q_0 X$, and $q_0 X Z_0 \rightsquigarrow q_F$.

When the input is ϵ, there is one additional minimal configuration change $\epsilon \rightsquigarrow \epsilon$ besides the configuration changes computed from Fig. 3. This trivial configuration change can not be computed by the algorithm because the algorithm finds configuration changes that involve at least one PDA action.

4.4 The Finiteness Condition: ϵ Boundedness

We suggest a condition for PDAs that makes the number of minimal configuration changes finite for any input string.

Definition 3 (ϵ bounded). *A PDA* **A** *is ϵ bounded if there exists a positive constant k s.t. $q_0 Z_0 \overset{*}{\mapsto} \gamma_1 \overset{\epsilon}{\mapsto} \gamma_2 \overset{\epsilon}{\mapsto} \cdots \overset{\epsilon}{\mapsto} \gamma_m$ implies $-k \leq |\gamma_1| - |\gamma_m| \leq k$.*

Theorem 3. *The followings are equivalent for the PDA* **A**.

– *The PDA is ϵ bounded.*
– *For any input, the size of the minimal configuration change is linearly bounded.*
– *For any input, the number of minimal configuration changes is finite.*

The family of ϵ bounded PDAs contains many practical PDAs. This family includes any no delay or finite delay PDAs. It contains simple machines[10] and super-deterministic PDAs[12]. This family also includes any deterministic pushdown parser whose transition involves only finite number of consecutive ϵ actions. We can directly use these PDAs for the analysis.

It turns out that the ϵ bounded PDAs can express any context-free languages. In [13], it is shown that any context-free language has an equivalent no delay PDA, which is ϵ bounded. The construction of such PDA is essentially based on a non-deterministic topdown parser for the grammar in Greibach normal form. Therefore, if the given reference PDA is not ϵ bounded, we can use another PDA which is ϵ bounded and describes the same language.

5 Formalizing String Analysis

In this section, we design a string analysis for the given reference PDA **A**. We first define the abstract domain $(\mathcal{C}_{\mathbf{A}}, \sqsubseteq)$, correctness relation as a Galois connection between $(2^{\Sigma^*}, \subseteq)$ and $(\mathcal{C}_{\mathbf{A}}, \sqsubseteq)$. We show the abstract concatenation operation and simple widening operation. We also revisit the previous motivating example equipped with the domain.

[2] This is the set of sentences derived from the non-terminal $[\gamma_m Z_m]_{i_m} \succ [\gamma Z]_*$.

5.1 The Predicate Domain

The main idea of our abstraction is that we can regard a configuration change of the reference PDA $\mathbf{A}$ as a predicate for string:

$$\delta \rightsquigarrow \delta'(w) = \exists \eta.\ (q_0 Z_0 \overset{*}{\mapsto} \delta\eta) \wedge (\eta \text{ is intact while } \delta\eta \overset{w}{\mapsto} \delta'\eta).$$

If at least one string has the configuration change $\delta \rightsquigarrow \delta'$, we say $\delta \rightsquigarrow \delta'$ is *inhabited.*

Lemma 3. *For inhabited predicates $\delta \rightsquigarrow \delta'$ and $\delta\eta \rightsquigarrow \delta'\eta$, $\delta \rightsquigarrow \delta'$ implies $\delta\eta \rightsquigarrow \delta'\eta$.*

Let us represent the above syntactic implication between inhabited predicates as $\delta \rightsquigarrow \delta' \Rrightarrow \delta\eta \rightsquigarrow \delta'\eta$. The inhabited predicates along with $\lambda w.\ true$ and $\lambda w.\ false$ form a well-founded complete lattice using '$\Rrightarrow$' as the order, where the supremum is $\lambda w.\ true$ and the infimum is $\lambda w.\ false$.

5.2 The Domain $(\mathcal{C}_\mathbf{A}, \sqsubseteq)$

Since the domain of predicates does not have the best abstraction property in general, we define the analysis domain $(\mathcal{C}_\mathbf{A}, \sqsubseteq)$ which is basically the conjunction completion[6, Example 6.6] of the predicate domain.

The set $\mathcal{C}'_\mathbf{A}$ is a collection of conjunctions of inhabited predicates. We will use $C = \bigwedge_{i\in I} \eta_i \rightsquigarrow \kappa_i$ to denote a conjunction, and c to denote one inhabited predicate. We allow set operation on the conjunctions regarding them as sets of predicates. We demand two additional conditions for each conjunction $C \in \mathcal{C}'_\mathbf{A}$:

– Each predicate of C is not related to other predicates in C by '$\Rrightarrow$'.
– The conjunction C is inhabited. i.e. $\exists w \in \Sigma^*.\ \forall i \in I.\ \eta_i \rightsquigarrow \kappa_i(w)$.

The first condition is necessary for the order $\sqsubseteq$, which is defined below, to be antisymmetric. This condition can be enforced for any set of inhabited predicates P by the *norm* function: $norm(P) = \bigwedge\{c \in P \mid \forall c' \in P.\ (c' \Rrightarrow c) \Rightarrow (c = c')\}$. The second condition prevents many different elements from having the same concrete meaning of empty set of strings. Now, the set $\mathcal{C}_\mathbf{A}$ is defined as $\mathcal{C}_\mathbf{A} = \mathcal{C}'_\mathbf{A} \cup \{\bot\}$.

Definition 4 ($\sqsubseteq$). *For $C_1, C_2 \in \mathcal{C}'_\mathbf{A}$, $C_1 \sqsubseteq C_2$ if and only if $\forall c_2 \in C_2.\ \exists c_1 \in C_1.\ c_1 \Rrightarrow c_2$. The $\bot$ is the infimum of $(\mathcal{C}_\mathbf{A}, \sqsubseteq)$.*

The $\bot$ is the identity of $\bigsqcup$, and we define $\bigsqcup \emptyset = \bot$. The *least upper bound* of non-empty $S \subseteq \mathcal{C}'_\mathbf{A}$ is

$$\bigsqcup S = \bigwedge\{c \in \bigcup S \mid \forall C' \in S.\ \exists c' \in C'.\ c' \Rrightarrow c\}.$$

Theorem 4. *The domain $(\mathcal{C}_\mathbf{A}, \sqsubseteq)$ is a complete lattice.*

5.3 The Galois Connection

The concretization of a conjunction $C \in \mathcal{C}'_{\mathbf{A}}$ is the common inhabitants of all the predicates in C,

$$\gamma(C) = \{w \in \Sigma^* \mid \forall c \in C.\ c(w)\}.$$

We define $\gamma(\bot) = \emptyset$. The abstraction of non-empty set of strings $S \subseteq \Sigma^*$ is the normalized conjunction of common predicates of all the string in S,

$$\alpha(S) = norm(\{c \mid \forall w \in S.\ c(w)\}).$$

For the empty set, we define $\alpha(\emptyset) = \bot$.

Fact 1. *If the reference PDA A is ϵ bounded, the abstraction of a string is finite and computable: the abstraction is the same as the normalized conjunction of all minimal configuration changes of the string.*

Theorem 5. *The Galois connection, $(2^{\Sigma^*}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{C}_{\mathbf{A}}, \sqsubseteq)$ holds.*

5.4 Abstract String Concatenation Operation

Now we define the abstract string concatenation operation $\odot$ for conjunctions. Generaly, the *best* abstract transfer functions on Galois connection between complete lattices can be induced by the functions α, γ and the concrete semantics after removing redundancy in the abstract domain[8]. However, we could not find the redundancy elimination as a computable function. Hence, we manually define a *correct* abstract concatenation operation.

We first consider operation $\odot_1$ on predicates:

$$\eta_1 \rightsquigarrow \kappa_1 \odot_1 \eta_2 \rightsquigarrow \kappa_2 = \begin{cases} \eta_1 \rightsquigarrow \kappa_2 \delta & \text{if } \kappa_1 = \eta_2 \delta, \\ \eta_1 \delta \rightsquigarrow \kappa_2 & \text{if } \kappa_1 \delta = \eta_2 \text{ and } \eta_1 \delta \text{ is a valid prefix,} \\ \lambda w.\ true & \text{otherwise.} \end{cases}$$

Basically, the $\odot_1$ operation is simple composition of two configuration changes. It recovers and fixes the implicit configuration as much as it is needed to compose with the other predicate. If they can not be concatenated, the result is $\lambda w.\ true$. The operation $\odot$ on the conjunction is defined as

$$C_1 \odot C_2 = norm(\{c \mid c = c_1 \odot_1 c_2 \neq \lambda w.\ true, c_1 \in C_1, c_2 \in C_2\}).$$

The $\bot$ is zero of $\odot$ operation, i.e. $\bot \odot C = C \odot \bot = \bot \odot \bot = \bot$.

Theorem 6. *The operator $\odot$ is associative, and a correct upper approximation of concatenation operation.*

5.5 The Widening Operator

```
1 x := '';
2 while e do
3     x := ')' + x + '(';
4 end
```

Although we can design a computable analysis for ϵ bounded reference PDAs thanks to Fact. 1 and $\odot$ operation, the analysis can be non-terminating. Let's consider the program on the right and the reference automaton in Fig. 1a. After the nth visit of line 3, the abstract value of the variable x at the head of the loop is $q_0X^{n+1}\rightsquigarrow q_0X^{n+1} \wedge q_0X^nZ_0\rightsquigarrow q_0X^nZ_0$.

To guarantee the termination, we use the set intersection as the widening operator on conjunctions. We can use this widening at a loop after bounded number of iterations. When the reference PDA has a decision procedure for inclusion(e.g. [12,1,16,22]), it is possible to avoid using the widening operator. Instead, we can use the result from the literature and effectively find the maximum size of predicates. However, finding the maximum size is not recommendable since it requires investigation of the global structure[3] of the program and finding a feasible left context string for each abstract value. This approach is impossible for modular analysis. Usually, the heuristic constant bound of the size or the widening operator described earlier are sufficient for verification.

5.6 Example Revisited

We will analyze the program in Fig. 1b with a sign domain for integers, and $(\mathcal{C}_\mathbf{A}, \sqsubseteq)$ for strings, simultaneously. The sign domain is $\{\lambda n.\ \mathit{false},\ \lambda n.\ n = 0,\ \lambda n.\ n > 0,\ \lambda n.\ n < 0,\ \lambda n.\ \mathit{true}\}$. One peculiar aspect of this program is that the conditional on line 5 takes *true* branch only at the first iteration and this fact is crucial to the verification. Thus, we use the trace partitioning[20] technique and achieve the effect of loop unrolling by using token set $\{*, 1, 2+\}$. Here, the token $*$ means the control is not in the loop, and the token 1 means it is in the loop and doing the first iteration, and the token 2+ means it is in the loop and after the first iteration. We use the tuple of control location, token and abstract memory state for analysis. The label l_i is for labeling the location after the statement on line i, and the abstract memory state is the set of pairs of a variable name and its abstract value. Then, the stabilized abstract value on l_{11} is $(l_{11}, *, \{i : \lambda n.\ n > 0,\ n : \lambda n.\ n > 0,\ x : q_0X\rightsquigarrow q_0XXX \wedge q_0Z_0\rightsquigarrow q_0XXZ_0\})$ which takes into account the abstract values escaped from all iterations. Finally, the abstract value after line 12 is $(l_{12}, *, \{i : \lambda n.\ n > 0,\ n : \lambda n.\ n > 0,\ x : q_0X\rightsquigarrow q_0 \wedge q_0XZ_0\rightsquigarrow q_F\})$ since the abstraction of ')))' is $q_0XXX\rightsquigarrow q_0 \wedge q_0XXXZ_0\rightsquigarrow q_F$.

From the result, we have verified that the string in x after line 12 is correct subsentence of the reference PDA. More specifically, it has the effect of popping one symbol X, and also can make up a whole sentence when it is appended to a string that has the abstract value $q_0Z_0\rightsquigarrow q_0XZ_0$. Note that we don't need any string analysis specific preprocessing for the program before the analysis.

[3] For the simpler cases where a context-free grammar can be regarded as the program, we refer you to [16,22].

6 Related Work

In [17], the authors used the idea that a string can be represented by a function in $P \rightarrow P$ that maps a parse stack to resulting parse stack after processing the string. They subsequently approximate set of strings as the mapping between abstract parse stacks. Roughly speaking, our abstraction directly uses a mapping in $P \rightarrow P$ and its functional domain restriction as an abstraction for set of strings. While [17] supports $LR(k)$ languages, ours can support all context-free languages and modular analysis. In [4], the authors devised a string analyzer in an abstract interpretation framework. Their analyzer can verify the output of program at the level of regular languages.

In [26], the author presented a type system for document-generating program. The type-checking is based on constraint generation and verifying the constraints by an extended Earley parser. This technique can be used for any context-free grammar. Since the type for string is one of the non-terminals, the supported abstraction for string is inherently finite.

Abstract parsing[9] can provide abstraction for strings as a finite mapping from one stack top to a set of stack fragment that can be pushed on the stack top after reading the strings. This technique is very fast and precise for HTML generating programs. One weak point of this technique is that, it does not provide abstraction for strings which pop the initial stack top. For example, they do not directly provide the abstraction for '))' in matched parentheses parser. This makes their abstraction not enough for designing abstract interpretation. However, they allow this kind of string to appear in the middle of data flow equations when some stack symbols already have been pushed onto the initial stack top by the preceding vocabulary string of the equation.

The analysis in [15] uses the verification and grammar transformation for parenthesis languages described in [16] to check the XML validity of the output. The characteristic function $c(x)$ and $d(x)$ in [16] for string x is closely related to our abstraction. For a string x, the values of $d(x)$ and $c(x) + d(x)$ correspond respectively to the size of η and κ, where $\eta \rightsquigarrow \kappa$ is a minimal configuration change for x. The technique in [22] uses the verification and grammar transformation for balanced languages. The ρ function in [22, Section 2] maps a string to the sequence of unmatched close tags and open tags. These close tags and open tags match respectively η and the reverse of κ. The techniques in [15,22] can check the XML validity after grammar transformation. In our approach, we can check the XML validity without requiring a seperate phase by using an ϵ bounded PDA which also encodes the XML validity condition.

Substring parsing can provide useful information about document fragments. We used the parsing algorithm in [18] with different grammar construction. Their algorithm can handle arbitrary context-free language and gives all the possible parse for the input in a form of context-free grammar. The parser in [25] also handles arbitrary context-free grammar. However, this parser does not find all possible parses. The $LR(k)$ substring recognizer in [2,11] decides whether the string is a substring of the language in linear time without constructing the parse.

7 Discussion and Ongoing Work

The algorithm in Section 4 takes $O(n^3)$ if we decompose the PDA actions to smaller ones, and listing all the minimal configuration changes can take exponential time in the worst case. However, in practical languages, the number of minimal configuration changes does not show the exponential behavior. The main reason is that as the string gets longer the number of possible context for the string usually decreases.

We have implemented our string analysis on Soot framework[24] as a proof of concept. We have tested the analyzer on small reference PDAs and programs. We are planning to analyze more realistic programs for practical reference languages.

The $LR(k)$ parsers are not ϵ bounded in general, and the abstraction of abstract parsing [9] is based on collecting possible parse stacks. We are currently working on formalizing abstract parsing using configuration changes and abstract interpretation framework[6] to support modular abstract parsing.

Acknowledgments. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2010-0001714).

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC 2004: Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing, pp. 202–211. ACM, New York (2004)
2. Bates, J., Lavie, A.: Recognizing substrings of LR(k) languages in linear time. ACM Trans. Program. Lang. Syst. 16(3), 1051–1077 (1994)
3. Billot, S., Lang, B.: The structure of shared forests in ambiguous parsing. In: Proceedings of the 27th Annual Meeting on Association for Computational Linguistics, pp. 143–151. Association for Computational Linguistics, Morristown (1989)
4. Choi, T.-H., Lee, O., Kim, H., Doh, K.-G.: A practical string analyzer by the widening approach. In: APLAS, pp. 374–388 (2006)
5. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003), `http://www.brics.dk/JSA/`
6. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM, New York (1977)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 269–282. ACM, New York (1979)
9. Doh, K.-G., Kim, H., Schmidt, D.A.: Abstract parsing: Static analysis of dynamically generated string output using LR-parsing technology. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 256–272. Springer, Heidelberg (2009)

10. Friedman, E.P.: The inclusion problem for simple languages. Theor. Comput. Sci. 1(4), 297–316 (1976)
11. Goeman, H.: On parsing and condensing substrings of LR languages in linear time. Theor. Comput. Sci. 267(1-2), 61–82 (2001)
12. Greibach, S.A., Friedman, E.P.: Superdeterministic pdas: A subcase with a decidable inclusion problem. J. ACM 27(4), 675–700 (1980)
13. Greibach, S.A.: A new normal-form theorem for context-free phrase structure grammars. J. ACM 12(1), 42–52 (1965)
14. Kim, S.-W.: Proofs for formalizing string analysis within abstract interpretation framework. Technical report, KAIST (2010), `http://pllab.kaist.ac.kr/~sewon.kim/papers/pf-saai-tr.pdf`
15. Kirkegaard, C., Møller, A.: Static analysis for java servlets and JSP. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 336–352. Springer, Heidelberg (2006)
16. Knuth, D.E.: A characterization of parenthesis languages. Information and Control 11(3), 269–289 (1967)
17. Kong, S., Choi, W., Yi, K.: Abstract parsing for two-staged languages with concatenation. In: GPCE 2009: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering, pp. 109–116. ACM, New York (2009)
18. Lang, B.: Deterministic techniques for efficient non-deterministic parsers. In: Loeckx, J. (ed.) ICALP 1974. LNCS, vol. 14, pp. 255–269. Springer, Heidelberg (1974)
19. Lang, B.: Parsing incomplete sentences. In: Proceedings of the 12th conference on Computational linguistics, pp. 365–371. ACL, Morristown (1988)
20. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
21. Minamide, Y.: Static approximation of dynamically generated web pages. In: WWW 2005: Proceedings of the 14th International Conference on World Wide Web, pp. 432–441. ACM, New York (2005)
22. Minamide, Y., Tozawa, A.: Xml validation for context-free grammars. In: APLAS, pp. 357–373 (2006)
23. Nederhof, M.-J., Bertsch, E.: Linear-time suffix parsing for deterministic languages. J. ACM 43(3), 524–554 (1996)
24. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a java optimization framework. In: Proceedings of CASCON 1999, pp. 125–135 (1999)
25. Rekers, J., Koorn, W.: Substring parsing for arbitrary context-free grammars. SIGPLAN Not. 26(5), 59–66 (1991)
26. Thiemann, P.: Grammar-based analysis of string expressions. In: TLDI 2005: Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, pp. 59–70. ACM, New York (2005)