A Practical String Analyzer by the Widening Approach[⋆]

Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh[⋆⋆]

Department of Computer Science and Engineering, Hanyang University, Ansan, Korea
{thchoi, oukseh, hhkim, doh}@pllab.hanyang.ac.kr

Abstract. The static determination of approximated values of string expressions has many potential applications. For instance, approximated string values may be used to check the validity and security of generated strings, as well as to collect the useful string properties. Previous string analysis efforts have been focused primarily on the maxmization of the precision of regular approximations of strings. These methods have not been completely satisfactory due to the difficulties in dealing with heap variables and context sensitivity. In this paper, we present an abstract-interpretation-based solution that employs a heuristic widening method. The presented solution is implemented and compared to JSA. In most cases, our solution gives results as precise as those produced by previous methods, and it makes the additional contribution of easily dealing with heap variables and context sensitivity in a very natural way. We anticipate the employment of our method in practical applications.

1 Introduction

Strings are used in many applications to build SQL queries, construct semi-structured Web documents, create XPath and JavaScript expressions, and so on. After being dynamically generated from user inputs, strings are sent to their respective processors. However, strings are not evaluated for their validity or security despite the potential usefulness of such metrics [5, 7, 6]. Hence, this paper aims to establish a method for statically determining the approximated values of string expressions in a string-generating program.

1.1 Related Works

Previous efforts to statically determine the approximated values of string expressions have attempted to maximize the precision of string approximations.

Christensen, Møller and Schwartzbach [2] developed a Java string analyzer (JSA) that approximates the values of string expressions using regular language.

[⋆] This work was supported in part by grant No.R01-2006-000-10926-0 from the Basic Research Program of the Korea Science and Engineering Foundation, and in part by Brain Korea 21.

[⋆⋆] Corresponding author

An interprocedural data-flow analysis is first used to extract context-free grammar from a Java program such that each string expression is represented as a nonterminal symbol. Then, Mohri and Nederhof's algorithm [8] is applied to approximate the context-free grammar with regular grammar. Eventually, the string analysis produces a finite state automaton that conservatively approximates the set of possible strings for each specified string expression. JSA tends to be adequate when every string value is stored in a local variable, but it falters when dealing with strings stored in heap variables. Perhaps the method could be extended to deal with such variables, but not in a straightforward and immediate manner.

To conduct string analysis based on regular expressions, Tabuchi, Sumii, and Yonezawa [11] created a type system for a minimally functional language equipped with string concatenation and pattern matching over strings. However, they failed to provide a type inference algorithm due to a technical problem with recursive constraint solving. Our analysis can be thought of as a solution to their problem based on a carefully crafted widening.

Thiemann [12] presented a type system for string analysis based on context-free grammar and provided a type inference algorithm derived from Earley's parsing algorithm. His analysis is more precise than those based on regular expressions, and though sound, his inference algorithm is incomplete because its context-free language inclusion problem cannot be solved. The weak point is that the grammar must be written in terms of single characters rather than tokens.

Minamide [7] also developed a static program analyzer that approximates string output using context-free grammar. His analyzer, which uses a variant of the JSA approach to produce context-free grammar from a PHP program, validates HTML documents generated by a PHP program either by extracting and testing sample documents or by considering documents with a bounded depth only.

1.2 Our Approach

Our work is motivated by a desire to statically determine which database application program accesses and updates which database tables and fields. Such information is particularly useful in maintaining huge enterprise software systems. To obtain this information statically, all possible SQL queries must be extracted from database application programs as strings.

Strings may be stored as field variables in object-oriented applications, thus a string analysis must be able to determine their value. For example, the Java application in Fig. 1 uses a field variable to construct strings. The class `SQLBuffer` is defined as a gateway for connecting to a database server. In this example, two `SQLBuffer` objects are allocated and each object has a separate string field, `buf`. To prevent the clouding of analysis precision, independent string fields should be maintained as such. Thus, heap memory analysis is required. Furthermore, the methods `add` and `set` are called multiple times in different contexts. As such, precise string analysis must also be context-sensitive. For the example in Fig. 1, our analyzer is able to distinguish possible queries as `SELECT .* FROM .*` and

`UPDATE .* SET .* = .*`, while JSA is unable to do so and only gives `.*` that means any string.

Our string analysis uses the standard monotone framework for abstract interpretation [3, 4], which allows for context-sensitive handling of field variables. However, use of the abstract-interpretation framework for string analysis requires the invention of a reasonable widening operator. Thus, to keep its precision as high as possible, our widening operator is designed with heuristics.

1.3 Paper Contributions

Our paper makes the following contributions:

- We design a string analyzer based on standard abstract-interpretation techniques. Until now, ascertaining widening operators for regular expressions has been believed to be difficult [2]. However, by selecting a restricted subset of regular expressions as our abstract domain, which results in limited loss of expressibility, and by using heuristics, we can devise a precise widening operator. String operators, such `concat`, `substring`, `trim`, and `replace`, are treated uniformly.
- The abstract-interpretation framework enables the integration of the following tasks into our analyzer:
  - handle memory objects and their field variables
  - recognize context sensitivity
  - integrate with constant propagation for integers
- Our string analyzer is implemented and tested. The results show the proposed analyzer to be as precise as JSAs in all cases, and even more precise for test programs dealing with memory objects and field variables.

1.4 Overview

The rest of this paper is organized as follows. Section 2 presents our key abstract domain, the set of regular strings. Section 3 explains the analysis for a simple imperative language and extends the analysis for integers, heap-manipulating statements, and procedures. Section 4 shows the experimental results, and the paper is concluded by Section 5.

2 Abstract Domain

An abstract string value is modeled as a *regular string* from within a restricted subset of regular expressions, and string operations are given abstract semantics. We first define a regular string and then explain the abstract semantics of concatenation and the widening operator. We subsequently give the abstract semantics of other string operators: `replace`, `trim`, and `substr`.

```
class SQLBuffer {
  String buf;
  Connection con;
  void set(String s) {
    buf = s;
  }
  void append(String s) {
    buf = buf + " " + s;
  }
  ResultSet execute() throws SQLException {
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(buf);
    buf = "";
    return rs;
  }
}
public class Example {
  public void some_fun(String[] args) throws SQLException {
    SQLBuffer sql1 = new SQLBuffer();
    SQLBuffer sql2 = new SQLBuffer();

    sql1.set("SELECT");
    sql1.add(args[2]);
    sql1.add("FROM");
    sql1.add(args[0]);

    ResultSet rs = sql1.execute();

    while (rs.next()) {
      sql2.set("UPDATE");
      sql2.add(args[1]);
      sql2.add("SET");
      sql2.add(args[2] + " = " + rs.getString(0));
      sql2.execute();
    }
  }
  // ...
}
```

Fig. 1. Example

2.1 Regular String

A regular string is a sequence of atoms that comprise either an abstract character or the repetition of a regular string, as shown in Fig. 2. An abstract character is a set of characters that is, in most cases, a singleton set. For brevity, we omit the set notation for a singleton set; for instance, instead of $\{a\}\,\{b,c\}\,\{d\}$, we write

Collecting domain:

$$\begin{array}{lll}
\mathsf{Var} & x & \\
\mathsf{Char} & c & \\
\mathsf{Str} & s & \in \ \{c_1 c_2 \cdots c_n \mid n \geq 0,\ c_i \in \mathsf{Char}\} \\
\mathsf{State}^{\mathsf{col}} & S & \in \ \mathcal{P}(\mathsf{Var} \rightarrow \mathsf{Str})
\end{array}$$

Abstract domain:

$$\begin{array}{lll}
\mathsf{Chars} & C & \in \ \mathcal{P}^{\mathrm{N}}(\mathsf{Char}) \text{ where } \mathcal{P}^{\mathrm{N}}(A) = \mathcal{P}(A) \setminus \{\emptyset\} \\
\mathsf{Atom} & a & ::= C \mid r^\star \\
\mathsf{Reg} & p, q, r & \in \ \{a_1 a_2 \cdots a_n \mid n \geq 0, a_i \in \mathsf{Atom},\ \neg\exists i.(a_i = p^\star \wedge a_{i+1} = q^\star)\} \\
\mathsf{State} & \sigma & \in \ (\mathsf{Var} \rightarrow \mathcal{P}^{\mathrm{N}}(\mathsf{Reg}))_\bot
\end{array}$$

Meaning:

$$\begin{array}{llcl}
\mathsf{Atom} \rightarrow \mathcal{P}(\mathsf{Str}) & \gamma_a(C) & = & C \\
 & \gamma_a(r^\star) & = & \{s_1 s_2 \cdots s_n \mid 0 \leq n, s_i \in \gamma_r(r)\} \\
\mathsf{Reg} \rightarrow \mathcal{P}(\mathsf{Str}) & \gamma_r(a_1 a_2 \cdots a_n) & = & \{s_1 s_2 \cdots s_n \mid s_i \in \gamma_a(a_i)\} \\
\mathcal{P}^{\mathrm{N}}(\mathsf{Reg}) \rightarrow \mathcal{P}(\mathsf{Str}) & \gamma_R(R) & = & \bigcup \{\gamma_r(r) \mid r \in R\} \\
\mathsf{State} \rightarrow \mathsf{State}^{\mathsf{col}} & \gamma_S(\bot) & = & \emptyset \\
 & \gamma_S(\sigma) & = & \{\lambda x.s_x \mid s_x \in \gamma_R(\sigma(x))\}
\end{array}$$

Order:

$$\begin{array}{ll}
\mathsf{Reg} & p \sqsubseteq q \ \text{ iff } \gamma_r(p) \subseteq \gamma_r(q) \\
\mathcal{P}^{\mathrm{N}}(\mathsf{Reg}) & P \sqsubseteq Q \text{ iff } \gamma_R(P) \subseteq \gamma_R(Q) \\
\mathsf{State} & \sigma \sqsubseteq \sigma' \text{ iff } \gamma_S(\sigma) \subseteq \gamma_S(\sigma')
\end{array}$$

Fig. 2. The Abstract Domain

$a\,\{b, c\}\,d$, which is equivalent to $a(b + c)d$ in regular expression. The meaning of a repetition is as usual.

A regular string is derived from a restricted subset of regular expressions, which is expressible enough for our purposes. The alternative operator $+$ is omitted, and the set notation is used to represent the collection of alternatives. Consecutive repetitions, such as $a^\star b^\star$, are not allowed. To force the termination of the analysis, the regular expression $a^\star b^\star$ is approximated as $\{a, b\}^\star$.

In an abstract state, each variable maps to the set of regular strings.

2.2 Concatenation and Widening

The abstract semantics of string concatenation is defined as follows: two regular strings are sequentially ordered, except for when initial and subsequent regular strings end and begin, respectively, with a repetition, as defined in Fig. 3. If the two repetitions are the same, one is thrown away; otherwise, the two are brutally merged.

The widening operator of two regular strings is designed minimize precision loss while allowing for analysis termination. Two sets of regular strings can be widened simply by widening every pair of two input sets, but with the possible result of an unnecessarily large string. For instance, consider where after

$$P \cdot Q = \{p \cdot q \mid p \in P,\ q \in Q\}$$

$$p \cdot q = \begin{cases} p'r^\star q' & \text{if } p = p'r^\star,\ q = r^\star q' \\ p'\,\{c \mid c \text{ appears in } r \text{ or } r'\}^\star\, q' & \text{if } p = p'r^\star,\ q = r'^\star q',\ \text{and } r \neq r' \\ pq & \text{otherwise} \end{cases}$$

$$\sigma \nabla^k \sigma' = \begin{cases} \sigma & \text{if } \sigma' = \bot \\ \sigma' & \text{if } \sigma = \bot \\ \lambda x.\, \left\{ .p \nabla^k .q \mid p \in \sigma(x),\ q \in \sigma'(x),\ p \,\mathcal{R}\, q \right\} & \text{otherwise} \\ \quad \text{for a total relation } \mathcal{R} : \sigma(x) \times \sigma'(x) & \end{cases}$$

$$p.q \nabla^k p'.q' = \begin{cases} pq \odot^k p'q' & \text{if } q = \epsilon \text{ or } q' = \epsilon \\ (p \odot^k p') \cdot a \cdot (.r \nabla^k .r') & \text{if } q = ar,\ q' = ar',\ \text{and } \textit{star-height}(a) \leq k \\ pa.r \nabla^k p'a.r' & \text{if } q = ar,\ q' = ar',\ \text{and } \textit{star-height}(a) > k \\ pa.r \nabla^k p'.a'r' & \text{if } q = ar,\ q' = a'r',\ a \neq a',\ \text{and } |q| > |q'| \\ p.ar \nabla^k p'a'.r' & \text{if } q = ar,\ q' = a'r',\ a \neq a',\ \text{and } |q| \leq |q'| \end{cases}$$
$$\text{where } |q| = n \text{ for } q = a_1 a_2 \cdots a_n$$
$$\text{and } \textit{star-height}(a) \text{ is the depth of repetitions of } q$$

$$p \odot^k q = \begin{cases} \epsilon & \text{if } p, q \in \{\epsilon\} \\ p^\star & \text{if } p \neq \epsilon,\ q = \epsilon,\ \text{and } \textit{star-height}(p^\star) \leq k \\ q^\star & \text{if } p = \epsilon,\ q \neq \epsilon,\ \text{and } \textit{star-height}(q^\star) \leq k \\ (.p' \nabla^{k-1} .q')^\star & \text{if } p = p'^\star,\ q = q'^\star,\ \text{and } k \geq 2 \\ (.p' \nabla^{k-1} .q)^\star & \text{if } p = p'^\star,\ q \neq q'^\star,\ \text{and } k \geq 2 \\ (.p \nabla^{k-1} .q')^\star & \text{if } p \neq p'^\star,\ q = q'^\star,\ \text{and } k \geq 2 \\ \{c \mid c \text{ appears in } p \text{ or } q\}^\star & \text{otherwise} \end{cases}$$

Fig. 3. Abstract Concatenation and Widening

one loop iteration of $\{a, b\}$ becomes $\{aa, ba\}$. The most reasonable analysis solution should be $\{aa^\star, ba^\star\}$, so we would want to choose $\{a\nabla aa, b\nabla bb\}$ instead of $\{a\nabla aa, a\nabla ba, b\nabla aa, b\nabla ba\}$. Hence, we define $P\nabla Q = \{p\nabla q \mid p \in P, q \in Q, p\,\mathcal{R}\,q\}$ to give total relation $\mathcal{R} : P \times Q$. The method for finding such a relation is discussed after the explanation of the widening operator for regular strings.

To widen two regular strings, we identify their common and different components, pick and leave unchanged the common parts, and then merge the different parts. For instance, suppose we compute $acd\nabla abc$, where the common components are the bold characters in $\mathbf{ac}d$ and $\mathbf{a}b\mathbf{c}$. We first pick a, then extract $b^\star$ from the merger of ϵ and b, then pick c, and then extract $d^\star$ from the merger of d and ϵ. Therefore, by concatenating the components, the two original inputs are widened to $ab^\star cd^\star$. This method is problematic, though, as the different components of multiple regular strings may be determined with different results. For instance, for cd and cdd, we can say that cd is common and the last d of cdd is different: $\mathbf{cd}$ and $\mathbf{cd}d$, or the middle d of cdd is different: $\mathbf{cd}$ and $\mathbf{c}d\mathbf{d}$. We solve this dilemma by traversing the string from left to right. The marker . is used to indicate the position of string traversal. That is, $p.q$ indicates that p has been traversed and identified as different, and that q has not been traversed. Thus the current atom is always next to the dot(.) on the right. There exist three possible cases of string traversal:

- After one regular string has been completely traversed, we conclude that the two regular strings are different. Thus, we merge them with the mash operator, $\odot$, which is discussed below.
- When we find a common atom, we merge the two different parts on the left, widen the rest of strings on the right, and then concatenate them in order.
- When two current atoms differ, we pick the longer string (the string with more atoms) and move the dot one atom to the right in the picked string.

For instance, consider the case of $.abc\nabla.ac$. First, we find that a is common and move to the adjacent string to the right, $.bc\nabla.c$, where the current atoms b and c are different. Since bc is longer than c, we conclude that b is different: $b.c\nabla.c$. We again meet the common character c, so we mash b and ϵ to obtain $b^\star$. In conclusion, abc and ac are widened to $ab^\star c$, where a and c are picked as common string components.

The mash operator $\odot$ yields precise results for the following cases.

- When one of its operands is empty, the other non-empty regular string is most likely to be repeated. Thus, the repetition of the non-empty regular string is returned. If both operands are empty, an empty regular string is returned.
- When both operands are repetitions, regular strings in the bodies of the repetitions are widened, and then the repetition of the widened result is returned.
- When only one of its operands is a repetition, a regular string in the body of the repetition and the other regular string are widened, and then the repetition of the widened result is returned.

For other cases, two regular strings are brutally mashed to conform to the form of $C^\star$.

During widening or mashing regular strings, we control the star height. The superscript k of widening operator ∇^k and mash operator $\odot^k$ indicates that the star height of the result should be less than or equal to k. In mashing two regular strings, k is decreased when we go one level deeper inside a repetition. When $k < 2$, instead of going one level deeper inside, we brutally merge two regular strings so that the star height of the result is one. In theory, we cannot guarantee the termination of our analysis without some form of star-height control. As shown by our experiments, however, our analysis seems to terminate without star-height control (i.e., $k = \infty$).

We now discuss in detail the clever widening of two sets of regular strings. The procedure aims to find the total relation of two sets so that similar regular strings are related. One pair of regular strings is more *similar* than the other if it maintains more common components. When the number of common components is equal, the pair with fewer differing components is considered to be more similar. The algorithm to find the total relation of two sets is as follows: (1) For each regular string in the smaller set, find the most similar regular string in the larger set and pick related pairs until the smaller set is empty; (2) The leftover regular strings in the larger set find their similar counterparts from the original smaller set. For instance, consider $\{a, b\}$ and $\{ba, bb, bc\}$. For a in the smaller set, we pick the most similar one ba. For b, since the leftovers bb and bc tie, we arbitrarily choose one bb. Since all regular strings in the smaller set are picked, the leftover bc finds the most similar one b from $\{a, b\}$.

Theorem 1. $\nabla^k : \mathsf{State} \times \mathsf{State} \to \mathsf{State}$ *is a widening operator which satisfies the followings:*

1. $\sigma \sqsubseteq \sigma\nabla^k\sigma'$ *and* $\sigma' \sqsubseteq \sigma\nabla^k\sigma'$*; and*
2. *the ascending chain by* ∇^k *is always finite when the cardinality of sets of regular strings is bounded.*

We only sketch the proof of the termination argument. The widening sequence of abstract states is finite if the sequence of regular strings is finite for each variable, which can be proved as follows. We can consider every regular string p as a form $r_1C_1r_2C_2r_3 \cdots C_nr_{n+1}$ where r_i is an empty string or a repetition because we do not allow adjacent repetitions. For instance, $abc^\star$ can be considered as $\epsilon a \epsilon bc^\star$. By using the canonical form, we define the *size tree* of regular strings:

– $|\epsilon|^T = \langle\omega\rangle$ where ω is an arbitrary big tree, and
– $|r_1C_1r_2C_2 \cdots C_nr_{n+1}|^T = \big\langle |r_1|^I, |C_1|, |r_2|^I, |C_2|, \cdots, |C_n|, |r_{n+1}|^I \big\rangle$ where $|C| = \mathsf{Int}(i)$ when i is the size of character set C.

where $|\epsilon|^I = \omega$, $|C^\star|^I = |C|$, and $|r^\star|^I = |r|^T$ if $r \neq C$. The order among trees is defined as: $\mathsf{Int}(i) \leq t \leq \omega$ for all tree t which is not an integer, $\mathsf{Int}(i) \leq \mathsf{Int}(j)$ if $i \geq j$, and $\langle t_1, t_2, \cdots t_n\rangle \leq \langle t'_1, t'_2, \cdots, t'_m\rangle$ if $n < m$, or $n = m$ and $t_i \leq t'_i$ for all $0 \leq i \leq n$. We proved that $|.p\nabla^k.q|^T \leq |p|^T$, and that $|.p\nabla^k.q|^T = |p|^T$ implies that $.p\nabla^k.q = p$. We also showed that every sequence $t_0, t_1, \cdots, t_n$ is finite when $t_i > t_{i+1}$ for all $0 \leq i < n$ because we limit the depth of the trees. Therefore, every sequence widened by ∇^k is finite.

2.3 Other String Operators

The abstract versions of string operators `trim`, `replace`, and `substr` are defined in Fig. 4. `replace`(c,c') replaces all occurrences of character c with character c' in the given regular string. `trim` removes blanks at both ends of the given string. However, for presentation brevity, we assume that `trim` removes blanks only at the front end. The abstract `trim` operator traverses the given regular string from left to right.

– If we reach an abstract character $\{'\ '\}$, we continue trimming.
– If we reach an abstract character C which includes a blank, we have to consider two possibilities: when the concretized character is a blank and when it is a non-blank.

Abstract operator $[\![op]\!] : \mathsf{Reg} \to \mathcal{P}(\mathsf{Reg})$ for $op ::= \mathtt{replace}(c, c') \mid \mathtt{trim} \mid \mathtt{substr}(i, j)$

$$[\![\mathtt{replace}(c,c')]\!]p = \{p\{c'/c\}\}$$

$$[\![\mathtt{trim}]\!]p = \begin{cases} [\![\mathtt{trim}]\!]q & \text{if } p = \{' \ '\}\,q \\ [\![\mathtt{trim}]\!]q \cup ((C \setminus \{' \ '\})q) & \text{if } p = Cq \text{ and } \{' \ '\} \subset C \\ [\![\mathtt{trim}]\!]q \cup (\{r' \cdot r^\star q \mid r' \in [\![\mathtt{trim}]\!]r,\ r' \neq \epsilon\} & \text{if } p = r^\star q \\ \{p\} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{substr}(i,j)]\!]p = \begin{cases} \{\epsilon\} & \text{if } i = 0 \text{ and } j = 0 \\ \{C\} \cdot [\![\mathtt{substr}(0, j-1)]\!]q & \text{if } i = 0,\ j > 0, \text{ and } p = Cq \\ [\![\mathtt{substr}(i-1, j-1)]\!]q & \text{if } i > 0,\ j > 0, \text{ and } p = Cq \\ [\![\mathtt{substr}(i,j)]\!](r \cdot r^\star q) \cup [\![\mathtt{substr}(i,j)]\!]q & \text{if } i \geq 0,\ j > 0, \text{ and } p = r^\star q \\ \{\} & \text{otherwise} \end{cases}$$

Fig. 4. Abstract String Operators

– If we reach a repetition $r^\star$, we consider two possibilities: (1) when $r^\star$ becomes empty after trimming it off, we continue trying for the rest; (2) when $r^\star$ becomes a non-empty string, we trim r off and put the result in front only when the result is not empty.
– If we reach an abstract character which does not include a blank, we stop.

$\mathtt{substr}(i, j)$ extracts a substring from the ith position to the $(j-1)$th position of the given string. When we reach a repetition $r^\star$ when finding a substring, we also consider two possibilities: (1) $r^\star$ is concretized to an empty string, and (2) $r^\star$ is concretized to a non-empty string. For possibility (2), $r^\star$ is unfolded once to yield $r \cdot r^\star$, from which substrings are extracted. Other cases are straightforward.

Previous string analyzers do not properly handle string operations. In JSA and Minamide's analyzer, string operations other than concatenation use rough approximations to break cycles of string productions [2, 7]. In our analyzer, abstract string operations are applied during analysis on demand. Hence, with our method, it is not at all an issue whether or not string operations are in cyclic productions. For example,

```
x = "a";
for(i=0; i<10; i++) {
  x = x + "b ";
  x.trim();
}
```

Our analyzer returns the exact answer: $\underline{\mathsf{ab}^\star}$, while JSA gives the most imprecise answer: $\underline{(\mathsf{a}+\mathsf{b}+\text{' '})^\star}$

3 Analysis

In this section, we describe our string analysis. We first define the analysis for a core imperative string-processing language. We next extend it to cover constant

$$
\begin{array}{lcl}
\mathcal{E}[\![e]\!] : \mathsf{State} \to \mathcal{P}(\mathsf{Reg}) \text{ for } e ::= s \mid x \mid e\texttt{+}e \mid e\,.\,op \\
\mathcal{E}[\![s]\!]\,\sigma & = & \{C_1 \cdots C_n \mid s = c_1 c_2 \cdots c_n,\ C_i = \{c_i\}\} \\
\mathcal{E}[\![x]\!]\,\sigma & = & \sigma(x) \\
\mathcal{E}[\![e_1\texttt{+}e_2]\!]\,\sigma & = & \mathcal{E}[\![e_1]\!]\,\sigma \cdot \mathcal{E}[\![e_2]\!]\,\sigma \\
\mathcal{E}[\![e\,.\,op]\!]\,\sigma & = & \bigcup\{[\![op]\!]p \mid p \in \mathcal{E}[\![e]\!]\,\sigma\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}[\![t]\!] : \mathsf{State} \to \mathsf{State} \text{ for } t ::= \texttt{skip} \mid x\texttt{:=}e \mid t\texttt{;}t \mid \texttt{if}\ t\ t \mid \texttt{while}\ t \\
\mathcal{T}[\![t]\!]\,\bot & = & \bot \\
\mathcal{T}[\![\texttt{skip}]\!]\,\sigma & = & \sigma \\
\mathcal{T}[\![x\texttt{:=}e]\!]\,\sigma & = & \begin{cases} \sigma[\mathcal{E}[\![e]\!]\,\sigma/x] & \text{if } \mathcal{E}[\![e]\!]\,\sigma \neq \emptyset \\ \bot & \text{if } \mathcal{E}[\![e]\!]\,\sigma = \emptyset \end{cases} \\
\mathcal{T}[\![t_1\,;\,t_2]\!]\,\sigma & = & \mathcal{T}[\![t_2]\!]\,(\mathcal{T}[\![t_1]\!]\,\sigma) \\
\mathcal{T}[\![\texttt{if}\ t_1\ t_2]\!]\,\sigma & = & \mathcal{T}[\![t_1]\!]\,\sigma \sqcup \mathcal{T}[\![t_2]\!]\,\sigma \\
\mathcal{T}[\![\texttt{while}\ t]\!]\,\sigma & = & \text{fix}^{\nabla}\,\lambda\sigma'.\sigma \sqcup \mathcal{T}[\![t]\!]\,\sigma'
\end{array}
$$

Fig. 5. The Analysis for the Core Language

propagation for integers. Then we show how to handle heap objects. Finally, we close this section by briefly explaining the interprocedural version.

3.1 Analysis for the Core Language

The analysis of the core imperative language is defined in Fig. 5 based on the standard abstract interpretation technique. An expression may be a string constant s, a variable x, a string concatenation $e\texttt{+}e$, or another string operator $x\,.\,op$. For a string concatenation, we use the abstract concatenation operator $\cdot$ defined in Fig. 3. For other string operators, we use their abstract version defined in Fig. 4. A statement is either a no-operation `skip`, an assignment $x\texttt{:=}e$, a sequence $t\texttt{;}t$, a conditional statement `if` t t, or a loop `while` t. For the case of a loop, we use the widening operator defined in Fig. 3 to compute a widen sequence until it is stabilized. Note that the boolean expression in conditional statement and loop is not considered.

3.2 Integers

String-manipulating programs sometimes convert integer values to strings. To increase the precision of our analysis, a constant propagation for integers is added to our analysis, as defined in Fig. 6. We assume that programs are well-typed. That is, we assume that each variable only has values of its type, and thus a widening operator may be applied to string-typed variables.

3.3 Handling Heap Objects

Our method uses a well-known technique [1] to handle heap objects: (1) a heap object is abstracted by its allocation site; for instance, two heap objects allocated

$$
\begin{array}{ll}
\text{Abstract domain:} & \textsf{Value } V \in \mathcal{P}^{\mathrm{N}}(\textsf{Reg}) + \mathbf{Z}^{\top} \\
 & \textsf{State } \sigma \in (\textsf{Var} \rightarrow \textsf{Value})_{\bot} \\
\text{Order:} & \\
 & \textsf{Value } V \sqsubseteq V' \text{ iff } V, V' \subseteq \textsf{Reg} \text{ and } \gamma_R(V) \subseteq \gamma_R(V') \\
 & \qquad\qquad\quad \text{or } V, V' \in \mathbf{Z}^{\top} \text{ and } (V' = \top \text{ or } V = V') \\
 & \textsf{State } \sigma \sqsubseteq \sigma' \text{ iff } \sigma(x) \sqsubseteq \sigma'(x) \text{ for all } x \in \textsf{Var}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{I}[\![ie]\!] : \textsf{State} \rightarrow \mathbf{Z}^{\top} & & \text{for } ie ::= \; i \in \mathbf{Z} \mid x \mid ie \; iop \; ie \text{ for } iop \in \{+, -, \times, \cdots\} \\
\mathcal{I}[\![i]\!]\sigma & = & i \\
\mathcal{I}[\![x]\!]\sigma & = & \sigma(x) \\
\mathcal{I}[\![ie_1 \; iop \; ie_2]\!]\sigma & = & \begin{cases} \mathcal{I}[\![ie_1]\!]\sigma \; iop \; \mathcal{I}[\![ie_2]\!]\sigma & \text{if } \mathcal{I}[\![ie_1]\!]\sigma \neq \top \text{ and } \mathcal{I}[\![ie_2]\!]\sigma \neq \top \\ \top & \text{if } \mathcal{I}[\![ie_1]\!]\sigma = \top \text{ or } \mathcal{I}[\![ie_2]\!]\sigma = \top \end{cases} \\
\mathcal{T}[\![t]\!] : \textsf{State} \rightarrow \textsf{State} & & \text{for } t ::= \cdots \mid x\texttt{:=}ie \\
\mathcal{T}[\![x\texttt{:=}ie]\!]\,\sigma & = & \sigma[\mathcal{I}[\![ie]\!]\sigma / x]
\end{array}
$$

Fig. 6. The Extension for Integers.

at the same program point are summarized as one abstract heap object; and (2) for each abstract heap object, we record the number of heap objects that are abstracted. This information is used to strongly update the content of a heap object. If an abstract heap object represents only one heap object, we can strongly update its content; otherwise, we cannot.

In the extended abstract domain for handling heap memory, shown in Fig 7, the location domain identifies allocation sites. The value domain is extended to include locations and a null-pointer value. The heap domain is a partial map from locations to their possible objects. An object consists of one value because we only consider objects with size equal to one. In addition, every object is tagged to indicate whether it is unique.

The analysis extended to deal with three heap-manipulating statements is defined in Fig 7. The additional statements are an allocation statement $x\texttt{:=new}^l$, a load statement $x\texttt{:=[}y\texttt{]}$, and a store statement $\texttt{[}x\texttt{]:=}y$. Note that every allocation statement is marked with a label, the size of every object is always one, and we assume that the initial value for a new heap object is `nil`.

- For the allocation statement $x\texttt{:=new}^l$, if there is no heap object previously abstracted as l, that is, l is not in the domain of the abstract heap, we add a new object to the abstract heap, initialize its content as `nil`, and tag it with 1. Otherwise, that is, if there already exist some objects abstracted by l, we weakly update its content by the initial value `nil` and tag it with ω.
- For the load statement $x\texttt{:=[}y\texttt{]}$, we get the content of y from the abstract heap and update x.
- For the store statement $\texttt{[}x\texttt{]:=}y$, if x points to a single, unique object, we strongly update its content. Otherwise, we weakly update the content of objects that x may point to.

Abstract domain:

$$
\begin{array}{lll}
\mathsf{Loc} & l & \\
\mathsf{Value} & V & \in \mathcal{P}^{\mathrm{N}}(\mathsf{Reg}) + \mathbf{Z}^{\top} + \mathcal{P}^{\mathrm{N}}(\mathsf{Loc} + \{\mathtt{nil}\}) \\
\mathsf{State} & \sigma & \in \mathsf{Var} \to \mathsf{Value} \\
\mathsf{Uniqueness} & u & \in \{1, \omega\} \\
\mathsf{Content} & V^{u} & \in \mathsf{Value} \times \mathsf{Uniqueness} \\
\mathsf{Heap} & h & \in \mathsf{Loc} \rightharpoonup \mathsf{Content}
\end{array}
$$

Order:

$$
\begin{array}{lll}
\mathsf{Value} & V \sqsubseteq V' & \text{iff } V, V' \subseteq \mathsf{Reg} \text{ and } \gamma_R(V) \subseteq \gamma_R(V') \\
 & & \text{or } V, V' \in \mathbf{Z}^{\top} \text{ and } (V' = \top \text{ or } V = V') \\
 & & \text{or } V, V' \subseteq \mathsf{Loc} \cup \{\mathtt{nil}\} \text{ and } V \subseteq V' \\
\mathsf{State} & \sigma \sqsubseteq \sigma' & \text{iff } \sigma(x) \sqsubseteq \sigma'(x) \text{ for all } x \in \mathsf{Var} \\
\mathsf{Uniqueness} & 1 \sqsubseteq \omega & \\
\mathsf{Content} & V_1^{u_1} \sqsubseteq V_2^{u_2} & \text{iff } V_1 \sqsubseteq V_2 \text{ and } u_1 \sqsubseteq u_2 \\
\mathsf{Heap} & h_1 \sqsubseteq h_2 & \text{iff } \mathrm{dom}(h_1) \subseteq \mathrm{dom}(h_2) \\
 & & \text{and } h_1(l) \sqsubseteq h_2(l) \text{ for all } l \in \mathrm{dom}(h_1) \\
(\mathsf{State} \times \mathsf{Heap})_{\perp} & \perp \sqsubseteq (\sigma, h) & \\
 & (\sigma_1, h_1) \sqsubseteq (\sigma_2, h_2) & \text{iff } \sigma_1 \sqsubseteq \sigma_2 \text{ and } h_1 \sqsubseteq h_2
\end{array}
$$

$$\mathcal{T}[\![t]\!] : (\mathsf{State} \times \mathsf{Heap})_{\perp} \to (\mathsf{State} \times \mathsf{Heap})_{\perp} \text{ for } t ::= \cdots \mid x\mathtt{:=new}^{l} \mid x\mathtt{:=[}x\mathtt{]} \mid \mathtt{[}x\mathtt{]:=}y$$

$$
\begin{array}{lcl}
\mathcal{T}[\![t]\!]\,\perp & = & \perp \\
\mathcal{T}[\![x\mathtt{:=new}^{l}]\!]\,(\sigma, h) & = & \begin{cases} (\sigma[\{l\}/x], h[\{\mathtt{nil}\}^{1}/l]) & \text{if } l \notin \mathrm{dom}(h) \\ (\sigma[\{l\}/x], h[(V \cup \{\mathtt{nil}\})^{\omega}/l]) & \text{if } l \in \mathrm{dom}(h) \text{ and } h(l) = V^{u} \end{cases} \\
\mathcal{T}[\![x\mathtt{:=[}y\mathtt{]}]\!]\,(\sigma, h) & = & \begin{cases} (\sigma[V'/x], h) & \text{if } V' \neq \emptyset \\ \perp & \text{if } V' = \emptyset \end{cases} \\
 & & \text{where } V' = \bigcup \{V \mid l \in \sigma(y),\ h(l) = V^{u}\} \\
\mathcal{T}[\![\mathtt{[}x\mathtt{]:=}y]\!]\,(\sigma, h) & = & \begin{cases} (\sigma, h[\sigma(y)^{1}/l]) & \text{if } \sigma(x) = \{l\} \text{ and } h(l) = V^{1} \\ (\sigma, h') & \text{otherwise} \end{cases}
\end{array}
$$

$$
\text{where } h' = \lambda l. \begin{cases} h(l) & \text{if } l \in \mathrm{dom}(h) \text{ and } l \notin \sigma(x) \\ (\sigma(y) \cup V)^{u} \text{ where } h(l) = V^{u} & \text{if } l \in \mathrm{dom}(h) \text{ and } l \in \sigma(x) \\ \text{undefined} & \text{if } l \notin \mathrm{dom}(h) \end{cases}
$$

Fig. 7. The Extension for the Heap

These statements may be straightforwardly extended to other cases. For the loop case, we apply widening to regular strings in both the abstract state and abstract heap.

3.4 Interprocedural Analysis

The interprocedural version of our analysis employs a standard technique named 1-CFA [10, 9]. We collect the possible states of each procedure at all of its call sites, making it possible to output states by computing the procedure body. The analysis result is achieved by a fixed-point iteration. If the procedure is called more than twice at different call sites, we separately keep the abstract state for

Example	**Lines**	**Hotspots**	**Calls**	**Objects**	**Loops**	**JSA**(s)	**OSA**(s)
`Switch`	21	1	1	0	0	1.33	0.42
`ReflectTest`	50	2	15	2	2	1.6	0.43
`SortAlgorithms`	54	1	3	0	0	1.35	0.4
`CarShop`	56	2	8	2	0	1.39	0.51
`ProdConsApp`	3,496	3	1,224	311	34	9.95	25.12
`Decades`	26	1	9	0	2	1.91	0.47
`SelectFromPer`	51	1	16	0	1	1.61	0.39
`LoadDriver`	78	1	20	0	1	1.84	0.4
`DB2Appl`	105	2	26	0	1	1.74	0.48
`AxionExample`	162	7	76	1	1	1.83	0.59
`Sample`	178	4	47	0	1	2.08	0.55
`GuestBookServlet`	344	4	131	6	3	4.18	0.71
`DBTest`	384	5	127	13	3	2.88	1.19
`CoercionTest`	591	4	378	18	11	18.38	1.58
`CustomFieldsMain`	1648	17	451	24	4	2.96	0.93
`CustomProxiesMain`	477	9	76	8	1	1.97	0.72
`CustomSequenceMain`	280	9	38	3	2	1.12	0.47
`ExternalizationFieldsMain`	666	2	164	21	0	2.09	1.52
`TextIndexMain`	396	8	71	11	6	1.51	0.46

Fig. 8. Experimental Results

each call site, and separately compute the procedure body for each call site. This is made possible by annotating contexts to abstract states. Since we use the 1-CFA technique, in which the context keeps the last call site only, the analysis precision can be blurred for nested calls.

Since recursive procedures may induce non-termination of our analysis, we also compute the widening sequence of inputs and outputs of the methods.

4 Experiments

We built a string analyzer for Java applications that employs our approach, and we tested its performance and precision for comparison with JSA. For a Java application with hotspots[1], our string analyzer produces a set of regular strings for each hotspot. We used Objective Caml as the implementation language and a Linux PC with an Intel PentiumD 830 processor (3.0 GHz) and 2 GByte memory.

The table in Fig. 8 shows the experimental results of 19 programs. The first 14 programs were those tested by JSA, and the final 5 programs were selected from sample programs in the BEA Kodo^{TM} Enterprise Data Access library. Both JSA and our string analyzer were tested for comparison. The number of lines ranged from 21 to 3,496. To show the characteristics of programs, we collected

[1] A hotspot is the program point where an interesting string expression is located.

Example	**JSA**	**OSA**
`CustomFieldsMain`	`Inserted: CustomFields<.*>:.* name: .* male: .* point: .* xml: .*`	`{Inserted: CustomFields<.*>:.* name: name.* male: false.* point: .*[x=1,y=2].* xml: .*}`
`ProdConsApp`	`.*`	`{Adv_SyncGet, Adv_SyncPut, .*}`
`SortAlgorithms`	`DefaultSortAlgorithms$(Counting+Quick)Sort`	`{.*}`

Fig. 9. Precision Comparison

the number of hotspots, the number of method calls, the number of new statements, and the number of loops. Columns JSA and OSA indicate analysis run times, in seconds, of JSA and our string analyzer. Our string analyzer completed analysis more quickly than JSA of all programs except the `ProdConsApp`, for which our analyzer was about 2.6 times slower. The speed-up is probably due to the implementation language used (Java versus OCaml). For the slower case, we guess that the large number of calls increased the number of times that method bodies were analyzed.

The results produced by our analyzer have been as precise as those yielded by JSA in most of the cases we have tested. However, the precision of some results differed, as shown in Fig. 9. For `CustomFieldsMain`, our analyzer gives more precise results due to its ability to analyze heap variables. For `ProdConsApp`, our string analyzer gives extra information than does JSA[2], as the two sets of regular strings are unioned when they are combined. On the other hand, JSA gives better results for `SortAlgorithms` because our current implementation ignores arrays.

5 Conclusion and Future Works

A string analyzer based on the abstract-interpretation framework is designed and implemented. A carefully crafted widening operator is devised to maintain the highest possible precision. Our solution generally gives results comparable to those of previous methods, and it understands heap variables and context sensitivity unlike others. We expect the method to be more suitable to practical applications.

Our string analyzer uses regular expressions that lack the expressibility required for checking the syntax of generated strings and for handling strings with escaped characters. Future work could aim to produce abstract string representations with more expression power while still employing the widening operator of our method.

[2] In theory, two results have the same precision. However, the extra information we get can be useful in practice.

Acknowledgement

The authors would like to thank anonymous reviewers for thoughtful comments and corrections.

References

1. David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310. ACM Press, 1990.
2. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, June 2003.
3. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
4. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
5. Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the International Conference on Software Engineering*, pages 645–654, May 2004.
6. Christian Kirkegaard and Anders Møller. Static analysis for Java servlets and JSP. In *Proceedings of the International Static Analysis Symposium*, August 2006.
7. Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International World Wide Web Conference Committee*, pages 432–441, 2005.
8. M. Mohri and M.-J. Nederhof. Regular approximation of context-free grammars through transformation. In J.-C. Junqua and G. van Noord, editors, *Robustness in Language and Speech Technology*, pages 153–163. Kluwer Academic Publisher, 2001.
9. Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 332–345. ACM Press, 1997.
10. Olin Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.
11. Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular expression types for strings in a text processing language. In *Proceedings of Workshop on Types in Programming*, pages 1–18, July 2002.
12. Peter Thiemann. Grammar-based analysis string expressions. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 59–70, 2004.