Static Analysis of String Values

Giulia Costantini[1], Pietro Ferrara[2], and Agostino Cortesi[1]

[1] University Ca' Foscari of Venice, Italy
{costantini,cortesi}@dsi.unive.it
[2] ETH Zurich, Switzerland
{pietro.ferrara}@inf.ethz.ch

Abstract. In this paper we propose a unifying approach for the static analysis of string values based on abstract interpretation, and we present several abstract domains that track different types of information. In this way, the analysis can be tuned at different levels of precision and efficiency, and it can address specific properties.

1 Introduction

Strings are widely used in modern programming languages. Their applications vary from providing an output to a user to the construction of programs executed through reflection. For instance, in `PHP` strings can be a way of communicating programs, while in `Java` they are widely used as SQL queries, or to access information about the classes through reflection. The execution of `str.substring(str.indexOf('a'))` raises an exception if `str` does not contain an `'a'` character: in this case, it would be useful being able to track the characters surely contained on the variable `str`. As another example, when dealing with SQL queries, what happens if we execute the query "`DELETE FROM Table WHERE ID =` " + `id` when `id` is equal to "`10 OR TRUE`"? The content of `Table` would be permanently erased! It's clear that a wrong manipulation of strings could lead not only to subtle exceptions, but to dramatic and permanent effects as well [20].

For all these reasons, the interest on approaches that automatically analyse and discover bugs on strings is constantly raising. On the other hand, the state-of-the-art in this field is still limited: approaches that rely on automata and use regular expressions are precise but slow, and they do not scale up [14, 24, 21, 13], while many other approaches are focused on particular properties or class of programs [10, 18, 12]. Genericity and scalability are the main advantages of the abstract interpretation approach [4, 5], though its instantiation to textual values has been quite limited up to now.

The main contribution of this paper is the formalisation of a *unifying* abstract interpretation based framework for string analysis, and its instantiations with *four different domains* that track distinct types of information. In this way, we can tune the analysis at diversified levels of *accuracy*, yielding to faster and rougher, or slower but more precise string analyses.

```
var query = "SELECT ’$\$$’ ||
  (RETAIL/100) FROM INVENTORY WHERE ";
if (l != null)
 query = query+"WHOLESALE > "+l+" AND ";

var per = "SELECT TYPECODE, TYPEDESC FROM
  TYPES WHERE NAME = ’fish’ OR NAME = ’meat’";
query = query+"TYPE IN (" + per + ");";
return query;
```

(a) The first running example

```
string x = "a";
while(cond)
  x = "0" + x + "1";
return x;
```

(b) The second running example

Fig. 1. The running examples

We inspired our work looking at the approach adopted for numerical domains for static analysis of software [7, 11, 19]. The interface of a numerical domain is nowadays standard: each domain has to define the semantics of arithmetic expressions (like $\mathtt{i} + \mathtt{5}$) and boolean conditions (like $\mathtt{i} < \mathtt{5}$). Similarly, we consider a limited list of basic string operators that can be easily extended to the various programming languages. The concrete semantics of these operators is approximated in the four different abstract domains. In addition, after 30 years of practice with numerical domains, it is clear that a monolithic domain precise on any program and property (e.g., Polyhedra [7]) gives up in terms of efficiency, while to achieve scalability we need specific approximations on a given property (e.g., Pentagons [17]) or class of programs (e.g., ASTRÉE [6]). With this scenario in mind, we develop several domains inside the same framework to tune the analysis at different levels of precision and efficiency w.r.t. the analysed program and property. Other abstractions are possible and welcomed, and we expect our framework to be generic enough to support them.

The paper is structured as follows. In the rest of this Section we introduce two running examples, and we recall some basics of abstract interpretation. Section 2 defines the syntax of the string operators we will consider. Section 3 introduces the concrete semantics, while in Section 4 the abstract domains are formalised. Finally, Section 5 discusses the related work, and Section 6 concludes.

1.1 Running Examples

Along the paper, we will always refer to the two examples reported in Tables 1(a) and 1(b). The first Java program is taken from [10], and it dynamically builds an SQL query by concatenating some strings. One of these concatenations applies only if a certain value (unknown at compile time) is not null. We are interested in checking if the SQL query resulting by the execution of such code is well formed. For the sake of readability, we will use some shortcuts to identify string constants of this program, as reported in Table 1. The second program modifies a string inside a while loop whose condition cannot be statically evaluated. Therefore, we will need to apply a widening operator [2] to force

Name	String constant
s_1	“SELECT ′\$′ || (RETAIL/100) FROM INVENTORY WHERE ”
s_2	“WHOLESALE > ”
s_3	“ AND ”
s_4	“SELECT TYPECODE, TYPEDESC FROM TYPES WHERE NAME = ′fish′ OR NAME = ′meat′”
s_5	“TYPE IN (”
s_6	“);”

Table 1. Shortcuts of string constants in the first running example

the convergence of the analysis. Intuitively, this program produces strings in the form “$0^n a 1^n$”.

1.2 Abstract Interpretation

Abstract interpretation is a theory to define and soundly approximate the semantics of a program [4, 5], focusing on some runtime properties of interest. Usually, each concrete state is composed by a set of elements (e.g., all the possible computational states), that is approximated by an unique element in the abstract domain. Formally, the concrete domain $\wp(\mathsf{D})$ forms a complete lattice $\langle \wp(\mathsf{D}), \subseteq, \emptyset, \mathsf{D}, \cup, \cap \rangle$. On this domain, a semantics $\mathbb{S}$ is defined. In the same way, an abstract semantics is defined, and it is aimed to approximate the concrete one in a computable way. Formally, the abstract domain $\overline{\mathsf{A}}$ has to form a complete lattice $\langle \overline{\mathsf{A}}, \leq_{\overline{\mathsf{A}}}, \bot_{\overline{\mathsf{A}}}, \top_{\overline{\mathsf{A}}}, \sqcup_{\overline{\mathsf{A}}}, \sqcap_{\overline{\mathsf{A}}} \rangle$. The concrete elements are related to the abstract domain by a concretization $\gamma_{\overline{\mathsf{A}}}$ and an abstraction $\alpha_{\overline{\mathsf{A}}}$ functions. In order to obtain a sound analysis, we require that the abstraction and concretization functions above form a Galois connection. An abstract semantics $\overline{\mathbb{S}}$ is defined as a sound approximation of the concrete one, i.e., $\forall \overline{\mathsf{a}} \in \overline{\mathsf{A}} : \alpha_{\overline{\mathsf{A}}} \circ \mathbb{S}[\![\gamma_{\overline{\mathsf{A}}}(\overline{\mathsf{a}})]\!] \leq_{\mathsf{A}} \overline{\mathbb{S}}[\![\overline{\mathsf{a}}]\!]$.

When abstract domains do not satisfy the ascending chain condition, a widening operator $\nabla_{\overline{\mathsf{A}}}$ is required in order to guarantee the convergence of the fixed point computation. This is an upper bound operator such that for all increasing chains $\overline{\mathsf{a}}_0 \leq_{\overline{\mathsf{A}}} \dots \overline{\mathsf{a}}_n \leq_{\overline{\mathsf{A}}} \dots$ the increasing chain defined as $\overline{\mathsf{w}}_0 = \overline{\mathsf{a}}_0, \dots, \overline{\mathsf{w}}_{i+1} = \overline{\mathsf{w}}_i \nabla_{\mathsf{A}} \overline{\mathsf{a}}_{i+1}$ is not strictly increasing.

2 Syntax

Different languages define different operators on strings, and usually each language supports a huge set of such operators: in Java 1.6 the `String` class contains 65 methods plus 15 constructors, `System.Text` in .Net contains about 12 classes that work with Unicode strings, and PHP provides 111 string functions. Considering all these operators would be quite verbose, and in addition the most part of them perform similar actions using slightly different data. We restrict our description on a small but representative set of common operators. We chose these

$$
\begin{aligned}
&\mathbb{S}[\![\texttt{new String(str)}]\!]() = \{\texttt{str}\}\\
&\mathbb{S}[\![\texttt{concat}]\!](\mathsf{S}_1, \mathsf{S}_2) = \{\mathsf{s}_1\mathsf{s}_2 : \mathsf{s}_1 \in \mathsf{S}_1 \land \mathsf{s}_2 \in \mathsf{S}_2\}\\
&\mathbb{S}[\![\texttt{readLine}]\!]() = \mathsf{S}\\
&\mathbb{S}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\mathsf{S}_1) = \{\mathsf{c}_\mathsf{b}..\mathsf{c}_\mathsf{e} : \mathsf{c}_1..\mathsf{c}_\mathsf{n} \in \mathsf{S}_1 \land \mathsf{n} \geq \mathsf{e} \land \mathsf{b} \leq \mathsf{e}\}\\
&\mathbb{B}[\![\texttt{contains}_\mathsf{c}]\!](\mathsf{S}_1) = \begin{cases} \mathsf{true} \text{ if } \forall \mathsf{s} \in \mathsf{S}_1 : \mathsf{c} \in \mathit{char}(\mathsf{s}) \\ \mathsf{false} \text{ if } \forall \mathsf{s} \in \mathsf{S}_1 : \mathsf{c} \notin \mathit{char}(\mathsf{s}) \\ \top_\mathsf{B} \quad \text{otherwise} \end{cases}
\end{aligned}
$$

Table 2. The concrete semantics, where $\top_\mathsf{B}$ represents that the condition could be evaluated to true or false depending on the string in S_1 we are considering.

operators looking at some case studies. Other operators can be easily added to our approach. For each operator, this would mean to define its concrete semantics, and its approximations on the different domains we will introduce.

A common operation is the creation of a new constant string (`new String(str)` where `str` is a sequence of characters). Usually programs concatenate strings (`concat(s1, s2)` where `s1` and `s2` are strings), read inputs from the user (`readLine()`), and take a substring of a given string ($\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}(\texttt{s})$, where `s` is a string, and `b` and `e` are integer values) as well. A common test is to check if a string contains a character ($\texttt{contains}_\mathsf{c}(\texttt{s})$, where `s` is a string and `c` is a character).

3 Concrete Domain and Semantics

3.1 Concrete Domain

Our concrete domain is simply made of strings. Given an alphabet K, that is a finite set of characters, we define strings as (possibly infinite) sequences of characters. Formally, $\mathsf{S} = \mathsf{K}^*$, where A^* is an ordered sequence of elements in A, that is, $\mathsf{A}^* = \{\mathsf{a}_1 \cdots \mathsf{a}_n : \forall \mathsf{i} \in [1..\mathsf{n}] : \mathsf{a}_i \in \mathsf{A}\}$. A string variable in our program could have different values in different executions, and our goal is to approximate all these values (potentially infinite, e.g., when dealing with user input) in a finite, computable, and hopefully efficient manner. Our lattice will be made of sets of strings. As usual in abstract interpretation, the partial order is the set inclusion. Formally, our concrete domain is defined by $\langle \wp(\mathsf{S}), \subseteq, \emptyset, \mathsf{S}, \cup, \cap \rangle$.

3.2 Semantics

Table 2 formalises the concrete semantics. For each statement of the language we introduced in Section 2, we define its semantics. For the first four statements, we define a semantics $\mathbb{S}$ that, given the statement and eventually some sets of concrete string values in S, returns a set of strings resulting from that operation. The semantics of `new String(str)` returns a singleton containing `str`, while the semantics of `readLine` returns a set containing all the possible strings, since we may read any string from the standard input. The semantics of `concat` returns all the possible concatenations of a string taken from the first set and a string taken from the second set (we denote by $\mathsf{s}_1\mathsf{s}_2$ the concatenation of strings s_1 and s_2), while the semantics of $\mathtt{substring}_\mathtt{b}^\mathtt{e}$ returns all the substrings from the $\mathtt{b}$-th to $\mathtt{e}$-th character of the given strings (note that if one of the strings is too short, there is not any substring for it in the resulting set, since this would cause a runtime error without producing any value). For $\mathtt{contains}_\mathtt{c}$ we define a particular semantics $\mathbb{B} : [\wp(\mathsf{S}) \rightarrow \{\mathsf{true}, \mathsf{false}, \top_\mathsf{B}\}]$ that, given a set of strings, returns true if all the strings contains the character c, false if none contains this character, and $\top_\mathsf{B}$ otherwise. This special boolean value represents a situation in which the boolean condition may be evaluated to true some times, and to false other times. We denoted by *char* a function that returns the set of characters contained in the string in input.

4 Abstract Domains and Semantics

What is the *relevant* information contained in a string? How can we approximate it in an *efficient* way? Tracking both sound and precise information at compile time on strings in an efficient way is infeasible. Then we need to introduce *approximation.* We want to track information precise enough to efficiently analyse the behaviours of interest, considering the string operators we defined in the previous section. Our purpose is to approximate strings as much as we can, preserving the information we deem relevant.

4.1 Character Inclusion

For the first abstract domain we aim at approximating a string through the characters we know it surely contains or it could contain. This information could be particularly useful to track if the indexes extrapolated from a string with operators like $\mathtt{indexOf(c)}$ could be used to cut the string (because $\mathtt{c}$ is surely contained in the string), or they could be invalid (e.g., -1). A string will be represented by a pair of sets: the set of *certainly* contained characters $\overline{\mathsf{C}}$ and the set of *maybe* contained characters $\overline{\mathsf{MC}}$ ($\overline{\mathcal{CI}} = \{(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) : \overline{\mathsf{C}}, \overline{\mathsf{MC}} \in \wp(\mathsf{K}) \wedge \overline{\mathsf{C}} \subseteq \overline{\mathsf{MC}}\} \cup \bot_{\overline{\mathcal{CI}}}$). The partial order $\leq_{\overline{\mathcal{CI}}}$ on $\overline{\mathcal{CI}}$ is the following one:

$$(\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1) \leq_{\overline{\mathcal{CI}}} (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2) \Leftrightarrow (\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1) = \bot_{\overline{\mathcal{CI}}} \vee (\overline{\mathsf{C}}_1 \supseteq \overline{\mathsf{C}}_2 \wedge \overline{\mathsf{MC}}_1 \subseteq \overline{\mathsf{MC}}_2)$$

This is because the more information we have on the string (that is, the more characters are certainly contained and the less characters are maybe contained), the less number of strings we are representing. For example the abstract element represented by the pair $(\{a\}, \{a\})$ is more precise than the one represented by $(\emptyset, \{a, b\})$. In fact, the first pair represents the concrete set of strings $\{a, aa, aaa, \dots\}$ while the second pair corresponds to $\{\epsilon, a, b, aa, bb, ba, ab, \dots\}$. For these reasons, the least upper bound is defined by $\sqcup_{\overline{\mathcal{CI}}}((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)) = (\overline{\mathsf{C}}_1 \cap \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2)$, and the greatest lower bound is defined by $\sqcap_{\overline{\mathcal{CI}}}((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)) = (\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cap \overline{\mathsf{MC}}_2)$. The widening operator corresponds to the $\sqcup_{\overline{\mathcal{CI}}}$ operator, and it ensures the convergence of the analysis since we supposed that the alphabet is finite. The top element of the lattice is $\top_{\overline{\mathcal{CI}}} = (\emptyset, \mathsf{K})$, while the bottom element $\bot_{\overline{\mathcal{CI}}}$ corresponds to a "failure" state.

$$\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{new String(str)}]\!]() = (\mathit{char}(\mathsf{str}), \mathit{char}(\mathsf{str}))$$
$$\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{concat}]\!]((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)) = (\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2)$$
$$\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{readLine}]\!]() = (\emptyset, \mathsf{K})$$
$$\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!]((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1)) = (\emptyset, \overline{\mathsf{MC}}_1)$$
$$\overline{\mathbb{B}_{\mathcal{CI}}}[\![\texttt{contains}_{\mathsf{c}}]\!]((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1)) = \begin{cases} \mathsf{true} \text{ if } \mathsf{c} \in \overline{\mathsf{C}}_1 \\ \mathsf{false} \text{ if } \mathsf{c} \notin \overline{\mathsf{MC}}_1 \\ \top_{\mathsf{B}} \quad \text{otherwise} \end{cases}$$

Table 3. The abstract semantics of $\overline{\mathcal{CI}}$

#I	Var	$\overline{\mathcal{CI}}$
1	query	$\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_1)$
3	l	$(\emptyset, \mathsf{K})$
3	query	$(\pi_1(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_1)) \cup \pi_1(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_2)) \cup$ $\pi_1(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_3)), \mathsf{K})$
4	query	$(\pi_1(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_1)), \mathsf{K})$
5	per	$\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_1)$
7	query	$(\pi_1(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_1)) \cup \pi_1(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_4)) \cup$ $\pi_1(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_5)) \cup \pi_1(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}_6)), \mathsf{K})$

(a) First running example

#I	Var	$\overline{\mathcal{CI}}$
1	x	$(\{a\}, \{a\})$
3	x	$(\{0, a, 1\}, \{0, a, 1\})$
4	x	$(\{a\}, \{0, a, 1\})$

(b) Second running example

Fig. 2. The results of $\overline{\mathcal{CI}}$

The function which abstracts a single string s is: $\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}) = (\mathit{char}(\mathsf{s}), \mathit{char}(\mathsf{s}))$. The abstraction function takes us from a set of strings to an element in $\overline{\mathcal{CI}}$, and it returns the upper bound of the abstraction of all the concrete strings. Let π_i be the projection on the i-th component of a tuple.

$$\alpha_{\overline{\mathcal{CI}}}(\mathsf{S}_1) = \bigsqcup_{\overline{\mathcal{CI}}, \mathsf{s} \in \mathsf{S}_1} \alpha'_{\overline{\mathcal{CI}}}(\mathsf{s}) = (\bigcap_{\mathsf{s} \in \mathsf{S}_1} \pi_1(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s})), \bigcup_{\mathsf{s} \in \mathsf{S}_1} \pi_2(\alpha'_{\overline{\mathcal{CI}}}(\mathsf{s})))$$

Semantics Table 3 defines the abstract semantics of the operators introduced in Section 2 on the abstract domain $\overline{\mathcal{CI}}$. We denote by $\overline{\mathbb{S}_{\mathcal{CI}}}$ and $\overline{\mathbb{B}_{\mathcal{CI}}}$ the abstract counterparts of $\mathbb{S}$ and $\mathbb{B}$ respectively.

When we evaluate a string, we know that the characters that are surely or maybe included are exactly the ones that appear in the string. The concatenation of two strings will contain all the characters that are surely or maybe contained in the two strings. `readLine` returns a top value, while if we take a substring of a given string, the result will possibly contain all the characters that are possibly contained in the initial string, while we know nothing about the surely contained characters. Finally, the semantics of $\texttt{contains}_{\mathsf{c}}$ is quite precise, as it checks if a character is surely contained or not contained respectively through $\overline{\mathsf{C}}$ and $\overline{\mathsf{MC}}$.

Running Example Consider the examples introduced in Section 1.1. The results of the analysis of the first program using $\overline{\mathcal{CI}}$ are depicted in Figure 2(a). At the beginning, variable `query` is related to a state that contains the abstraction of c_1, that is, both $\overline{\mathsf{C}}$ and $\overline{\mathsf{MC}}$ contain all the characters of s_1. Since we do not know the value of `l`, we compute the least upper bound between the abstract values of `query` after instructions 1 and 3. In this way, we obtain that after the `if` statement the abstract value of `query` contains the abstraction of s_1 in the $\overline{\mathsf{C}}$ component (since it surely contains all the characters of that constant string), and the top value in the $\overline{\mathsf{MC}}$ component (since we may have concatenated a string that may contain any character). At the end of the given code, `query` surely contains the characters of s_1, s_4, s_5, and s_6, and it may contain any character, since we possibly concatenated in `query` an input string (the `l` variable). As for the second program, in Figure 2(b) we see that after instruction `1` `x` surely contains 'a'. Inside the loop (line `3`), `x` surely contains 'a', '0' and '1'. In line `4` we report the least upper bound between the value of `x` *before* entering the loop (line `1`) and the value *after* the loop (line `4`): variable `x` surely contains the character 'a', and it also may contain the characters '0' and '1'.

4.2 Prefix and Suffix

The next abstract domain we consider approximates strings by their *prefix*. A string will be a sequence of characters which *begins* with a certain sequence of characters and ends with any string (we use $*$ to represent any string, ϵ included). For example, $abc*$ represents all the strings which begin with "*abc*", including "*abc*" itself. Since the asterisk $*$ at the end of the representation is always present, we do not include it in the domain and consider abstract elements made only of sequence of characters: $\overline{\mathcal{PR}} = \mathsf{K}^* \cup \bot_{\overline{\mathcal{PR}}}$ The partial order on this domain is:

$$\overline{\mathsf{S}} \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{T}} \Leftrightarrow \overline{\mathsf{S}} = \bot_{\overline{\mathcal{PR}}} \vee (\forall \mathsf{i} \in [0, \mathit{len}(\overline{\mathsf{T}}) - 1] : \mathit{len}(\overline{\mathsf{T}}) \leq \mathit{len}(\overline{\mathsf{S}}) \wedge \overline{\mathsf{T}}[\mathsf{i}] = \overline{\mathsf{S}}[\mathsf{i}])$$

An abstract string $\overline{\mathsf{S}}$ is smaller than $\overline{\mathsf{T}}$ if $\overline{\mathsf{T}}$ is a prefix of $\overline{\mathsf{S}}$ or if $\overline{\mathsf{S}}$ is the bottom $\bot_{\overline{\mathcal{PR}}}$ of the domain. The least upper bound operator is defined as the longest common prefix of two strings. The greater lower bound is defined by:

$$\sqcap_{\overline{\mathcal{PR}}}(\overline{\mathsf{S}}_1, \overline{\mathsf{S}}_2) = \begin{cases} \overline{\mathsf{S}}_1 & \text{if } \overline{\mathsf{S}}_1 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{S}}_2 \\ \overline{\mathsf{S}}_2 & \text{if } \overline{\mathsf{S}}_2 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{S}}_1 \\ \bot_{\overline{\mathcal{PR}}} & \text{otherwise} \end{cases}$$

The widening operator is simply the upper bound operator above, as the latter converges in finite time. Top and bottom elements are, respectively, ϵ (the empty prefix) and $\bot_{\overline{\mathcal{PR}}}$. The function which abstracts a single string is $\alpha'_{\overline{\mathcal{PR}}}(\mathsf{s}) = \mathsf{s}$. The abstraction function is $\alpha_{\overline{\mathcal{PR}}}(\mathsf{S}_1) = \bigsqcup_{\overline{\mathcal{PR}}, \mathsf{s} \in \mathsf{S}_1} \alpha'_{\overline{\mathcal{PR}}}(\mathsf{s})$. This means that we consider the longest common prefix amongst all strings in S_1.

We can track information about the *suffix* of a string as well. We define another abstract domain, $\overline{\mathcal{SU}}$, where a string will be something which *ends* with a certain sequence of characters. The notation and all the operators of this domain are dual to those of the previous domain. The definition of the domain is: $\overline{\mathcal{SU}} = \mathsf{K}^* \cup \bot_{\overline{\mathcal{SU}}}$. The partial order is:

$$\begin{aligned} \overline{\mathsf{S}} \leq_{\overline{\mathcal{SU}}} \overline{\mathsf{T}} \Leftrightarrow \overline{\mathsf{S}} = \bot_{\overline{\mathcal{SU}}} \vee (\forall \mathsf{i} \in [0, \mathit{len}(\overline{\mathsf{T}}) - 1] : \mathit{len}(\overline{\mathsf{T}}) \leq \mathit{len}(\overline{\mathsf{S}}) \wedge \\ \overline{\mathsf{T}}[\mathsf{i}] = \overline{\mathsf{S}}[\mathsf{i} + \mathit{len}(\overline{\mathsf{S}}) - \mathit{len}(\overline{\mathsf{T}})]) \end{aligned}$$

The least upper bound $\sqcup_{\overline{\mathcal{SU}}}$ is the longest common suffix, while the greatest lower bound $\sqcap_{\overline{\mathcal{SU}}}$ is the smallest suffix (if they are comparable) or $\bot_{\overline{\mathcal{SU}}}$ (if they are not comparable). The widening operator is the least upper bound operator above. The top element is ϵ. The function which abstracts a single string is: $\alpha'_{\overline{\mathcal{SU}}}(\mathsf{s}) = \mathsf{s}$, and the abstraction function is $\alpha_{\overline{\mathcal{SU}}}(\mathsf{S}_1) = \bigsqcup_{\overline{\mathcal{SU}}, \mathsf{s} \in \mathsf{S}_1} \alpha'_{\overline{\mathcal{SU}}}(\mathsf{s})$.

$$\overline{\mathbb{S}_{\mathcal{PR}}}[\![\texttt{new String(str)}]\!]() = \mathsf{str}$$
$$\overline{\mathbb{S}_{\mathcal{PR}}}[\![\texttt{concat}]\!](\bar{\mathsf{p}}_1, \bar{\mathsf{p}}_2) = \bar{\mathsf{p}}_1$$
$$\overline{\mathbb{S}_{\mathcal{PR}}}[\![\texttt{readLine}]\!]() = \epsilon$$
$$\overline{\mathbb{S}_{\mathcal{PR}}}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\bar{\mathsf{p}}) = \begin{cases} \bar{\mathsf{p}}[\mathsf{b} \cdots \mathsf{e} - 1] & \text{if } \mathsf{e} \leq \mathit{len}(\bar{\mathsf{p}}) \\ \bar{\mathsf{p}}[\mathsf{b} \cdots \mathit{len}(\bar{\mathsf{p}}) - 1] & \text{if } \mathsf{e} > \mathit{len}(\bar{\mathsf{p}}) \wedge \mathsf{b} < \mathit{len}(\bar{\mathsf{p}}) \\ \epsilon & \text{otherwise} \end{cases}$$
$$\overline{\mathbb{B}_{\mathcal{PR}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\bar{\mathsf{p}}) = \begin{cases} \mathsf{true} \text{ if } \mathsf{c} \in \mathit{char}(\bar{\mathsf{p}}) \\ \top_{\mathsf{B}} \;\; \text{otherwise} \end{cases}$$

Table 4. The abstract semantics of $\overline{\mathcal{PR}}$

$$\overline{\mathbb{S}_{\mathcal{SU}}}[\![\texttt{new String(str)}]\!]() = \mathsf{str}$$
$$\overline{\mathbb{S}_{\mathcal{SU}}}[\![\texttt{concat}]\!](\bar{\mathsf{s}}_1, \bar{\mathsf{s}}_2) = \bar{\mathsf{s}}_2$$
$$\overline{\mathbb{S}_{\mathcal{SU}}}[\![\texttt{readLine}]\!]() = \epsilon$$
$$\overline{\mathbb{S}_{\mathcal{SU}}}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\bar{\mathsf{s}}) = \epsilon$$
$$\overline{\mathbb{B}_{\mathcal{SU}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\bar{\mathsf{s}}) =$$
$$= \begin{cases} \mathsf{true} \text{ if } \mathsf{c} \in \mathit{char}(\bar{\mathsf{s}}) \\ \top_{\mathsf{B}} \;\; \text{otherwise} \end{cases}$$

(a) The abstract semantics of $\overline{\mathcal{SU}}$

#I	Var	$\overline{\mathcal{PR}}$	$\overline{\mathcal{SU}}$
1	query	$\overline{\mathsf{s}_1}$	$\overline{\mathsf{s}_1}$
3	l	ϵ	ϵ
3	query	$\overline{\mathsf{s}_1}$	$\overline{\mathsf{s}_3}$
4	query	$\overline{\mathsf{s}_1}$	“ ”
5	per	$\overline{\mathsf{s}_4}$	$\overline{\mathsf{s}_4}$
7	query	$\overline{\mathsf{s}_1}$	$\overline{\mathsf{s}_6}$

(b) First running example

#I	Var	$\overline{\mathcal{PR}}$	$\overline{\mathcal{SU}}$
1	x	a	a
3	x	0	1
4	x	$\top$	$\top$

(c) Second running example

Fig. 3. The abstract semantics of $\overline{\mathcal{SU}}$ and the running examples

These abstract domains could be particularly useful to check if some simple syntactic properties (e.g., a string that is used as an SQL command always begins with “`SELECT`” and ends with “;”) are respected by all possible executions.

Semantics Table 4 and 3(a) define the abstract semantics on $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$ respectively. The most precise suffix and prefix of a constant string are the string itself. When we concatenate two strings, we consider as prefix and suffix of the resulting string the abstract value of the left and right operand respectively. As usual, the semantics of `readLine` returns the top value. The same happens for $\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}$ in $\overline{\mathcal{SU}}$, since we do not know how many characters there are before the suffix. Instead, $\overline{\mathcal{PR}}$ can be more precise if `b` (and eventually `e`) are smaller than the length of the prefix we have. Finally, the semantics of $\texttt{contains}_{\mathsf{c}}$ returns true iff c is in the prefix or suffix, and $\top_{\mathsf{B}}$ otherwise, since we have no information at all about which characters are after the prefix or before the suffix.

Running Example The results of the analyses using the prefix and suffix domains on our running examples are reported in Figures 3(b) and 3(c).

For the first program, at line 1, `query` contains the whole string s_1 as both prefix and suffix. As already pointed out, `l` is an input of the user, so we do not know what its prefix and suffix are. On the other hand, when we concatenate it at line 3, we still have some information on the prefix and suffix of the resulting string. Thus, at the end of the analyses, we get that the prefix of `query` is string s_1, its suffix is s_6, although we lose information about what there is in the middle.

For the second program, before entering the loop we know the prefix and suffix of `x`. Inside the loop after line 3, the convergence for `x` is ‘0’ as prefix and ‘1’ as suffix. This state, combined through the lub operator with the state before the loop, unfortunately goes to $\top$ (the longest common prefixes and suffixes are empty), making us lose all the information.

4.3 Bricks

The next abstract domain, $\overline{\mathcal{BR}}$, captures both *inclusion and order* amongst characters, using a simplification of regular expressions. Therefore, the information tracked by this domain could be adopted to prove more sophisticated properties than the previous domains (e.g., the well-formedness of SQL queries). A string is approximated by a combination of *bricks*. A brick is defined as an element of: $\overline{\mathcal{B}} = [\wp(\mathsf{S})]^{\mathsf{min},\mathsf{max}}$, where min and max are two integer positive values. A brick represents all the strings which can be built through the given strings, taken between min and max times altogether. For example, $[\{\text{“}mo\text{”}, \text{“}de\text{”}\}]^{1,2} = \{mo, de, momo, dede, mode, demo\}$. We represent strings as ordered lists of bricks. For example we have that $[\{\text{“}straw\text{”}\}]^{0,1}[\{\text{“}berry\text{”}\}]^{1,1} = \{berry, strawberry\}$ since $[\{\text{“}straw\text{”}\}]^{0,1}$ concretizes to $\{\epsilon, \text{“}straw\text{”}\}$ and $[\{\text{“}berry\text{”}\}]^{1,1}$ to $\{\text{“}berry\text{”}\}$. Since a particular set of strings could be represented by more than one combination of bricks, we adopted a normalised form in which the lists are made of bricks like $[\mathsf{T}]^{1,1}$ or $[\mathsf{T}]^{0,\mathsf{max}>0}$, where T is a set of strings. We defined a function $\overline{normBricks}(\overline{\mathsf{L}})$ which, given a list of bricks $\overline{\mathsf{L}}$, returns its normalized version.
The abstract domain of bricks is defined as: $\overline{\mathcal{BR}} = \overline{\mathcal{B}}^{*}$, that is, the set of all finite sequences composed of bricks. The top element $\top_{\overline{\mathcal{BR}}}$ is a list containing only $\top_{\overline{\mathcal{B}}}$. The bottom element is $\bot_{\overline{\mathcal{BR}}}$, an empty list or any list which contains at least one invalid element ($\bot_{\overline{\mathcal{B}}}$). The partial order between single bricks is: $[\overline{\mathsf{C}}_1]^{\mathsf{min}_1,\mathsf{max}_1} \leq_{\overline{\mathcal{B}}} [\overline{\mathsf{C}}_2]^{\mathsf{min}_2,\mathsf{max}_2} \Leftrightarrow (\overline{\mathsf{C}}_1 \subseteq \overline{\mathsf{C}}_2 \wedge \mathsf{min}_1 \geq \mathsf{min}_2 \wedge \mathsf{max}_1 \leq \mathsf{max}_2) \vee [\overline{\mathsf{C}}_2]^{\mathsf{min}_2,\mathsf{max}_2} = \top_{\overline{\mathcal{B}}} \vee [\overline{\mathsf{C}}_1]^{\mathsf{min}_1,\mathsf{max}_1} = \bot_{\overline{\mathcal{B}}}$ where $\top_{\overline{\mathcal{B}}}$ and $\bot_{\overline{\mathcal{B}}}$ are special bricks, respectively greater and smaller than any other brick. The partial order between lists of bricks $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ is as follows:

$$\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \Leftrightarrow (\overline{\mathsf{L}}_2 = \top_{\overline{\mathcal{BR}}}) \vee (\overline{\mathsf{L}}_1 = \bot_{\overline{\mathcal{BR}}}) \vee (\forall \mathsf{i} \in [1, \mathsf{n}] : \overline{\mathsf{L}}_1[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_2[\mathsf{i}])$$

where we make $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ have the same size n by adding empty bricks ($[\emptyset]^{0,0}$) at the end of the shorter list. The upper bound operator on a single brick is:

$$\bigsqcup_{\overline{\mathcal{B}}}([\overline{\mathsf{S}}_1]^{\mathsf{m}_1,\mathsf{M}_1}, [\overline{\mathsf{S}}_2]^{\mathsf{m}_2,\mathsf{M}_2}) = [\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2]^{\min(\mathsf{m}_1,\mathsf{m}_2),\max(\mathsf{M}_1,\mathsf{M}_2)}$$

The upper bound operator on lists of bricks (elements of our domain) is as follows: given two lists $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, we make them to have the same size n adding empty bricks to the shorter one. Then: $\bigsqcup_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = \overline{\mathsf{L}}_R[1]\overline{\mathsf{L}}_R[2] \ldots \overline{\mathsf{L}}_R[\mathsf{n}]$ where $\forall \mathsf{i} \in [1, \mathsf{n}] : \overline{\mathsf{L}}_R[\mathsf{i}] = \sqcup_{\overline{\mathcal{B}}}(\overline{\mathsf{L}}_1[\mathsf{i}], \overline{\mathsf{L}}_2[\mathsf{i}])$.
Let k_L, k_I and k_S be three constant integer values. The widening operator $\nabla_{\overline{\mathcal{BR}}} : (\overline{\mathcal{BR}} \times \overline{\mathcal{BR}}) \rightarrow \overline{\mathcal{BR}}$ is defined as follows:

$$\nabla_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = \begin{cases} \top_{\overline{\mathcal{BR}}} & \text{if } (\overline{\mathsf{L}}_1 \not\leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \wedge \overline{\mathsf{L}}_2 \not\leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1) \vee \\ & \quad (\exists \mathsf{i} \in [1, 2] : len(\overline{\mathsf{L}}_i) > \mathsf{k}_L) \\ w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) & \text{otherwise} \end{cases}$$

where $w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = [\overline{\mathcal{B}}_1^{\mathsf{new}}(\overline{\mathsf{L}}_1[1], \overline{\mathsf{L}}_2[1]); \overline{\mathcal{B}}_2^{\mathsf{new}}(\overline{\mathsf{L}}_1[2], \overline{\mathsf{L}}_2[2]); \ldots; \overline{\mathcal{B}}_n^{\mathsf{new}}(\overline{\mathsf{L}}_1[\mathsf{n}], \overline{\mathsf{L}}_2[\mathsf{n}])]$, with n being the size of the bigger list (we make them to have the same size n adding empty bricks to the shorter one), and $\overline{\mathcal{B}}_i^{\mathsf{new}}(\overline{\mathsf{L}}_1[\mathsf{i}], \overline{\mathsf{L}}_2[\mathsf{i}])$ is defined by:

$$\begin{array}{l}
\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{new String}(\texttt{str})]\!]() = [\{\texttt{str}\}]^{1,1} \\
\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{concat}]\!](\overline{\mathsf{b}}_1, \overline{\mathsf{b}}_2) = \overline{\mathit{normBricks}}(\overline{\mathit{concatList}}(\overline{\mathsf{b}}_1, \overline{\mathsf{b}}_2)) \\
\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{readLine}]\!]() = \top_{\overline{\mathcal{BR}}} \\
\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\overline{\mathsf{b}}) = \begin{cases} [\overline{\mathsf{T}}']^{1,1} \text{ if } \overline{\mathsf{b}}[0] = [\overline{\mathsf{T}}]^{1,1} \wedge \forall \overline{\mathsf{t}} \in \overline{\mathsf{T}} : \mathit{len}(\overline{\mathsf{t}}) \geq \mathsf{e} \\ \top_{\overline{\mathcal{BR}}} \quad \text{otherwise} \end{cases} \\
\overline{\mathbb{B}_{\mathcal{BR}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\overline{\mathsf{b}}) = \begin{cases} \mathsf{true} \text{ if } \exists \overline{\mathsf{B}} \in \overline{\mathsf{b}} : \overline{\mathsf{B}} = [\overline{\mathsf{T}}]^{m,M} \wedge 1 \leq m \leq M \wedge (\forall \overline{\mathsf{t}} \in \overline{\mathsf{T}} : \mathsf{c} \in \mathit{char}(\overline{\mathsf{t}})) \\ \mathsf{false} \text{ if } \forall [\overline{\mathsf{T}}]^{\mathsf{m},\mathsf{M}} \in \overline{\mathsf{b}}, \forall \overline{\mathsf{t}} \in \overline{\mathsf{T}} : \mathsf{c} \notin \mathit{char}(\overline{\mathsf{t}}) \\ \top_{\mathsf{B}} \quad \text{otherwise} \end{cases}
\end{array}$$

Table 5. The abstract semantics of $\overline{\mathcal{BR}}$

$$\overline{\mathcal{B}}_i^{\mathsf{new}}([\overline{\mathsf{S}}_{1i}]^{\mathsf{m}_{1i},\mathsf{M}_{1i}}, [\overline{\mathsf{S}}_{2i}]^{\mathsf{m}_{2i},\mathsf{M}_{2i}}) = \begin{cases} \top_{\overline{\mathcal{B}}} & \text{if } |\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}| > \mathsf{k}_S \\ & \quad \vee \overline{\mathsf{L}}_1[\mathsf{i}] = \top_{\overline{\mathcal{B}}} \vee \overline{\mathsf{L}}_2[\mathsf{i}] = \top_{\overline{\mathcal{B}}} \\ [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(0,\infty)} & \text{if } (\mathsf{M} - \mathsf{m}) > \mathsf{k}_I \\ [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(\mathsf{m},\mathsf{M})} & \text{otherwise} \end{cases}$$

where $\mathsf{m} = \min(\mathsf{m}_{1i}, \mathsf{m}_{2i})$ and $\mathsf{M} = \max(\mathsf{M}_{1i}, \mathsf{M}_{2i})$. $\nabla_{\overline{\mathcal{BR}}}$ is an upper bound operator because it returns either $\top_{\overline{\mathcal{BR}}}$ or $w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$, which builds a new list of bricks which is bigger (with respect to $\leq_{\overline{\mathcal{BR}}}$) than both $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$. The resulting list is greater or equal because each brick is greater than or equal to the two corresponding bricks in $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, since we always take the union of the two strings sets and an index range bigger than the initial two. Moreover, this operator converges because a value of an ascending chain can increase along three axes: (i) the length of the brick list, (ii) the indices range of a certain brick, and (iii) the strings contained in a certain brick. The growth of an abstract value is bounded along each axis with the help of the three constants. After the list has reached k_L elements, the entire abstract value is approximated to $\top_{\overline{\mathcal{BR}}}$. If the range of a certain brick becomes larger than k_I, the range is approximated to $(0, +\infty)$. Finally, if the strings set of a certain brick reaches k_S elements, the brick is approximated to $\top_{\overline{\mathcal{B}}}$. The lower bound operator is dual with respect to the upper bound operator above. Formally, $\sqcap_{\overline{\mathcal{B}}}([\overline{\mathsf{S}}_1]^{\mathsf{m}_1,\mathsf{M}_1}, [\overline{\mathsf{S}}_2]^{\mathsf{m}_2,\mathsf{M}_2}) = [\overline{\mathsf{S}}_1 \cap \overline{\mathsf{S}}_2]^{\max(\mathsf{m}_1,\mathsf{m}_2),\min(\mathsf{M}_1,\mathsf{M}_2)}$. The abstraction function is defined by: $\alpha'_{\overline{\mathcal{BR}}}(\mathsf{s}) = [\{\mathsf{s}\}]^{(1,1)}$ and

$$\alpha_{\overline{\mathcal{BR}}}(\mathsf{S}_1) = \bigsqcup_{\overline{\mathcal{BR}}, \mathsf{s} \in \mathsf{S}_1} \alpha'_{\overline{\mathcal{BR}}}(\mathsf{s}) = [\mathsf{S}_1]^{(1,1)}$$

Semantics Table 5 defines the abstract semantics on $\overline{\mathcal{BR}}$. When a constant string is evaluated, the semantics returns a single brick containing exactly that string with $[1, 1]$ as index. For the concatenation of two strings, we rely on the $\overline{\mathit{concatList}}$ function that concatenates two lists of bricks, and then we normalise its result. `readLine` returns the top value, while $\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}$ returns the substring iff the first brick of the list has index [1, 1] and the length of all the strings contained in it is greater than e. Notice that $\overline{\mathsf{T}}' = \{\overline{\mathsf{t}}.\texttt{substring}(\mathsf{b}, \mathsf{e}) \forall \overline{\mathsf{t}} \in \overline{\mathsf{T}}\}$. Finally, the semantics of $\texttt{contains}_{\mathsf{c}}$ returns true iff there is surely at least one brick that contains `c` and whose minimal index is at least 1. It returns false iff all the bricks do not contain `c`, and $\top_{\mathsf{B}}$ otherwise.

#I	Var	$\overline{\mathcal{BR}}$
1	query	$[\{\mathsf{s}_1\}]^{1,1}$
3	l	$\top_{\overline{\mathcal{B}}}$
3	query	$[\{\mathsf{s}_1 + \mathsf{s}_2\}]^{1,1}\top_{\overline{\mathcal{B}}}[\{\mathsf{s}_3\}]^{1,1}$
4	query	$[\{\mathsf{s}_1, \mathsf{s}_1 + \mathsf{s}_2\}]^{1,1}\top_{\overline{\mathcal{B}}}[\{\mathsf{s}_3\}]^{0,1}$
5	per	$[\{\mathsf{s}_4\}]^{1,1}$
7	query	$[\{\mathsf{s}_1, \mathsf{s}_1 + \mathsf{s}_2\}]^{1,1}\top_{\overline{\mathcal{B}}}[\{\mathsf{s}_3\}]^{0,1}$ $[\{\mathsf{s}_5 + \mathsf{s}_4 + \mathsf{s}_6\}]^{1,1}$

(a) First running example

#I	Var	$\overline{\mathcal{BR}}$
1	x	$[\{“a”\}]^{1,1}$
3	x	$\top$
4	x	$\top$

(b) Second running example

Fig. 4. The results of $\overline{\mathcal{BR}}$

Running Example The results of the analysis of the running examples using $\overline{\mathcal{BR}}$ are depicted in Figures 4(a) and 4(b). For the first program, the bricks of the final result on query are four: (i) the first brick represents a string between s_1 and $\mathsf{s}_1 + \mathsf{s}_2$, (ii) the second brick corresponds to the input l, (iii) the third brick could be the empty string ϵ or s_3, and (iv) the fourth brick represents the concatenation of s_5, s_4, and s_6. We can see that the precision is higher than in the previous domains, but still not the best we aim to get: amongst the concrete results we have, for example, $\mathsf{s}_1 + \mathsf{s}_3 + \mathsf{s}_5 + \mathsf{s}_4 + \mathsf{s}_6$, which cannot be computed in any execution of the analysed code. For the second program, the result is unsatisfactory: the use of the widening operator makes us lose all information. At the end of the program, variable x has value $\top$.

4.4 String Graphs

The last abstract domain we introduce exploits type graphs, a data structure which represents tree automata [15], adapting them to represent sets of strings. A type graph $\overline{\mathsf{T}}$ is a triplet $(\overline{\mathsf{N}}, \overline{\mathsf{A}}_F, \overline{\mathsf{A}}_B)$ where $(\overline{\mathsf{N}}, \overline{\mathsf{A}}_F)$ is a rooted tree whose arcs in $\overline{\mathsf{A}}_F$ are called forward arcs, and $\overline{\mathsf{A}}_B$ is a restricted class of arcs, backward arcs, superimposed on $(\overline{\mathsf{N}}, \overline{\mathsf{A}}_F)$. Each node $\overline{\mathsf{n}} \in \overline{\mathsf{N}}$ of a type graph has a label, denoted by $\overline{lb}(\overline{\mathsf{n}})$, indicating the kind of term it describes, and the nodes are divided into three classes: simple, functor and OR nodes. We use the convention that $\overline{\mathsf{n}}/\mathsf{i}$ denotes the i-th son of node $\overline{\mathsf{n}}$, and the set of sons of a node $\overline{\mathsf{n}}$ is then denoted as $\{\overline{\mathsf{n}}/1, \dots, \overline{\mathsf{n}}/k\}$ with $\overline{\mathsf{k}} = \overline{outdegree}(\overline{\mathsf{n}})$ where $\overline{outdegree}$ is a function that given a node returns the number of its sons. We define a modified version of type graphs, called string graphs, which represent strings instead of types. String graphs have the same basic structure of type graphs. The following differences distinguish them: (i) simple nodes have labels from the set $\{\mathsf{max}, \bot, \epsilon\} \cup \mathsf{K}$; (ii) the only functor we consider is concat (with its obvious meaning of string concatenation). Thus, functor nodes are labelled with $\mathsf{concat/k}$. An example is depicted in Figure 5. The root of the string graph is an OR node with two sons: a simple node (b) and a concat node with two sons of its own. The second son of the concat node is the root (with the use of a backward arc). Such string graph represents the following set of strings: $\{b, ab, aab, aaab, \dots\} = a^*b$.

The abstract domain is: $\overline{\mathcal{SG}} = \overline{\mathsf{NSG}}$, where $\overline{\mathsf{NSG}}$ is the set of all Normal String Graphs. In fact, the type graphs are very suitable for representing a set of terms. However, several distinct type graphs can have the same denotation. The existence of superfluous nodes and arcs makes operations needed during abstract interpretation, such as the $\leq$-operation, quite complex and inefficient. In order to reduce this variety of type graphs, additional restrictions are imposed (for details see [15]), defining normal type graphs. We added a few other restrictions (specific for string graphs), thus obtaining the definition of normal string graphs. For example, we impose that concat nodes are not allowed to have only one son (they should be replaced by the son itself) or that a $\mathtt{concat}$ node cannot have two successive sons with both label concat (they should be merged together). An algorithm of normalisation ($\overline{\mathit{normStringGraph}}$), encapsulating all those rules, is defined as well.

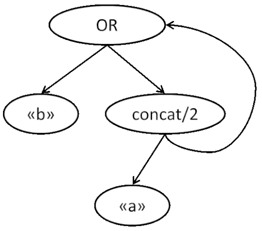

Fig. 5. An example of string graph

The bottom element $\bot_{\overline{\mathcal{SG}}}$ is a string graph made by one bottom node. The top element is a string graph made by only one node, a max-node. To define the partial order of the domain we can exploit an algorithm defined in [15]: $\leq(\overline{\mathsf{n}}, \overline{\mathsf{m}}, \emptyset)$. The algorithm compares the two nodes in input $(\overline{\mathsf{n}}, \overline{\mathsf{m}})$. In some cases the procedure is recursively called, for example if $\overline{\mathsf{n}}$ and $\overline{\mathsf{m}}$ are both concat or OR nodes. Note that the recursive call adds a new edge $(\{\overline{\mathsf{n}}, \overline{\mathsf{m}}\})$ to the third input parameter (a set of edges). If, at the next execution of the procedure $(\leq(\overline{\mathsf{n}'}, \overline{\mathsf{m}}', \overline{\mathsf{E}}))$, the edge $\{\overline{\mathsf{n}'}, \overline{\mathsf{m}'}\}$ is contained in $\overline{\mathsf{E}}$ then the procedure immediately returns true. The order is then:

$$\overline{\mathsf{T}}_1 \leq_{\overline{\mathcal{SG}}} \overline{\mathsf{T}}_2 \Leftrightarrow \overline{\mathsf{T}}_1 = \bot_{\overline{\mathcal{SG}}} \vee (\leq(\overline{\mathsf{n}}_0, \overline{\mathsf{m}}_0, \emptyset) : \overline{\mathsf{n}}_0 = \overline{\mathit{root}}(\overline{\mathsf{T}}_1) \wedge \overline{\mathsf{m}}_0 = \overline{\mathit{root}}(\overline{\mathsf{T}}_2))$$

where $\overline{\mathit{root}}(\overline{\mathsf{T}})$ is the root element of the tree defined in $\overline{\mathsf{T}}$. The least upper bound between two string graphs $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$ can be computed creating a new string graph $\overline{\mathsf{T}}$ whose root is an OR-node and whose sons are $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$. Then we apply the compaction algorithm that will transform $\overline{\mathsf{T}}$ in a normal string graph:

$$\bigsqcup_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2) = \overline{\mathit{normStringGraph}}(\mathsf{OR}(\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2))$$

The greatest lower bound operator is described in the appendix of [15], while the widening operator is described in [23]. The abstraction of a string is: $\alpha'_{\overline{\mathcal{SG}}}(\mathsf{s}) = \mathsf{concat/k}\{\mathsf{s[i]} : \mathsf{i} \in [0, \mathsf{k}-1]\}$ where $\mathsf{k} = \mathit{len}(\mathsf{s})$, and the abstraction function is:

$$\alpha_{\overline{\mathcal{SG}}}(\mathsf{S}_1) = \bigsqcup_{\overline{\mathcal{SG}}, \mathsf{s} \in \mathsf{S}_1} \alpha'_{\overline{\mathcal{SG}}}(\mathsf{s}) = \overline{\mathit{normStringGraph}}(\mathsf{OR}\{\alpha'_{\overline{\mathcal{SG}}}(\mathsf{s}) : \mathsf{s} \in \mathsf{S}_1\})$$

Semantics Table 6 defines the abstract semantics on $\overline{\mathcal{SG}}$. The evaluation of a string returns a concat containing the sequence of all the characters of the string. When we concatenate two strings, the semantics returns the normalisation of a concat node containing the two strings in sequence. As usual, the semantics of `readLine` returns the top value. The semantics of $\mathtt{substring}_{\mathsf{b}}^{\mathsf{e}}$ (where $\overline{\mathsf{res}} = \mathsf{concat}/(\mathsf{e}-\mathsf{b})\{(\overline{\mathit{root}}(\overline{\mathsf{t}})/\mathsf{i}) : \mathsf{i} \in [\mathsf{b}, \mathsf{e}-1]\}$) returns a precise value only if the root is a `concat` node with at least `e` characters. Finally, $\mathtt{contains}_{\mathsf{c}}$ returns true iff there is a concat node containing `c` in the tree, and without any OR node in the path from the root to this node.

$$
\begin{array}{l}
\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{new String(str)}]\!]() = \mathsf{concat}/\mathsf{k}\{\texttt{str}[\mathsf{i}] : \mathsf{i} \in [0, \mathsf{k}-1]\} \\
\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{concat}]\!](\bar{\mathsf{t}}_1, \bar{\mathsf{t}}_2) = \overline{\mathit{normStringGraph}}(\mathsf{concat}/2\{\bar{\mathsf{t}}_1, \bar{\mathsf{t}}_2\}) \\
\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{readLine}]\!]() = \top_{\overline{\mathcal{SG}}} \\
\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\bar{\mathsf{t}}) = \begin{cases} \overline{\mathsf{res}} & \text{if } \overline{\mathit{root}}(\bar{\mathsf{t}}) = \mathsf{concat}/\mathsf{k} \wedge \forall \mathsf{i} \in [0, \mathsf{e}-1] : \overline{\mathit{lb}}(\overline{\mathit{root}}(\bar{\mathsf{t}})/\mathsf{i}) \in \mathsf{K} \\ \top_{\overline{\mathcal{SG}}} & \text{otherwise} \end{cases} \\
\overline{\mathbb{B}_{\mathcal{SG}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\bar{\mathsf{t}}) = \begin{cases} \mathsf{true} \text{ if } \exists \overline{\mathsf{m}} \in \bar{\mathsf{t}} : \overline{\mathsf{m}} = \mathsf{concat}/\mathsf{k} \wedge \mathsf{OR} \notin \overline{\mathit{path}}(\overline{\mathsf{root}}, \overline{\mathsf{m}}) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \exists \mathsf{i} : \overline{\mathit{lb}}(\overline{\mathsf{m}}/\mathsf{i}) = \mathsf{c} \\ \mathsf{false} \text{ if } \nexists \overline{\mathsf{n}} \in \bar{\mathsf{t}} : \overline{\mathit{lb}}(\overline{\mathsf{n}}) = \mathsf{max} \vee \overline{\mathit{lb}}(\overline{\mathsf{n}}) = \mathsf{c} \\ \top_{\mathsf{B}} \quad \text{otherwise} \end{cases}
\end{array}
$$

Table 6. The abstract semantics of $\overline{\mathcal{SG}}$

#I	Var	$\overline{\mathcal{SG}}$
1	query	$\mathsf{concat}[\mathsf{s}_1]$
3	l	max
3	query	$\mathsf{concat}[\mathsf{s}_1 + \mathsf{s}_2; \mathsf{max}; \mathsf{s}_3]$
4	query	$\overline{\mathsf{SG}}_1 = \mathsf{OR}[\mathsf{concat}[\mathsf{s}_1];$ $\mathsf{concat}[\mathsf{s}_1 + \mathsf{s}_2; \mathsf{max}; \mathsf{s}_3]]$
5	per	$\mathsf{concat}[\mathsf{s}_4]$
7	query	$\mathsf{concat}[\overline{\mathsf{SG}}_1;$ $\mathsf{concat}[\mathsf{s}_5 + \mathsf{s}_4 + \mathsf{s}_6]]$

(a) First running example

#I	Var	$\overline{\mathcal{SG}}$
1	x	$\mathsf{concat}[``a"]$
3	x	$\mathsf{OR}_1[``a"; \mathsf{concat}[``0"; OR_1; ``1"]]$
4	x	$\mathsf{OR}_1[``a"; \mathsf{concat}[``0"; OR_1; ``1"]]$

(b) Second running example

Fig. 6. The results of $\overline{\mathcal{SG}}$

Running Example The results of the analysis of the running examples through string graphs are depicted in Figures 6(a) and 6(b). For sake of simplicity, we adopt the notation $\mathsf{concat}[\mathsf{s}]$ to indicate a string graph with a concat node whose sons are all the characters of string s. The symbol + represents, as usual, string concatenation, while ; is used to separate different sons of a node.
For the first program, the resulting string graph for `query` represents exactly the two possible outcomes of the procedure. For the second program, the resulting string graph for `x` represents exactly all the concrete possible values of `x`. Note that the resulting string graph contains a backward arc to allow the repetition of the pattern $0^n \dots 1^n$. This abstract domain is the most precise domain for the analysis of both running examples: it tracks information similarly to $\overline{\mathcal{BR}}$ domain, but its lub and widening operators are definitely more accurate.

4.5 Discussion: Relations Between the Four Domains

The abstract domains we introduced in the previous sections track different types of information. Let us discuss the relations between different domains. Intuitively, there are two axes on which the analyses of string values can work: the characters contained in a string, and their position inside the string. It is easy to see that the

$\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$ are less precise than $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$. In fact, $\overline{\mathcal{CI}}$ domain considers only character inclusion and completely disregards the order. $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$ domains consider also the order, but limiting themselves to the initial/final segment of the string, and in the same way they collect only partial information about character inclusion. $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$, instead, track both inclusion and order along the string. In [3] we studied these relationships in details: we defined pairs of functions (abstraction and concretization) from domain to domain, and showed that $\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$ are more abstract (i.e., less precise) than both $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$. In the case of $\overline{\mathcal{BR}}$ versus $\overline{\mathcal{SG}}$, the comparison is more complex, since they exploit very different data structures. For example, $\overline{\mathcal{SG}}$ has OR-nodes, while $\overline{\mathcal{BR}}$ can only trace alternatives inside bricks but not outside (like: "these three bricks *or* these other two"). From this perspective, $\overline{\mathcal{SG}}$ is more precise than $\overline{\mathcal{BR}}$. Another important difference is that $\overline{\mathcal{SG}}$ has backward arcs which allow repetitions of patterns, but they can be traversed how many times we want (even infinite times). With $\overline{\mathcal{BR}}$, instead, we can indicate exactly how many times a certain pattern should be repeated (through the range of bricks). This makes $\overline{\mathcal{BR}}$ more expressive than $\overline{\mathcal{SG}}$ in that respect. So, these domains are not directly comparable. We obtain the lattice depicted in Figure 7, where the upper domains are more approximated. We denote by $\top$ the abstract domain that does not track any information about string values, and by $\wp(\mathsf{K}^*)$ the (naïve and uncomputable) domain that tracks all the possible strings values we can have.

In conclusion, the first three domains ($\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$, $\overline{\mathcal{SU}}$) are not so precise but the complexity is kept linear, whereas the other domains ($\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$) are more demanding (though in the practice complexity is still kept polynomial) but also more precise.

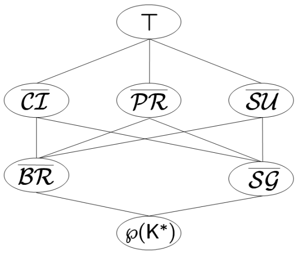
⊤
$\overline{\mathcal{CI}}$
$\overline{\mathcal{PR}}$
$\overline{\mathcal{SU}}$
$\overline{\mathcal{BR}}$
$\overline{\mathcal{SG}}$
$\wp(\mathsf{K}^*)$

Fig. 7. The hierarchy of abstract domains

5 Related Work

The static analysis of strings was addressed in various directions.

Kim and Choe [16] introduced recently an approach based on abstract interpretation. They abstract strings with pushdown automata (PDA). The result of the analysis is compared with a grammar to determine if all the strings generated by the PDA belong to the grammar. This approach has a fixed precision, and in the worst case (not often encountered in practice) it has exponential complexity.

Hosoya and Pierce [14] used tree automata to verify dynamically generated XML documents. The regular expression types of this approach recall our $\overline{\mathcal{BR}}$ domain, while the tree automata recall our $\overline{\mathcal{SG}}$ domain. However, they are focused on building XML documents, while our focus is on collecting possible values of generic string variables. In addition, they require to manually annotate the code through types while our approach is completely automatic.

A more recent work was developed by Yu *et al.* [24]. It presented an automata-based approach for the verification of string operations in PHP programs. The information tracked by this analysis is fixed, and it is specific for PHP programs.

Tabuchi *et al.* [21] presented a type system based on regular expressions. It is focused on a λ-calculus supporting concatenation, and pattern matching. Some type annotation is required when dealing with recursive function.

Thiemann [22] introduced a type system for string analysis based on context-free grammars. Their analysis is more precise than those based on regular expressions, but the only supported string operator is concatenation, and the analysis is tuned at a fixed level of precision.

Context-free grammars are also the basis of the analysis of Christensen *et al.* [1]. This analysis is tuned at a fixed level of abstraction. In the second running example of this paper, $\overline{\mathcal{SG}}$ domain reaches a better precision than theirs.

Minamide [18] presented an analysis to statically check some properties of Web pages generated dynamically by a server-side program. This work is specific for HTML pages, while we do not need to know the reference grammar *a priori*. Also in this case, $\overline{\mathcal{SG}}$ obtain a better precision on the loop example.

Doh *et al.* [8] proposed a technique called "abstract parsing": it combines LR(k)-parsing technology and data-flow analysis to analyse dynamically generated documents. Their technique is quite precise, but the level of abstraction is fixed, and it cannot be tuned at different levels of precision and efficiency.

Given this context, our work is the first one that (i) is a generic, flexible, and extensible approach to the analysis of string values, and (ii) can be tuned at different levels of precision and efficiency.

6 Conclusion and Future Work

In this paper we introduced a new framework for the static analysis of string values, and four different abstract domains. We chose some string operators on which we focused our approach defining the concrete and the abstract semantics. **Future work** We are working on the implementation of our approach in Sample (Static Analyzer of Multiple Programming LanguagEs) [9]. We plan to apply our analysis to some case studies to study the precision of our analysis. In order to check the scalability and performances of our approach, we plan to apply our analysis to some Scala standard libraries. Some preliminary experimental results point out that $\overline{\mathcal{CI}}$ and $\overline{\mathcal{PR}} \times \overline{\mathcal{SU}}$ are quite efficient, $\overline{\mathcal{BR}}$ is slower but still fast, while $\overline{\mathcal{SG}}$'s performances seem to be still critical.

Acknowledgments. Work partially supported by RAS project "TESLA - Tecniche di enforcement per la sicurezza dei linguaggi e delle applicazioni", and by SNF project "Verification-Driven Inference of Contracts".

References

1. A. Christensen, A. Moller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of SAS '03*, pages 1–18. Springer-Verlag, 2003.

2. A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. In *Computer Languages, Systems and Structures*, volume 37(1), pages 24–42. Elsevier, 2011.
3. G. Costantini. Abstract domains for static analysis of strings. Master's thesis, Ca' Foscari University of Venice, 2010.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*. ACM, 1977.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79*. ACM, 1979.
6. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *Proceedings of ESOP '05*, LNCS. Springer-Verlag, 2005.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of POPL '78*. ACM Press, 1978.
8. K. Doh, H. Kim, and D. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *Proceedings of SAS '09*, pages 256–272. Springer-Verlag, 2009.
9. P. Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Proceedings of FORTE/FMOODS '10*, LNCS. Springer, 2010.
10. C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of ICSE '04*, pages 645–654. IEEE Computer Society, 2004.
11. P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proceedings TAPSOFT '91*, LNCS. Springer-Verlag, 1991.
12. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of POPL '11*. ACM, 2011.
13. P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *Proceedings of VMCAI '11*. Springer Verlag, 2011.
14. H. Hosoya and B. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003.
15. G. Janssens and M. Bruynooghe. Deriving description of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.
16. S.-W. Kim and K.-M. Choe. String analysis as an abstract interpretation. In *Proceedings of VMCAI '11*. Springer Verlag, 2011.
17. F. Logozzo and M. Fähndrich. Pentagons: A weakly relational domain for the efficient validation of array accesses. In *Proceedings of SAC '08*. ACM Press, 2008.
18. Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of WWW '05*, pages 432–441. ACM, 2005.
19. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006.
20. R.Halder and A.Cortesi. Obfuscation-based analysis of sql injection attacks. In IEEE, editor, *Proceedings of ISCC 2010*, 2010.
21. N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language. *Electr. Notes Theor. Comput. Sci.*, 75, 2002.
22. P. Thiemann. Grammar-based analysis of string expressions. In *Proceedings of TLDI '05*, pages 59–70. ACM, 2005.
23. P. van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179–208, 1995.
24. F. Yu, T. Bultan, M. Cova, and O. Ibarra. Symbolic string verification: An automata-based approach. In *Proceedings of SPIN '08*, 2008.