SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–41
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

A Suite of Abstract Domains for Static Analysis of String Values

G. Costantini [1] * P. Ferrara [2] and A. Cortesi [1]

[1] *University Ca' Foscari of Venice*
[2] *ETH Zurich*

SUMMARY

Strings are widely used in modern programming languages in various scenarios. For instance, strings are used to build up SQL queries that are then executed. Malformed strings may lead to subtle bugs, as well as non-sanitized strings may rise security issues in an application. For these reasons, the application of static analysis to compute safety properties over string values at compile time is particularly appealing. In this article we propose a generic approach for the static analysis of string values based on abstract interpretation. In particular, we design a suite of abstract semantics for strings, where each abstract domain tracks a different kind of information. We discuss the tradeoff between efficiency and accuracy when using such domains to catch the properties of interest. In this way, the analysis can be tuned at different levels of precision and efficiency, and it can address specific properties. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: static analysis; abstract interpretation; abstract domains; strings

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 Java they are widely used to build up SQL queries, or to access information about the classes through reflection. The properties of interest over string values are extremely wide. For instance, 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`. 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 run-time errors, but to dramatic and permanent effects too [21].

The interest on approaches that automatically analyze and discover bugs on strings is constantly raising. 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 [22, 23, 38, 41], while many other approaches are focused on particular properties or classes of programs [1, 18, 20, 32, 34, 35]. As genericity and scalability are the main advantages of the abstract interpretation approach [8, 9] (since it allows to define analyses at different levels of precision and efficiency), in this article we investigate abstract interpretation as an alternative approach to string analysis.

The main contribution of this article is the formalization of a *unifying* generic abstract interpretation based framework for string analysis, and its instantiations with *five different domains*

*Correspondence to: via Torino 172, 30172, Mestre (Venice), Italy. E-mail: costantini@dsi.unive.it

```
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, `prog1`

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

(b) The second running example, `prog2`

Figure 1. The running examples

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.

The methodology is inspired by the approach adopted for numerical domains for static analysis of software [13, 19, 33]. The interface of a numerical domain is nowadays standard: each domain has to define the semantics of arithmetic expressions and Boolean conditions. Similarly, we consider a limited set of basic string operators supported by all the mainstream programming languages. The concrete semantics of these operators is approximated in different ways by the five 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 [13]) gives up in terms of efficiency, while to achieve scalability we need specific approximations on a given property (e.g., Pentagons [31]) or class of programs (e.g., ASTRÉE [12]). 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 analyzed class of programs and property. Other abstractions are possible and welcomed, and we expect our framework to be generic enough to support them.

This article [†] is structured as follows. In the rest of this Section we introduce two running examples, and we recall some basic concepts of abstract interpretation. Section 2 defines the syntax of the string operators we will consider in the rest of the article. Section 3 introduces their concrete semantics, while in Section 4 the five abstract domains, the core of this work, are formalized and used to analyze the running examples. In Section 5 more experimental results are presented. Finally, Section 6 discusses the related work, and Section 7 concludes.

1.1. Running Examples

Throughout all the article, we will always refer to the two examples reported in Figures 1(a) and 1(b).

The first Java program, prog1, is taken from [18], and it dynamically builds a SQL query by concatenating several strings. One of these concatenations applies only if a given input 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 always well formed. For the sake of readability, we will use some shortcuts to identify the string constants of this program, as reported in Table I.

[†]The article is a fully revised and extended version of [7]

Table I. Shortcuts of string constants in `prog1`

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	"); "

The second program, `prog2`, modifies a string inside a `while` loop whose condition cannot be statically evaluated. Intuitively, this program produces strings of 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 [8, 9], 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 a 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 concrete 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 $\langle \overline{\mathsf{A}}, \leq_{\overline{\mathsf{A}}}, \perp_{\overline{\mathsf{A}}}, \top_{\overline{\mathsf{A}}}, \sqcup_{\overline{\mathsf{A}}}, \sqcap_{\overline{\mathsf{A}}} \rangle$. The concrete and abstract domains are related by a concretization $\gamma_{\overline{\mathsf{A}}}$ and an abstraction $\alpha_{\overline{\mathsf{A}}}$ functions, and, in order to obtain a sound analysis, these have to form a Galois connection. Formally, $\langle \wp(\mathsf{D}), \subseteq \rangle \underset{\alpha_{\overline{\mathsf{A}}}}{\overset{\gamma_{\overline{\mathsf{A}}}}{\longleftrightarrow}} \langle \overline{\mathsf{A}}, \leq_{\overline{\mathsf{A}}} \rangle$. One function univocally identifies the other, and in the rest of the paper we will focus on concretization-based Galois connection, and in particular on the following theorem (Proposition 7 of [10]).

Theorem 1.1 (Concretization-based Galois connection)
Let the concretization function $\gamma_{\overline{\mathsf{A}}} : \overline{\mathsf{A}} \to \wp(\mathsf{D})$ be a complete meet preserving map. Define the abstraction function by $\alpha_{\overline{\mathsf{A}}} = \lambda \mathsf{Y}. \sqcap_{\overline{\mathsf{A}}} \{\bar{\mathsf{z}} : \gamma_{\overline{\mathsf{A}}}(\bar{\mathsf{z}}) \subseteq \mathsf{Y}\}$.
If $\alpha_{\overline{\mathsf{A}}}$ is well-defined then $\langle \wp(\mathsf{D}), \subseteq \rangle \underset{\alpha_{\overline{\mathsf{A}}}}{\overset{\gamma_{\overline{\mathsf{A}}}}{\longleftrightarrow}} \langle \overline{\mathsf{A}}, \leq_{\overline{\mathsf{A}}} \rangle$.

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 $\bar{\mathsf{a}}_0 \leq_{\overline{\mathsf{A}}} \dots \bar{\mathsf{a}}_n \leq_{\overline{\mathsf{A}}} \dots$ the increasing chain defined as $\overline{\mathsf{w}}_0 = \bar{\mathsf{a}}_0, \dots, \overline{\mathsf{w}}_{i+1} = \overline{\mathsf{w}}_i \nabla_{\mathsf{A}} \bar{\mathsf{a}}_{i+1}$ converges after a finite number of steps [5].

An abstract semantics $\overline{\mathbb{S}}$ is a sound approximation of the concrete one if $\forall \bar{\mathsf{a}} \in \overline{\mathsf{A}} : \gamma_{\overline{\mathsf{A}}}(\overline{\mathbb{S}}[\![\bar{\mathsf{a}}]\!]) \supseteq \mathbb{S}[\![\gamma_{\overline{\mathsf{A}}}(\bar{\mathsf{a}})]\!]$.

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 and 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 focus on a small but representative set of common operators. We chose these operators analyzing some case studies, and they are supported by all the mainstream programming languages. Other operators

Table II. String operators in Java , C# and PHP

Operator	Java	C#	PHP
`new String(str)`	new String(str) or “str”	new String(str) or “str”	“str”
`concat(s1, s2)`	s1.concat(s2) or s1 + s2	string.concat(s1,s2) or s1+s2	s1 . s2
$\texttt{substring}_{\texttt{b}}^{\texttt{e}}(\texttt{s})$	s.substring(b, e)	s.substring(b, e)	substr(s, b, e-b)
$\texttt{contains}_{\texttt{c}}(\texttt{s})$	s.contains(c)	s.contains(c)	preg _ match(c, s)

can be easily added to our semantics. For each operator, this would mean to define its concrete semantics, and its approximations on the different domains we will introduce.

Common operations or tests made by programs on string values are the following:

- `new String(str)` (where `str` is a sequence of characters) creates a new constant string;
- `concat(s1, s2)` (where `s1` and `s2` are strings) concatenates two strings. Note that the concatenation operation can also be written with the $+$ operator: `concat(s1, s2)` is the same as $\texttt{s1} + \texttt{s2}$.
- $\texttt{substring}_{\texttt{b}}^{\texttt{e}}(\texttt{s})$ (where `s` is a string, and `b` and `e` are integer values representing the first and last index to use for the substring creation) extracts a substring from a given string;
- $\texttt{contains}_{\texttt{c}}(\texttt{s})$ (where `s` is a string and `c` is a character) checks if a string contains a specific character.

Another common operation is the reading of some input from the user with the `readLine()` statement, but we do not include this operator because its abstract semantics is the same in any abstract domain we could define, i.e. it simply returns the top element $\top$ of the considered domain. Also, note that here we considered the operator $\texttt{contains}_{\texttt{c}}$ which checks if a certain *character* is contained in a string, but in [6] we presented the semantics of the extended version of this operator, i.e., $\texttt{contains}_{\texttt{seq}}$ which checks if a certain *sequence of characters* `seq` is contained in a string. In [6] we also presented two additional operators, i.e. $\texttt{indexOf}_{\texttt{c}}$ and $\texttt{lastIndexOf}_{\texttt{c}}$.

In Tables II we present the syntax of the corresponding string operations in three commonly used programming languages, i.e. Java, C#, PHP.

2.1. Notation

We will omit the quotation marks (“ ”) when writing strings and the context is not ambiguous (e.g., abc instead of “abc”). Similarly, we will omit the apices ($'$) when writing characters (e.g., a instead of $'a'$). In addition, we define here some notation which we will use throughout the article.

Let $char(s)$ be a function that returns the set of characters contained in the string s in input, while $charAt_{\mathsf{i}}(s)$ is a function that returns the character at index i in s.

Let $trunc(s, n)$ be a function which, given a string s and a positive number n, returns the truncation of s at index n, i.e. all characters from index n onwards are removed from the string. Note that, after the application of $trunc(s, n)$, the resulting string is made by n characters.

Given a finite set of elements A, A^* is the set containing all ordered sequences of elements in A (that is, $\mathsf{A}^* = \{\mathsf{a}_1 \cdots \mathsf{a}_n : \forall \mathsf{i} \in [1..\mathsf{n}] : \mathsf{a}_i \in \mathsf{A}\}$).

Given a sequence of characters $\bar{\mathsf{s}}$, we use the notation $\bar{\mathsf{s}}[i \cdots j]$ to indicate the subsequence starting at $\bar{\mathsf{s}}[i]$ and ending at $\bar{\mathsf{s}}[j]$ (extremes included).

Given two lists l_1, l_2 of any kind, let $concatList(l_1, l_2)$ be the function that returns their concatenation.

3. CONCRETE DOMAIN AND SEMANTICS

3.1. Concrete Domain

Our concrete domain is simply made of sets of strings. Given an alphabet K, that is a finite set of characters, we define strings as sequences of characters. Formally, $\mathsf{S} = \mathsf{K}^*$

Table III. Concrete semantics

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

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 efficient manner. Our concrete lattice, aimed at formalizing the run-time behaviors of a program, is made of sets of strings: the powerset of S, that is the set containing all the subsets of S, $\wp(\mathsf{S})$. The partial order is then the set inclusion $\subseteq$.

The other lattice operators are induced by $\subseteq$. Therefore, the least upper bound (lub) corresponds to set union $\cup$, and the greatest lower bound (glb) corresponds to set intersection $\cap$. Finally, the top element $\top$ is the set S (a superset of any subset of S), while the bottom element $\bot$ is $\emptyset$ (a subset of any other set).

3.2. *Concrete Semantics*

The concrete semantics of the language introduced in Section 2 is formalized in Table III.

For the first three statements, we define the semantics $\mathbb{S}$ that, given the statement and eventually some sets of concrete string values in S (containing the values of the arguments of the statement), returns a set of concrete strings resulting from that operation. In particular, (i) `new String(str)` returns a singleton containing `str`, (ii) `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), and (iii) $\texttt{substring}_\mathsf{b}^\mathsf{e}$ returns all the substrings from the b-th to e-th character of the given strings. Note that if one of the strings is too short, there is no substring for it in the resulting set, since this would cause a runtime error.

For $\texttt{contains}_\mathsf{c}$ we define a particular semantics $\mathbb{B} : \wp(\mathsf{S}) \rightarrow \{\mathsf{true}, \mathsf{false}, \top_\mathsf{B}\}$. Given a set of strings, the semantics of this operator returns true if all the strings contain 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, depending on the string in S_1 we are considering. Therefore, we define a partial order $\geq_\mathsf{B}$ over these values such that (i) $\forall b \in \{\mathsf{true}, \mathsf{false}, \top_\mathsf{B}\} : \top_\mathsf{B} \geq_\mathsf{B} b$, (ii) $\mathsf{true} \geq_\mathsf{B} \mathsf{true}$, and (iii) $\mathsf{false} \geq_\mathsf{B} \mathsf{false}$.

4. ABSTRACT DOMAINS AND SEMANTICS

Before starting the construction of abstract domains for strings, we have to ask ourselves some questions: what is a string made of? What is the *relevant* information contained in a string? How can we approximate it in an *efficient* way? At the beginning of Section 3 we already answered the first question. The other two questions arise from the fact that it is impossible to track both sound and complete information about all possible executions at compile time. It is thus necessary to introduce some kind of *approximation*. We want to track information precise enough to efficiently analyze the behaviors of interest (considering the string operators we defined in Section 2). So our purpose is to approximate strings as much as possible, while preserving information we deem relevant. Therefore, we will have to make compromises. The first level of approximation we will introduce is a representation in which we maintain all the information we have about characters inclusion but nothing about order (Section 4.1). This approximation would behave well in programs which use string operators like `contains`. The second kind of representation we will define keeps some information about the order but not about inclusion in itself (e.g., a string which begins with an "a" and ends with a "b", but nothing about other characters which the string could contain). This representation (Section 4.2) could be particularly useful for programs which use the `substring` operator. Finally, we will present abstractions that track information on both character inclusion and order (Sections 4.3 and 4.4).

The domains introduced in Sections 4.3 and 4.4 are strictly more precise than the ones presented in Sections 4.1 and 4.2, but they are less efficient as well. Nevertheless, in some contexts the less precise domains would be precise enough to prove some properties of interest, while in other cases we would need the more complex domains. Therefore, one can tune the analysis at different levels of precision and efficiency by choosing different domains.

4.1. Character Inclusion

The first abstract domain approximates strings with the characters we know the strings *surely* contain, and ones that they *could* contain. This information could be particularly useful if the indices extrapolated from a string with operators like `indexOf(c)` could be used to cut the string (because it is interesting to know if the index is invalid, i.e., -1).

In this domain, denoted by $\overline{\mathcal{CI}}$, 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 $\overline{\mathsf{MC}}$:

$$\overline{\mathcal{CI}} = \{(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) : \overline{\mathsf{C}}, \overline{\mathsf{MC}} \in \wp(\mathsf{K}) \wedge \overline{\mathsf{C}} \subseteq \overline{\mathsf{MC}}\} \cup \bot_{\overline{\mathcal{CI}}}$$

Partial order The partial order $\leq_{\overline{\mathcal{CI}}}$ on $\overline{\mathcal{CI}}$ is defined by $(\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1) \leq_{\overline{\mathcal{CI}}} (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2) \Leftrightarrow (\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1) = \bot_{\overline{\mathcal{CI}}} \vee (\overline{\mathsf{C}}_1 \supseteq \overline{\mathsf{C}}_2 \wedge \overline{\mathsf{MC}}_1 \subseteq \overline{\mathsf{MC}}_2)$. This is because the more information we have on the string (that is, the more characters are certainly contained and the fewer characters are maybe contained), the fewer strings we are representing. Consequently, the top element of the lattice is $\top_{\overline{\mathcal{CI}}} = (\emptyset, \mathsf{K})$, while the bottom element of the lattice is defined as $\bot_{\overline{\mathcal{CI}}} = \{(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) : \overline{\mathsf{C}} \nsubseteq \overline{\mathsf{MC}}\}$.

Least upper bound and greatest lower bound The definition of these two operators is induced by the definition of the partial order. Formally, the least upper bound is defined by $\sqcup_{\overline{\mathcal{CI}}}(v_1, v_2) = \sqcup_{\overline{\mathcal{CI}}}((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)) = (\overline{\mathsf{C}}_1 \cap \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2)$.

Similarly, the greatest lower bound, instead, is defined by:

$$\sqcap_{\overline{\mathcal{CI}}}(v_1, v_2) = \sqcap_{\overline{\mathcal{CI}}}((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)) = \begin{cases} (\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cap \overline{\mathsf{MC}}_2) & \text{if } \overline{\mathsf{C}}_1 \subseteq \overline{\mathsf{MC}}_2 \wedge \overline{\mathsf{C}}_2 \subseteq \overline{\mathsf{MC}}_1 \\ \bot_{\overline{\mathcal{CI}}} & \text{otherwise} \end{cases}$$

The fact that $\sqcup_{\overline{\mathcal{CI}}}$ and $\sqcap_{\overline{\mathcal{CI}}}$ are the least upper bound and the greatest lower bound operator respectively follows from basic properties of set union and intersection.

Lemma 4.1
The abstract domain $\overline{\mathcal{CI}}$ is a complete lattice.

Proof
The proof follows straightforwardly from the fact that, for any set C, $\langle \wp(\mathsf{C}), \subseteq \rangle$ and $\langle \wp(\mathsf{C}), \supseteq \rangle$ are both complete lattices. □

Widening operator The widening operator $\nabla_{\overline{\mathcal{CI}}} : (\overline{\mathcal{CI}} \times \overline{\mathcal{CI}}) \rightarrow \overline{\mathcal{CI}}$ is defined by $(\overline{\mathsf{C}}_1, \overline{\mathsf{MC}_1}) \nabla_{\overline{\mathcal{CI}}} (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2) = (\overline{\mathsf{C}}_1, \overline{\mathsf{MC}_1}) \sqcup_{\overline{\mathcal{CI}}} (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)$ because in domains with finite height the least upper bound operator is also a widening operator since it converges in finite time. Our domain has finite height, since the height of the powerset lattice of a set S based on $\subseteq$ or $\supseteq$ is $|\mathsf{S}| + 1$, and we always consider only finite alphabets.

Abstraction and concretization functions The concretization function maps an abstract element to a set of strings. Given an abstract element $(\overline{\mathsf{C}}, \overline{\mathsf{MC}})$, the resulting strings will have to (i) contain at least all the characters in $\overline{\mathsf{C}}$, and (ii) contain at most the characters in $\overline{\mathsf{MC}}$. This is defined as follows:

$$\gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) = \{s : c_1 \in \overline{\mathsf{C}} \Rightarrow c_1 \in s \wedge c_2 \in s \Rightarrow c_2 \in \overline{\mathsf{MC}}\}$$

Theorem 4.2
Let the abstraction function $\alpha_{\mathcal{CI}}$ be defined by $\alpha_{\mathcal{CI}} = \lambda \mathsf{Y}.\, \sqcap_{\mathcal{CI}} \{(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) : \gamma_{\mathcal{CI}}((\overline{\mathsf{C}}, \overline{\mathsf{MC}})) \subseteq \mathsf{Y}\}$.
Then $\langle \wp(\mathsf{S}), \subseteq \rangle \underset{\alpha_{\mathcal{CI}}}{\overset{\gamma_{\mathcal{CI}}}{\leftrightarrows}} \langle \mathcal{CI}, \leq_{\mathcal{CI}} \rangle$.

Proof
By Theorem 1.1 we only need to prove that $\gamma_{\mathcal{CI}}$ is a complete meet morphism. Formally, we have to prove that $\gamma_{\mathcal{CI}}(\underset{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}}{\bigsqcap_{\mathcal{CI}}} (\overline{\mathsf{C}}, \overline{\mathsf{MC}})) = \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \gamma_{\mathcal{CI}}(\overline{\mathsf{C}}, \overline{\mathsf{MC}})$.

$$
\begin{aligned}
&\gamma_{\mathcal{CI}}(\underset{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}}{\bigsqcap_{\mathcal{CI}}} (\overline{\mathsf{C}}, \overline{\mathsf{MC}}))\\
&\quad\text{by Definition of } \sqcap_{\mathcal{CI}}\\
&= \gamma_{\mathcal{CI}}(\bigcup_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \overline{\mathsf{C}}, \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \overline{\mathsf{MC}})\\
&\quad\text{by Definition of } \gamma_{\mathcal{CI}}\\
&= \{s : c_1 \in \bigcup_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \overline{\mathsf{C}} \Rightarrow c_1 \in s \wedge c_2 \in s \Rightarrow c_2 \in \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \overline{\mathsf{MC}}\}\\
&\quad\text{by logic rules of set theory}\\
&= \{s : \forall (\overline{\mathsf{C}}, \overline{\mathsf{MC}}) \in \overline{\mathsf{X}} : c_1 \in \overline{\mathsf{C}} \Rightarrow c_1 \in s \wedge c_2 \in s \Rightarrow c_2 \in \overline{\mathsf{MC}}\}\\
&\quad\text{by Definition of } \cap\\
&= \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \{s : c_1 \in \overline{\mathsf{C}} \Rightarrow c_1 \in s \wedge c_2 \in s \Rightarrow c_2 \in \overline{\mathsf{MC}}\}\\
&\quad\text{by Definition of } \gamma_{\mathcal{CI}}\\
&= \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \gamma_{\mathcal{CI}}(\overline{\mathsf{C}}, \overline{\mathsf{MC}})
\end{aligned}
$$

□

Semantics Table IV defines the abstract semantics (in the abstract domain $\overline{\mathcal{CI}}$) of the operators introduced in Section 2. 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 constant (`new String(str)`), we know that the characters that are surely or maybe included are exactly the ones that appear in the string `str`.

The `concat` operator takes in input two strings and concatenates them. If a character appears (or could appear) in one of the two input strings, then it will appear (or it could appear) in the resulting string too. For this reason, we employ set union.

The `substring` operator returns a new string that is a substring of the string s in input. The $\overline{\mathsf{MC}}_1$ set remains the same, while the only sound approximation of the certainly contained characters is $\emptyset$, because we do not know the position of the certainly contained characters inside s.

The `contains` operator returns true if and only if the string in input (let it be s) contains the specified character (c). Its semantics is quite precise, as it checks if a character is surely contained or not contained respectively through $\overline{\mathsf{C}}_1$ and $\overline{\mathsf{MC}}_1$.

Theorem 4.3 (Soundness of the abstract semantics)
$\overline{\mathbb{S}_{\mathcal{CI}}}$ and $\overline{\mathbb{B}_{\mathcal{CI}}}$ are sound over-approximations of $\mathbb{S}$ and $\mathbb{B}$, respectively. Formally, $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathbb{S}_{\mathcal{CI}}}[\![\mathsf{s}]\!](\overline{\mathsf{IC}})) \supseteq \{\mathbb{S}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{IC}})\}$ and $\overline{\mathbb{B}_{\mathcal{CI}}}[\![\mathsf{s}]\!](\overline{\mathsf{IC}}) \geq_{\mathsf{B}} \{\mathbb{B}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{IC}})\}$.

Proof
We prove the soundness separately for each operator.

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

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

- $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{new String(str)}]\!]()) \supseteq \{\mathbb{S}[\![\texttt{new String(str)}]\!]()\}$ follows immediately from the definition of $\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{new String(str)}]\!]()$ and of $\gamma_{\overline{\mathcal{CI}}}$.
- Consider the binary operator `concat`. Let $\overline{\mathsf{a}}_1 = (\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), \overline{\mathsf{a}}_2 = (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)$ be two abstract states. We have to prove that $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{concat}]\!](\overline{\mathsf{a}}_1, \overline{\mathsf{a}}_2)) \supseteq \{\mathbb{S}[\![\texttt{concat}]\!](\mathsf{c}_1, \mathsf{c}_2) : \mathsf{c}_1 \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{a}}_1) \wedge \mathsf{c}_2 \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{a}}_2)\}$. A generic element $\mathsf{c}_1 \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{a}}_1)$ is a string which contains at least one occurrence of each character of $\overline{\mathsf{C}}_1$ and which characters all belong to $\overline{\mathsf{MC}}_1$; the same goes for $\mathsf{c}_2 \in \gamma(\overline{\mathsf{a}}_2)$. The concatenation of c_1 and c_2 then, by definition of $\mathbb{S}$, produces a string which contains at least one occurrence of each character of $\overline{\mathsf{C}}_1$ and of $\overline{\mathsf{C}}_2$, and which characters all belong to $\overline{\mathsf{MC}}_1$ or $\overline{\mathsf{MC}}_2$. Then, this string belongs to $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{concat}]\!](\overline{\mathsf{a}}_1, \overline{\mathsf{a}}_2))$, because $\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{concat}]\!](\overline{\mathsf{a}}_1, \overline{\mathsf{a}}_2) = (\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2)$ by definition of $\overline{\mathbb{S}_{\mathcal{CI}}}$. Then $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2)$ contains, by definition of $\gamma_{\overline{\mathcal{CI}}}$, all strings which contain at least one occurrence of each character of $\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2$, and which characters all belong to $\overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2$.
- Consider the unary operator $\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}$. Our theorem trivially holds since the abstract semantics returns the top element of $\overline{\mathcal{CI}}$, that concretizes to all the possible strings. This trivially overapproximates any possible result of the concrete semantics.
- Consider the unary operator $\texttt{contains}_{\mathsf{c}}$ and let $\overline{\mathsf{a}} = (\overline{\mathsf{C}}, \overline{\mathsf{MC}})$ be an abstract state. Considering the character `c`, we have three cases:
  - If $\mathsf{c} \in \overline{\mathsf{C}}$, all the strings belonging to $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{a}})$ contain at least one occurrence of `c` by definition of $\gamma_{\overline{\mathcal{CI}}}$. Then, the concrete semantics returns always `true` on this set. On the other hand, the abstract semantics on $\overline{\mathsf{a}}$ returns the $\overline{\mathsf{true}}$ value of the boolean domain, so it soundly approximates the concrete semantics.
  - If $\mathsf{c} \in \overline{\mathsf{MC}}$ and $\mathsf{c} \notin \overline{\mathsf{C}}$, then the abstract semantics returns $\top_{\mathsf{B}}$ that trivially approximates any possible result of the concrete semantics.
  - If $\mathsf{c} \notin \overline{\mathsf{C}} \wedge \mathsf{c} \notin \overline{\mathsf{MC}}$, no string belonging to $\gamma(\overline{\mathsf{a}})$ will contain the character. The concrete semantics will therefore return always `false`, and the abstract semantics on $\overline{\mathsf{a}}$ returns the $\overline{\mathsf{false}}$ value of the boolean domain as well.

□

Running Example Consider now the examples introduced in Section 1.1.

The results of the analysis of `prog1` using $\overline{\mathcal{CI}}$ are depicted in Figure 2(a). At the beginning, the variable `query` is related to a state that contains the abstraction of s_1. The value of `l` is unknown, so we must compute the least upper bound between the abstract values of `query` after instructions 1 and 4. The set $\overline{\mathsf{C}}$ of `query` after instruction 4 contains all the character of $\mathsf{s}_1, \mathsf{s}_2$ and s_3, because they are all concatenated; the $\overline{\mathsf{MC}}$ set instead is K because of the concatenation with `l`. Then, after the `if` statement (line 5) the abstract value of `query` contains the abstraction of s_1 in $\overline{\mathsf{C}}$, and K in $\overline{\mathsf{MC}}$ (because of `l`). The variable `per` is related (line 6) to a state that contains the abstraction of s_4. At line 8, `query` is concatenated to $\mathsf{s}_4, \mathsf{s}_5$ and s_6. Then, at the end of the given code, `query` surely

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

(a) Analysis of `prog1`

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

(b) Analysis of `prog2`

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

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 `prog2`, in Figure 2(b) we see that after instruction `1 x` surely contains the character 'a'. After the first iteration of the loop (line `3`), `x` surely contains 'a', '0' and '1'. At 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'. This is the final result of the program. In fact, we do not know the value of `cond`, so we cannot know beforehand how many iterations will be done by the loop. In such cases, we have to use the widening to reach the convergence. Here the analysis converges immediately after the second iteration, since the abstract value obtained after two iterations (that is, $(\{0, a, 1\}, \{0, a, 1\})$) is the same as the one obtained after one iteration.

4.2. *Prefix and Suffix*

First of all, we define a domain that abstracts strings through their *prefix*. We represent a prefix by a sequence of characters followed by an asterisk $*$. The asterisk represents any string (the empty 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. Formally, $\overline{\mathcal{PR}} = \mathsf{K}^* \cup \bot_{\overline{\mathcal{PR}}}$.

Partial order The partial order is defined by:

$$\overline{\mathsf{p}}_1 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2 \Leftrightarrow \overline{\mathsf{p}}_1 = \bot_{\overline{\mathcal{PR}}} \vee (len(\overline{\mathsf{p}}_2) \leq len(\overline{\mathsf{p}}_1) \wedge (\forall \mathsf{i} \in [0, len(\overline{\mathsf{p}}_2) - 1] : \overline{\mathsf{p}}_2[\mathsf{i}] = \overline{\mathsf{p}}_1[\mathsf{i}]))$$

An abstract string $\overline{\mathsf{S}}$ is $\leq_{\overline{\mathcal{PR}}}$ than another abstract string $\overline{\mathsf{T}}$ 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 top element is $*$, since $*$ is the empty prefix, which is prefix of any other prefix. Instead the bottom value is the special element $\bot_{\overline{\mathcal{PR}}}$.

Note that this domain has an infinite height. In fact, given any prefix, we can always add a character at the end of it, thus obtaining a new prefix, longer (and smaller according to the order $\leq_{\overline{\mathcal{PR}}}$) than the first one. However, the domain respects the ascending chain condition (ACC), and we do not need to define a widening operator to ensure the convergence of the analysis. In fact, given a certain prefix $\overline{\mathsf{p}}$, where $len(\overline{\mathsf{p}}) = n$, the ascending chain starting at $\overline{\mathsf{p}}$ is $\overline{\mathsf{p}} \to \overline{\mathsf{p}}_1 \to \overline{\mathsf{p}}_2 \to \cdots \to \overline{\mathsf{p}}_n$ where $\overline{\mathsf{p}}_1 = trunc(\overline{\mathsf{p}}, n-1)$ (that is, $\overline{\mathsf{p}}_1$ corresponds to $\overline{\mathsf{p}}$ without its last character), $\overline{\mathsf{p}}_2 = trunc(\overline{\mathsf{p}}_1, n-2)$, $\overline{\mathsf{p}}_3 = trunc(\overline{\mathsf{p}}_2, n-3)$, and so on, until we reach $\overline{\mathsf{p}}_n = trunc(\overline{\mathsf{p}}_{n-1}, n-n) = trunc(\overline{\mathsf{p}}_{n-1}, 0) = \epsilon$. $\overline{\mathsf{p}}_n$ corresponds to an empty prefix: it is $*$, which represents any string, the top of our domain. Thus, given any prefix $\overline{\mathsf{p}}$ of length n (which is finite), the ascending chain starting at $\overline{\mathsf{p}}$ has finite length $n+1$.

Least upper bound and greatest lower bound Given two prefixes, their least upper bound $\sqcup_{\overline{\mathcal{PR}}}$ is their longest common prefix. If the two prefixes do not have anything in common, the least upper bound is $*$ (the prefix is empty). Instead, the greatest lower bound operator is defined by:

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

Lemma 4.4
$\sqcup_{\overline{\mathcal{PR}}}$ is the least upper bound operator.

Proof
Let $\overline{\mathsf{p}} = \overline{\mathsf{p}}_1 \sqcup_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2$ be the least upper bound of $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. Then we have to prove the two following conditions:

1. $\overline{\mathsf{p}}_1 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}} \wedge \overline{\mathsf{p}}_2 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}$ straightforwardly holds, since $\overline{\mathsf{p}}$ is the longest common prefix between $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ by definition of $\sqcup_{\overline{\mathcal{PR}}}$, so it is a prefix of both. This implies $\overline{\mathsf{p}}_1 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}} \wedge \overline{\mathsf{p}}_2 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}$ by definition of $\leq_{\overline{\mathcal{PR}}}$.

2. $\overline{\mathsf{p}} \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}' \; \forall$ upper bound $\overline{\mathsf{p}}'$ of $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. By definition of the lattice structure of $\overline{\mathcal{PR}}$, $\overline{\mathsf{p}}'$ has to be a prefix of *both* $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. Since $\overline{\mathsf{p}}$ is the *longest* common prefix between $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ by definition of $\sqcup_{\overline{\mathcal{PR}}}$, we know for sure that $\overline{\mathsf{p}}'$ cannot be longer than $\overline{\mathsf{p}}$: $\overline{\mathsf{p}}'$ is then a prefix of $\overline{\mathsf{p}}$, and so we proved $\overline{\mathsf{p}} \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}'$ by definition of $\leq_{\overline{\mathcal{PR}}}$.

□

Lemma 4.5
$\sqcap_{\overline{\mathcal{PR}}}$ is a greatest lower bound operator.

Proof
Let $\overline{\mathsf{p}} = \overline{\mathsf{p}}_1 \sqcap_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2$ be the greatest lower bound of $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. Then we have to prove the two following conditions:

1. $\overline{\mathsf{p}} \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_1 \wedge \overline{\mathsf{p}} \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2$ comes straightforwardly from the definition of $\sqcap_{\overline{\mathcal{PR}}}$.
2. $\overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}} \; \forall$ lower bound $\overline{\mathsf{p}}'$ of $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. If $\overline{\mathsf{p}}' = \bot_{\overline{\mathcal{PR}}}$, by definition of $\leq_{\overline{\mathcal{PR}}}$ it surely holds that $\overline{\mathsf{p}}' \leq_{\overline{\mathcal{CI}}} \overline{\mathsf{p}}$. Otherwise, it must hold that *both* $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ are prefixes of $\overline{\mathsf{p}}'$ by definition of the lattice structure of $\overline{\mathcal{PR}}$. Then, it holds that $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ are one the prefix of the other one (since they are both prefixes of the same string $\overline{\mathsf{p}}'$). Suppose that $\overline{\mathsf{p}}_1$ is the prefix of $\overline{\mathsf{p}}_2$ (the other case is symmetrical): then, $\overline{\mathsf{p}}_2 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_1$ by definition of $\leq_{\overline{\mathcal{PR}}}$. If this is the case, by definition of $\sqcap_{\overline{\mathcal{PR}}}$, we also know that $\overline{\mathsf{p}} = \overline{\mathsf{p}}_2$. Since $\overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2$ by hypothesis and $\overline{\mathsf{p}} = \overline{\mathsf{p}}_2$, we get that $\overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}$ by definition of $\leq_{\overline{\mathcal{PR}}}$.

□

Lemma 4.6
The abstract domain $\overline{\mathcal{PR}}$ is a complete lattice.

Proof
The order based on prefixes is a partial order. Informally: (i) a string is always a prefix of itself (reflexivity); (ii) if a string is prefix of another one and viceversa, then the two strings have to be the same string (antisymmetry); (iii) if a string s_1 is prefix of another string s_2 and s_2 is prefix of another string s_3, then s_1 is also a prefix of s_3 (transitivity).

The fact that $\sqcup_{\overline{\mathcal{PR}}}$ and $\sqcap_{\overline{\mathcal{PR}}}$ are the least upper bound and the greatest lower bound operators is proved by the two previous Theorems.

□

Abstraction and concretization functions The concretization function is defined as follows:

$$\gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}) = \begin{cases} \emptyset & \text{if } \overline{\mathsf{p}} = \bot_{\overline{\mathcal{PR}}} \\ \{s : s \in \mathsf{K}^* \wedge len(s) \geq len(\overline{\mathsf{p}}) \wedge \; \forall i \in [0, len(\overline{\mathsf{p}}) - 1] : s[i] = \overline{\mathsf{p}}[i]\} & \text{otherwise} \end{cases}$$

The abstract value p maps to the set of the strings which begin with the sequence of characters represented by p.

Theorem 4.7
Let the abstraction function $\alpha_{\overline{\mathcal{PR}}}$ be defined by $\alpha_{\overline{\mathcal{PR}}} = \lambda \mathsf{Y}. \sqcap_{\overline{\mathcal{PR}}} \{\overline{\mathsf{p}} : \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}) \subseteq \mathsf{Y}\}$.

Then $\langle \wp(\mathsf{S}), \subseteq \rangle \underset{\alpha_{\overline{\mathcal{PR}}}}{\overset{\gamma_{\overline{\mathcal{PR}}}}{\leftrightarrows}} \langle \overline{\mathcal{PR}}, \leq_{\overline{\mathcal{PR}}} \rangle$.

Proof
By Theorem 1.1 we only need to prove that $\gamma_{\overline{\mathcal{PR}}}$ is a complete meet morphism. Formally, we have to prove that $\gamma_{\overline{\mathcal{PR}}}(\bigsqcap_{\substack{\mathcal{PR} \\ (\overline{\mathsf{p}}) \in \overline{\mathsf{X}}}} \overline{\mathsf{p}}) = \bigcap_{\overline{\mathsf{p}} \in \overline{\mathsf{X}}} \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}})$.

By definition of $\bigsqcap_{\overline{\mathcal{PR}}}$, we can have only the two following cases:

Table V. The abstract semantics of $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$

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

1. $\exists \overline{\mathsf{p}}' \in \overline{\mathsf{X}} : \forall \overline{\mathsf{p}} \in \overline{\mathsf{X}} : \overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}$. Then we have the following inference chain:

$$\begin{aligned}
&\gamma_{\overline{\mathcal{PR}}}(\textstyle\bigsqcap_{\overline{\mathcal{PR}}\,(\overline{\mathsf{p}})\in\overline{\mathsf{X}}} \overline{\mathsf{p}}) \\
&\text{by Definition of } \sqcap_{\overline{\mathcal{PR}}} \\
&= \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}') \\
&\text{by Definition of } \gamma_{\overline{\mathcal{PR}}} \\
&= \{s : s \in \mathsf{K}^* \wedge len(s) \geq len(\overline{\mathsf{p}}') \wedge \ \forall i \in [0, len(\overline{\mathsf{p}}') - 1] : s[i] = \overline{\mathsf{p}}'[i]\} \\
&\text{by Definition of } \leq_{\overline{\mathcal{PR}}} \text{ since } \forall \overline{\mathsf{p}} \in \overline{\mathsf{X}} : \overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}} \\
&= \textstyle\bigcap_{\overline{\mathsf{p}}\in\overline{\mathsf{X}}} \{s : s \in \mathsf{K}^* \wedge len(s) \geq len(\overline{\mathsf{p}}) \wedge \ \forall i \in [0, len(\overline{\mathsf{p}}) - 1] : s[i] = \overline{\mathsf{p}}[i]\} \\
&\text{by Definition of } \gamma_{\overline{\mathcal{PR}}} \\
&= \textstyle\bigcap_{\overline{\mathsf{p}}\in\overline{\mathsf{X}}} \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}})
\end{aligned}$$

2. otherwise, $\gamma_{\overline{\mathcal{PR}}}(\bigsqcap_{\overline{\mathcal{PR}}\,(\overline{\mathsf{p}})\in\overline{\mathsf{X}}} \overline{\mathsf{p}}) = \gamma_{\overline{\mathcal{PR}}}(\perp_{\overline{\mathcal{PR}}}) = \emptyset$. Then $\bigcap_{\overline{\mathsf{p}}\in\overline{\mathsf{X}}} \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{a}}) = \emptyset$, since there is no concretized strings in common among abstract states that represent different prefixes.

□

Similarly, we can track information about the *suffix* of a string. We introduce another abstract domain, $\overline{\mathcal{SU}}$, where a string is approximated by the *end* of a certain sequence of characters, while we do not track anything about the string *before* such suffix. The notation and all the operators of this domain are dual to those of $\overline{\mathcal{PR}}$ domain.

The domain definition is: $\overline{\mathcal{SU}} = \mathsf{K}^* \cup \perp_{\overline{\mathcal{SU}}}$. As for the partial order, $\overline{\mathsf{s}}_1 \leq_{\overline{\mathcal{SU}}} \overline{\mathsf{s}}_2$ if $\overline{\mathsf{s}}_2$ is a suffix of $\overline{\mathsf{s}}_1$ or if $\overline{\mathsf{s}}_1$ is the bottom value of the domain, $\perp_{\overline{\mathcal{SU}}}$. The top element $\top_{\overline{\mathcal{SU}}}$ is $*$, while the bottom value is the special element $\perp_{\overline{\mathcal{SU}}}$. The least upper bound operator $\sqcup_{\overline{\mathcal{SU}}}$, dually to $\sqcup_{\overline{\mathcal{PR}}}$, is defined as the longest common suffix between the two suffixes in input. As for the greatest lower bound, if the two suffixes are not comparable with respect to the order $\leq_{\overline{\mathcal{SU}}}$ (e.g., $*a$ and $*b$), then the string sets they represent have nothing in common and their glb is thus $\perp_{\overline{\mathcal{SU}}}$. If they are comparable, the smaller element between the two is the greatest lower bound. $\overline{\mathcal{SU}}$ is a domain with infinite height, just like $\overline{\mathcal{PR}}$. In fact, given any suffix, we can always add a character at its beginning, thus obtaining a new suffix, longer (therefore smaller, according to the order $\leq_{\overline{\mathcal{SU}}}$) than the initial one. As it happened with $\overline{\mathcal{PR}}$, though, this domain respects the ACC condition, and it does not need a widening operator. The concretization function maps an abstract value $\overline{\mathsf{a}}$ to the set of the strings which end with the sequence of characters represented by $\overline{\mathsf{a}}$.

All the proofs for this domain are symmetrical to those presented for $\overline{\mathcal{PR}}$.

Semantics Table V defines the abstract semantics on $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$. Let us explain in detail the semantics of each operator. When we evaluate a constant string value (`new String(str)`), the most precise suffix and prefix are the string itself. When we concatenate two strings, we create a new string which starts with the first one and ends with the second one. Then, we consider as prefix and suffix of the resulting string the abstract value of the left and right operand, respectively. The semantics of $\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}$ is $\top_{\overline{\mathcal{SU}}}$ in $\overline{\mathcal{SU}}$, since we do not know how many characters there are *before* the suffix ($\mathtt{b}$ and $\mathtt{e}$ are relative to the beginning of the string). With $\overline{\mathcal{PR}}$, instead, we *do* know how the string begins, so we can be more precise if $\mathtt{b}$ (and eventually $\mathtt{e}$) are smaller than the length of the prefix we have. We have to distinguish three different cases: (i) if $\mathtt{e} \leq len(\overline{\mathtt{p}})$, the substring is completely included in the known prefix; (ii) if $\mathtt{e} > len(\overline{\mathtt{p}})$ but $\mathtt{b} < len(\overline{\mathtt{p}})$, only the first part of the substring is in the prefix; (iii) if $\mathtt{b} \geq len(\overline{\mathtt{p}})$, the substring is completely further the prefix and we return $\top_{\overline{\mathcal{PR}}}$. The semantics of $\mathtt{contains}_\mathtt{c}$ returns $\mathtt{true}$ iff $\mathtt{c}$ is contained in the prefix or in the suffix, and $\top_\mathtt{B}$ otherwise, since we have no information at all about which characters are after the prefix or before the suffix.

Theorem 4.8 (Soundness of the abstract semantics)
$\overline{\mathbb{S}_{\mathcal{PR}}}$ and $\overline{\mathbb{B}_{\mathcal{PR}}}$ are a sound overapproximation of $\mathbb{S}$ and $\mathbb{B}$, respectively. Formally, $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathtt{s}]\!](\overline{\mathtt{p}})) \supseteq \{\mathbb{S}[\![\mathtt{s}]\!](\mathtt{c}) : \mathtt{c} \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathtt{p}})\}$ and $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{B}_{\mathcal{PR}}}[\![\mathtt{s}]\!](\overline{\mathtt{p}})) \geq_\mathtt{B} \{\mathbb{B}[\![\mathtt{s}]\!](\mathtt{c}) : \mathtt{c} \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathtt{p}})\}$.

Proof
We prove the soundness separately for each operator. We only prove the soundness for the $\overline{\mathcal{PR}}$ domain: the proof for $\overline{\mathcal{SU}}$ are simply their mirror image.

- $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathtt{new\ String(str)}]\!]()) \supseteq \{\mathbb{S}[\![\mathtt{new\ String(str)}]\!]()\}$ follows immediately from the definition of $\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathtt{new\ String(str)}]\!]()$ and of $\gamma_{\overline{\mathcal{PR}}}$.
- Consider the binary operator $\mathtt{concat}$. Let $\overline{\mathtt{p}}_1$ and $\overline{\mathtt{p}}_2$ be two prefixes. We have to prove that $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathtt{concat}]\!](\overline{\mathtt{p}}_1, \overline{\mathtt{p}}_2)) \supseteq \{\mathbb{S}[\![\mathtt{concat}]\!](\mathtt{c}_1, \mathtt{c}_2) : \mathtt{c}_1 \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathtt{p}}_1) \wedge \mathtt{c}_2 \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathtt{p}}_2)\}$. A generic element $\mathtt{c}_1 \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathtt{p}}_1)$ is a string which starts with the prefix $\overline{\mathtt{p}}_1$; the same goes for $\mathtt{c}_2 \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathtt{p}}_2)$. The concatenation of $\mathtt{c}_1$ and $\mathtt{c}_2$ produces a string which starts with $\overline{\mathtt{p}}_1$ and afterwards contains $\overline{\mathtt{p}}_2$ (in an unknown position) by definition of $\mathbb{S}$. Then, this string belongs to $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathtt{concat}]\!](\overline{\mathtt{p}}_1, \overline{\mathtt{p}}_2))$, since $\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathtt{concat}]\!](\overline{\mathtt{p}}_1, \overline{\mathtt{p}}_2) = \overline{\mathtt{p}}_1$ by definition of $\overline{\mathbb{S}_{\mathcal{PR}}}$, and $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathtt{p}}_1)$ returns all the strings which start with $\overline{\mathtt{p}}_1$.
- Consider the unary operator $\mathtt{substring}_\mathtt{b}^\mathtt{e}$ and let $\overline{\mathtt{p}}$ be an abstract state. A generic string $\mathtt{c} \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathtt{p}})$ is a string which starts with $\overline{\mathtt{p}}$ by definition of $\gamma_{\overline{\mathcal{PR}}}$. We may have only the following three cases:
  - if $\mathtt{e} \leq len(\overline{\mathtt{p}})$, the substring of $\mathtt{c}$ from the $\mathtt{b}$-th character to the $\mathtt{e}$-th character is completely known (since the prefix $\overline{\mathtt{p}}$ is longer than $\mathtt{e}$ characters) and the result of the concrete semantics applied to $\mathtt{c}$ is the substring from the $\mathtt{b}$th to the $\mathtt{e} - 1$th character. $\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathtt{substring}_\mathtt{b}^\mathtt{e}]\!](\overline{\mathtt{p}})$ returns the prefix composed by the substring from $\overline{\mathtt{p}}[\mathtt{b}]$ to $\overline{\mathtt{p}}[\mathtt{e} - 1]$ by definition of $\overline{\mathbb{S}_{\mathcal{PR}}}$. The concretization of this result returns all the strings starting with the substring from $\overline{\mathtt{p}}[\mathtt{b}]$ to $\overline{\mathtt{p}}[\mathtt{e} - 1]$ by definition of $\gamma_{\overline{\mathcal{PR}}}$, thus it contains also such substring that is the result of the concrete semantics.
  - if $\mathtt{e} > len(\overline{\mathtt{p}}) \wedge \mathtt{b} < len(\overline{\mathtt{p}})$, since $\mathtt{c}$ starts with $\overline{\mathtt{p}}$ by definition of $\gamma_{\overline{\mathcal{PR}}}$, we surely know that the substring of $\mathtt{c}$ from the $\mathtt{b}$-th character to the $\mathtt{e}$-th character starts with the characters from $\overline{\mathtt{p}}[\mathtt{b}]$ to $\overline{\mathtt{p}}[len(\overline{\mathtt{p}}) - 1]$ by definition of $\mathbb{S}$. $\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathtt{substring}_\mathtt{b}^\mathtt{e}]\!](\overline{\mathtt{p}})$ returns the prefix made by the characters from $\overline{\mathtt{p}}[\mathtt{b}]$ to $\overline{\mathtt{p}}[len(\overline{\mathtt{p}}) - 1]$, thus representing all strings starting with such characters by definition of $\gamma_{\overline{\mathcal{PR}}}$. Therefore, it surely contains also the resulting substring of $\mathtt{c}$.
  - otherwise, $\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathtt{substring}_\mathtt{b}^\mathtt{e}]\!](\overline{\mathtt{p}})$ returns ϵ, that is, the top element of $\overline{\mathcal{PR}}$, that trivially overapproximates any possible result of the concrete semantics.
- Consider the unary operator $\mathtt{contains}_\mathtt{c}$ and let $\overline{\mathtt{p}}$ be an abstract prefix. Regarding the character $\mathtt{c}$, we have two possible cases:
  - If $\mathtt{c} \in char(\overline{\mathtt{p}})$, all the strings belonging to $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathtt{p}})$ contain at least one occurrence of $\mathtt{c}$, because they start with the prefix $\overline{\mathtt{p}}$ by definition of $\gamma_{\overline{\mathcal{PR}}}$, and such prefix contains the character $\mathtt{c}$. Then, the concrete semantics returns always $\mathtt{true}$, and the abstract semantics returns the same result.
  - Otherwise, $\mathtt{c} \notin char(\overline{\mathtt{p}})$, and the abstract semantics returns $\top_\mathtt{B}$, that trivially overapproximates any possible result of the concrete semantics.

□

#I	Var	$\overline{\mathcal{PR}}$	$\overline{\mathcal{SU}}$
1	query	$\overline{s_1}$	$\overline{s_1}$
3	l	ϵ	ϵ
4	query	$\overline{s_1}$	$\overline{s_3}$
5	query	$\overline{s_1}$	" "
6	per	$\overline{s_4}$	$\overline{s_4}$
8	query	$\overline{s_1}$	$\overline{s_6}$

(a) Analysis of `prog1`

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

(b) Analysis of `prog2`

Figure 3. The results of $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$

Running Example The results of the analysis using the prefix and suffix domains on the two running examples are reported by Figure 3.

For `prog1`, at line 1, `query` contains the whole string s_1 as both prefix and suffix. `l` is an input, so its prefix and suffix are both empty. After the concatenation at line 4, the prefix will be equal to s_1, the suffix to s_3 because we keep the prefix of the first string being concatenated and the suffix of the last one. Since the value of `l` is unknown, we must compute the least upper bound between the abstract values of `query` after lines 1 and 4. Then, at line 5, the prefix is s_1 and the suffix is a space character (the longest common suffix between s_1 and s_3). The variable `per` is associated at line 6 to s_4 for both the prefix and the suffix. At the end of the analysis, from the concatenation of line 8 we get that the prefix of `query` is string s_1 and its suffix is s_6, although we lose information about what there is in the middle.

For `prog2`, before entering the loop we know that the prefix and suffix of `x` are both an 'a' character. After the first iteration of the loop we get that the prefix of `x` is '0' and its suffix is '1'. The least upper bound of such state with the state *before* the loop (prefix and suffix are both an 'a' character), unfortunately goes to $\top$ (the longest common prefixes and suffixes are empty). Then, we reached convergence after just one iteration (since the least upper bound of any element with the $\top$ value returns always the $\top$ value), but we lost all the information.

4.3. Bricks

The domains already introduced do not track precise information about the order of characters. In fact, in $\overline{\mathcal{CI}}$ (Section 4.1) each character of the abstract representation was completely unrelated with regard to the others, while in the $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$ domains (Section 4.2) we also considered order, but limited at the beginning (or at the end) of the string. Instead, the abstract domain we will define in this Section will consider both inclusion and order among characters, but not limited to the beginning or the end of the string. 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). Obviously, this comes at a price: this abstract domain (called $\overline{\mathcal{BR}}$) is more expressive than $\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$, and $\overline{\mathcal{SU}}$. $\overline{\mathcal{BR}}$ is based on the idea of identifying a string through a regular expression, but full regular expressions are too much complex for our purposes, and thus we will approximate them.

In $\overline{\mathcal{BR}}$, a string is approximated by a sequence of *bricks*. A single brick is defined by $\overline{\mathcal{B}} = [\wp(\mathsf{S})]^{\mathsf{min},\mathsf{max}}$, where min and max are two integer positive values and S is the set of all strings. A brick represents all the strings which can be built through concatenation of the given strings (a subset of S), taken between min and max times altogether. For instance, $[\{\text{“}mo\text{”}, \text{“}de\text{”}\}]^{1,2}$ corresponds to $\{mo, de, momo, dede, mode, demo\}$

Elements in $\overline{\mathcal{BR}}$ represent strings as ordered lists of bricks. For instance, $[\{\text{“}straw\text{”}\}]^{0,1}[\{\text{“}berry\text{”}\}]^{1,1} = \{berry, strawberry\}$ since $[\{\text{“}straw\text{”}\}]^{0,1}$ concretizes to $\{\epsilon, \text{“}straw\text{”}\}$ and $[\{\text{“}berry\text{”}\}]^{1,1}$ to $\{\text{“}berry\text{”}\}$. Formally, concatenation between bricks is defined as:

$$\overline{\mathsf{B}}_1\overline{\mathsf{B}}_2 = \{\alpha\beta : \alpha \in strings(\overline{\mathsf{B}}_1) \land \beta \in strings(\overline{\mathsf{B}}_2)\}$$

where $strings(\overline{\mathsf{B}})$ represents all the strings which can be built from the single brick $\overline{\mathsf{B}}$.

Since a particular set of strings could be represented by more than one combination of bricks (for example, abc is represented by $[\{abc\}]^{1,1}$ but also by $[\{a\}]^{1,1}[\{b\}]^{1,1}[\{c\}]^{1,1}$, etc...), we adopted a normalized form. The normalization algorithm is based on five normalizing rules. The normal representation can be seen as the fixpoint of the application of the five rules to a given representation. We call $\overline{normBricks}(\overline{\mathsf{L}})$ the function which, given a list of bricks $\overline{\mathsf{L}}$, returns its normalized version. The five normalization rules are as follows [‡]:

Rule 1 remove unnecessary bricks, i.e., bricks of the form: $[\emptyset]^{0,0}$, since they represent only the empty string, which is the neutral element of the concatenation operation.

Rule 2 merge successive bricks with the same indices, $\mathsf{min} = 1$ and $\mathsf{max} = 1$, in a new single brick where the indices remain the same ($\mathsf{min} = \mathsf{max} = 1$), and the strings set is the concatenation the two original strings sets (i.e., each string is made by the concatenation of one string from the first set and one from the second set, in this order). For example, the two bricks $\overline{\mathsf{B}_0} = [\{a, cd\}]^{(1,1)}$ and $\overline{\mathsf{B}_1} = [\{b, ef\}]^{(1,1)}$ become, after the application of the second rule, the new single brick $\overline{\mathsf{B}'} = [\{ab, aef, cdb, cdef\}]^{(1,1)}$.

Rule 3 transform a brick in which the number of applications is constant ($\mathsf{min} = \mathsf{max}$) into one in which the indices are 1 ($\mathsf{min} = \mathsf{max} = 1$). Formally, a brick of the form $\overline{\mathsf{B}_0} = [\mathsf{S}_0]^{(m,m)}$ becomes the brick $\overline{\mathsf{B}}' = [{\mathsf{S}_0}^m]^{(1,1)}$, where ${\mathsf{S}_0}^m$ represents the concatenation of S_0 with itself for m times. For example, $\overline{\mathsf{B}} = [\{a, b\}]^{(2,2)}$ becomes $\overline{\mathsf{B}'} = [\{aa, ab, ba, bb\}]^{(1,1)}$.

Rule 4 merge two successive bricks in which the set of strings is the same ($\mathsf{S}_\mathsf{i} = \mathsf{S}_{\mathsf{i}+1}$) into a single one modifying the indices. Formally, the bricks $\overline{\mathsf{B}_\mathsf{i}} = [\mathsf{S}_\mathsf{i}]^{(m_1,M_1)}$ and $\overline{\mathsf{B}_{\mathsf{i}+1}} = [\mathsf{S}_\mathsf{i}]^{(m_2,M_2)}$ become the new single bricks $\overline{\mathsf{B}} = [\mathsf{S}_\mathsf{i}]^{(m_1+m_2,M_1+M_2)}$.

Rule 5 break a single brick with $\mathsf{min} \geq 1 \wedge \mathsf{max} \neq \mathsf{min}$ into two simpler bricks. More precisely, a brick of the form $\overline{\mathsf{B}_\mathsf{i}} = [\mathsf{S}_\mathsf{i}]^{(min,max)}$, where $\mathsf{min} \geq 1 \wedge \mathsf{max} \neq \mathsf{min}$, becomes the concatenation of $\overline{\mathsf{B}_{\mathsf{i}1}} = [{\mathsf{S}_\mathsf{i}}^{min}]^{(1,1)}$ and $\overline{\mathsf{B}_{\mathsf{i}2}} = [\mathsf{S}_\mathsf{i}]^{(0,max-min)}$. A simple example is the following one: the brick $\overline{\mathsf{B}} = [\{a\}]^{(2,5)}$ becomes the concatenation of the two bricks $\overline{\mathsf{B}_1} = [\{aa\}]^{(1,1)}$ and $\overline{\mathsf{B}_2} = [\{a\}]^{(0,3)}$.

Let us present an example of the normalization process. Consider the bricks list $[\{a\}]^{(1,1)}[\{a, b\}]^{(2,3)}[\{a, b\}]^{(0,1)}$. First, we can apply the fourth rule to the second and third brick, merging them because their strings set is the same. We obtain the new bricks list $[\{a\}]^{(1,1)}[\{a, b\}]^{(2,4)}$. Now we can apply the fifth rule to the second brick ($[\{a, b\}]^{(2,4)}$), which gets split into the concatenation of two bricks: $[\{aa, ab, ba, bb\}]^{(1,1)}$ and $[\{a, b\}]^{(0,2)}$. The resulting bricks list is then: $[\{a\}]^{(1,1)}[\{aa, ab, ba, bb\}]^{(1,1)}[\{a, b\}]^{(0,2)}$. Finally, we can apply the second rule to the first two bricks, merging them because of their indices range $(1, 1)$. The final bricks list is then: $[\{aaa, aab, aba, abb\}]^{(1,1)}[\{a, b\}]^{(0,2)}$. We cannot apply any more rules to such representation, therefore we have reached a normal state.

Note that, in a normalized element of $\overline{\mathcal{BR}}$, there cannot be two successive bricks with both $\mathsf{min} = \mathsf{max} = 1$. In fact, they would be merged into one single brick by the second rule. Moreover, we cannot have a brick with $\mathsf{min} \geq 1 \wedge \mathsf{max} \geq \mathsf{min}$, since it would be simplified by the third (if $\mathsf{min} = \mathsf{max}$) or fifth (if $\mathsf{max} > \mathsf{min}$) rule. In addition, we cannot have bricks with indices $\mathsf{min} = \mathsf{max} = 0$, since they would be removed by the first rule. Thus, every brick of the normalized list will be in the form $[\mathsf{T}]^{1,1}$ or $[\mathsf{T}]^{0,\mathsf{max}>0}$ (where T is a set of strings).

The abstract domain of bricks is defined by $\overline{\mathcal{BR}} = \overline{normBricks}(\overline{\mathcal{B}}^*)$, that is, the set of all finite normalized sequences of bricks.

Comparison between lists of bricks In the definition of lattice and semantics operators, we will often need have to deal with various lists of bricks of different length. However, it is usually convenient to deal with lists of the same size to define effective operators. When dealing with two abstract elements, this means to augment the shorter list with some empty bricks ($E = [\emptyset]^{(0,0)}$). In fact,

[‡]After presenting the concretization function, we will prove the soundness of these normalization rules

empty bricks represent the empty string, and adding empty bricks in *any* position of a bricks list will not change the set of strings represented by such bricks list.

A crucial question is *where* to insert the empty bricks in the shorter list. Let $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ be two lists of bricks, and let $\overline{\mathsf{L}}_1$ be the shortest one. Let n_1 be the number of bricks of $\overline{\mathsf{L}}_1$, n_2 the number of bricks of $\overline{\mathsf{L}}_2$, and n be their difference ($n = n_2 - n_1$). Then, we have to add n empty bricks to $\overline{\mathsf{L}}_1$. The simplest solution would be to insert *all* n bricks at the beginning (or end) of $\overline{\mathsf{L}}_1$. However, this method often induces loss of precision, because it does not consider possible "similarities" between bricks from the two lists. Hence, we choose to adopt a different and more precise approach. The idea is that, for each brick of the shorter list, we check if the same brick appears in the other list. If so, we modify the shorter list by adding empty bricks such that the two equal bricks will appear in the same position in the two lists. If no pair of equal bricks is found, the algorithm works in a way that all n empty bricks are added at the beginning of the shorter list. More formally, the algorithm used to pad the shorter list with empty bricks is as follows:

Algorithm 1 Algorithm for making two lists of bricks of the same size, by padding the shorter one with empty bricks, where $removeHead(L)$ is a helper function which removes the first value of the list L in input and $L.add(v)$ is a function which adds the value v at the end of the list L, and E represents the empty brick

```
 1: function $padList(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$
 2:     $n_1 \leftarrow length(\overline{\mathsf{L}}_1)$
 3:     $n_2 \leftarrow length(\overline{\mathsf{L}}_2)$
 4:     $n \leftarrow n_2 - n_1$
 5:     $\overline{\mathsf{L}}_{new} \leftarrow List.empty$
 6:     $emptyBricksAdded \leftarrow 0$
 7:     for $i = 0 \rightarrow n_2 - 1$ do
 8:         if $emptyBricksAdded \geq n$ then
 9:             $\overline{\mathsf{L}}_{new} \leftarrow \overline{\mathsf{L}}_{new}.add(\overline{\mathsf{L}}_1[0])$
10:             $removeHead(\overline{\mathsf{L}}_1)$
11:         else if $empty(\overline{\mathsf{L}}_1) \vee \overline{\mathsf{L}}_1[0]! = \overline{\mathsf{L}}_2[i]$ then
12:             $\overline{\mathsf{L}}_{new} \leftarrow \overline{\mathsf{L}}_{new}.add(E)$
13:             $emptyBricksAdded \leftarrow emptyBricksAdded + 1$
14:         else
15:             $\overline{\mathsf{L}}_{new} \leftarrow \overline{\mathsf{L}}_{new}.add(\overline{\mathsf{L}}_1[0])$
16:             $removeHead(\overline{\mathsf{L}}_1)$
17:         end if
18:     end for
19:     return $\overline{\mathsf{L}}_{new}$
20: end function
```

The purpose of Algorithm 1 is to build a new list $\overline{\mathsf{L}}_{new}$ which has the same length of $\overline{\mathsf{L}}_2$ (assuming it is the longest one) and contains all bricks of $\overline{\mathsf{L}}_1$ plus some empty bricks E, trying to maximize the positional correspondences of equal bricks in $\overline{\mathsf{L}}_{new}$ and $\overline{\mathsf{L}}_2$. To do this, we process each brick b of $\overline{\mathsf{L}}_2$ (`for` loop at line 7) and, in the same position of $\overline{\mathsf{L}}_{new}$ we put:

- an empty brick E if $\overline{\mathsf{L}}_1$ is empty (i.e., we have already inserted all its bricks in $\overline{\mathsf{L}}_{new}$) or if b and the first brick of $\overline{\mathsf{L}}_1$ are different (lines 11-13);
- b itself, if the first brick of $\overline{\mathsf{L}}_1$ is equal to b. In this case, we also remove the first brick from $\overline{\mathsf{L}}_1$, to avoid inserting it multiple times in the new list. (lines 14-16)

When the empty bricks have all been added (i.e., $emptyBricksAdded \geq n$), we proceed to insert in $\overline{\mathsf{L}}_{new}$ all remaining bricks in $\overline{\mathsf{L}}_1$, one at a time (lines 8-10).

This padding is particularly useful in order to maximize the number of bricks in the two lists that are equals and at the same position. For instance, consider the case $\overline{\mathsf{L}}_1 = [b_0; b_1; b_2]$ and $\overline{\mathsf{L}}_2 = [b_3; b_0; b_1; b_4; b_5]$. The result of the padding is $\overline{\mathsf{L}}_{new} = [E; b_0; b_1; E; b_2]$. We managed to put

b_0 and b_1 in the same position as they appear in $\overline{\mathsf{L}}_2$. Thanks to this feature, the lattice and semantic operator will be in position to obtain precise results traversing the list of bricks only once.

Partial order To define an order on *lists* of bricks, we have first to define a partial order on *single* bricks. $\leq_{\overline{\mathcal{B}}}$ is defined as follows:

$$[C_1]^{\mathsf{m}_1,\mathsf{M}_1} \leq_{\overline{\mathcal{B}}} [\overline{\mathsf{C}}_2]^{\mathsf{m}_2,\mathsf{M}_2}$$
$$\Updownarrow$$
$$(\overline{\mathsf{C}}_1 \subseteq \overline{\mathsf{C}}_2 \wedge \mathsf{m}_1 \geq \mathsf{m}_2 \wedge \mathsf{M}_1 \leq \mathsf{M}_2) \vee ([\overline{\mathsf{C}}_2]^{\mathsf{m}_2,\mathsf{M}_2} = \top_{\overline{\mathcal{B}}}) \vee ([\overline{\mathsf{C}}_1]^{\mathsf{m}_1,\mathsf{M}_1} = \bot_{\overline{\mathcal{B}}})$$

where $\top_{\overline{\mathcal{B}}}$ and $\bot_{\overline{\mathcal{B}}}$ are two special bricks, greater and smaller than any other brick, respectively.

Given two lists $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, we augment the shorter list using Algorithm 1 in order to have lists of the same size. Then, we proceed by extracting one brick from each list and comparing the two bricks, until we reach the end of the two lists.

Formally, given two lists $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, we make them have the same size n by applying Algorithm 1, thus obtaining $\overline{\mathsf{L}}'_1$ and $\overline{\mathsf{L}}'_2$. Then:

$$\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \Leftrightarrow (\overline{\mathsf{L}}_2 = \top_{\overline{\mathcal{BR}}}) \vee (\overline{\mathsf{L}}_1 = \bot_{\overline{\mathcal{BR}}}) \vee (\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}'_1[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}'_2[\mathsf{i}])$$

Lemma 4.9 ($\leq_{\overline{\mathcal{BR}}}$ is a partial order)
The order $\leq_{\overline{\mathcal{BR}}}$ is a partial order.

Proof
We refer to [6] for the proofs that $\leq_{\overline{\mathcal{BR}}}$ is reflexive and transitive, and here we prove that it is antisymmetric. Formally, we must prove that, given two lists of bricks $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ of the same length n (otherwise we add empty bricks inside the shorter one through Algorithm 1, without changing the represented set of strings), it holds:

$$\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \wedge \overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1 \Rightarrow \overline{\mathsf{L}}_1 = \overline{\mathsf{L}}_2$$

This trivially holds if one of the two abstract states is $\top_{\overline{\mathcal{BR}}}$ or $\bot_{\overline{\mathcal{BR}}}$ by definition of $\leq_{\overline{\mathcal{BR}}}$. Otherwise, since $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$, we know that $\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}_1[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_2[\mathsf{i}]$ by definition of $\leq_{\overline{\mathcal{BR}}}$. But we also know that $\overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1$, and this means that $\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}_2[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_1[\mathsf{i}]$. Consider then a generic pair of bricks $\overline{\mathsf{L}}_1[\mathsf{i}]$ and $\overline{\mathsf{L}}_2[\mathsf{i}]$. Neither of these two bricks can be equal to $\bot_{\overline{\mathcal{B}}}$, since otherwise the abstract state to which it belongs would be equal to $\bot_{\overline{\mathcal{BR}}}$ and we already excluded this case. If one brick is equal to $\top_{\overline{\mathcal{B}}}$, then also the other one must be too, otherwise our hypothesis would not hold. In this case, then, the two bricks are equal. Otherwise (neither brick is top nor bottom), let $\overline{\mathsf{L}}_1[\mathsf{i}] = [C_1]^{\mathsf{m}_1,\mathsf{M}_1}$ and $\overline{\mathsf{L}}_2[\mathsf{i}] = [C_2]^{\mathsf{m}_2,\mathsf{M}_2}$. Since $\overline{\mathsf{L}}_1[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_2[\mathsf{i}]$, it holds that $(\overline{\mathsf{C}}_1 \subseteq \overline{\mathsf{C}}_2 \wedge \mathsf{m}_1 \geq \mathsf{m}_2 \wedge \mathsf{M}_1 \leq \mathsf{M}_2)$. Also, since $\overline{\mathsf{L}}_2[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_1[\mathsf{i}]$, it holds that $(\overline{\mathsf{C}}_2 \subseteq \overline{\mathsf{C}}_1 \wedge \mathsf{m}_2 \geq \mathsf{m}_1 \wedge \mathsf{M}_2 \leq \mathsf{M}_1)$. Then: (i) from $\overline{\mathsf{C}}_1 \subseteq \overline{\mathsf{C}}_2$ and $\overline{\mathsf{C}}_2 \subseteq \overline{\mathsf{C}}_1$ it follows $\overline{\mathsf{C}}_1 = \overline{\mathsf{C}}_2$; (ii) from $\mathsf{m}_1 \geq \mathsf{m}_2$ and $\mathsf{m}_2 \geq \mathsf{m}_1$ it follows $\mathsf{m}_1 = \mathsf{m}_2$; (iii) from $\mathsf{M}_1 \leq \mathsf{M}_2$ and $\mathsf{M}_2 \leq \mathsf{M}_1$ it follows $\mathsf{M}_1 = \mathsf{M}_2$. This means that $\overline{\mathsf{L}}_1[\mathsf{i}] = \overline{\mathsf{L}}_2[\mathsf{i}]$, and this is valid for all $i \in [0, \mathsf{n}-1]$. This implies that $\overline{\mathsf{L}}_1 = \overline{\mathsf{L}}_2$. □

Given an alphabet of characters K, we define the top element $\top_{\overline{\mathcal{B}}}$ as the brick $[\mathsf{K}]^{(0,+\infty)}$. Instead, the bottom element is defined by $\bot_{\overline{\mathcal{B}}} = [\emptyset]^{(m,M)\neq(0,0)} \vee ([\overline{\mathsf{S}}]^{(m,M)} \wedge M < m) \vee [\overline{\mathsf{S}} \neq \emptyset]^{(0,0)}$. The three possible definitions are all bricks which do not represent any string. They are invalid bricks and they correspond to $\emptyset$. Note that $[\emptyset]^{(0,0)}$ is a valid brick which corresponds only to the empty string ϵ.

The top element of $\overline{\mathcal{BR}}$ ($\top_{\overline{\mathcal{BR}}}$) is then a list containing only one brick, $\top_{\overline{\mathcal{B}}}$. Since $\top_{\overline{\mathcal{B}}}$ represents all the strings, $\top_{\overline{\mathcal{BR}}}$ does too. The bottom element $\bot_{\overline{\mathcal{BR}}}$ is an empty list (it does not represent any string at all, not even the empty string) or any list which contains at least one invalid element ($\bot_{\overline{\mathcal{B}}}$).

The lattice of $\overline{\mathcal{BR}}$ is depicted in Figure 4. For visual clarity we only pictured lists of size one and we considered the alphabet $\mathsf{K} = \{a, b\}$.

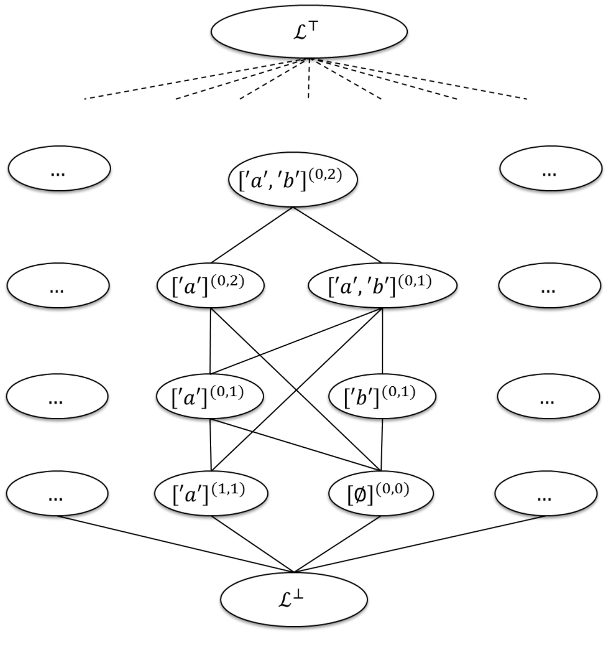


Figure 4. The abstract domain $\overline{\mathcal{BR}}$ with $\mathsf{K} = \{a, b\}$

Least upper bound and greatest lower bound As we did for the partial order, we define the least upper bound operator on single bricks at first:

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

where $m = \min(\mathsf{m}_1, \mathsf{m}_2)$ and $M = \max(\mathsf{M}_1, \mathsf{M}_2)$. For example, the least upper bound between $[\{a, b\}]^{(1,3)}$ and $[\{a, c\}]^{(0,2)}$ is the brick $[\{a, b, c\}]^{(0,3)}$.

To compute the least upper bound between elements of $\overline{\mathcal{BR}}$ (lists of bricks), we proceed exactly as we did to define the partial order $\leq_{\overline{\mathcal{BR}}}$. Given two lists $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, we make them have the same size n by using Algorithm 1, thus obtaining $\overline{\mathsf{L}}_1'$ and $\overline{\mathsf{L}}_2'$. Then, $\sqcup_{\overline{\mathcal{BR}}}$ is defined as follows:

$$\bigsqcup_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = \bigsqcup_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1', \overline{\mathsf{L}}_2') = \overline{\mathsf{L}}_R[0]\overline{\mathsf{L}}_R[1] \ldots \overline{\mathsf{L}}_R[\mathsf{n}-1]$$

where $\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}_R[\mathsf{i}] = \bigsqcup_{\overline{\mathcal{B}}}(\overline{\mathsf{L}}_1'[\mathsf{i}], \overline{\mathsf{L}}_2'[\mathsf{i}])$.

The greatest lower bound operator works very similarly to the least upper bound one. The glb operator on single bricks is defined as follows:

$$\bigsqcap_{\overline{\mathcal{B}}}([\overline{\mathsf{S}}_1]^{(\mathsf{m}_1,\mathsf{M}_1)}, [\overline{\mathsf{S}}_2]^{(\mathsf{m}_2,\mathsf{M}_2)}) = [\overline{\mathsf{S}}_1 \cap \overline{\mathsf{S}}_2]^{(m,M)}$$

where $m = \max(\mathsf{m}_1, \mathsf{m}_2)$ and $M = \min(\mathsf{M}_1, \mathsf{M}_2)$. For example, the greatest lower bound between $[\{a, b\}]^{(1,3)}$ and $[\{a, c\}]^{(0,2)}$ is the brick $[\{a\}]^{(1,2)}$. Note that sometimes the result of the glb is an invalid brick (for example because the max index is smaller than the min one). To conclude the description of the glb operator, we have to define how it works with lists of bricks (the elements of $\overline{\mathcal{BR}}$). Given two lists $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, we make them have the same size n by using Algorithm 1, thus obtaining $\overline{\mathsf{L}}_1'$ and $\overline{\mathsf{L}}_2'$. Then $\sqcap_{\overline{\mathcal{BR}}}$ is defined as follows:

$$\bigsqcap_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = \bigsqcap_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1', \overline{\mathsf{L}}_2') = \overline{\mathsf{L}}_R[0]\overline{\mathsf{L}}_R[1] \ldots \overline{\mathsf{L}}_R[\mathsf{n}-1]$$

where $\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}_R[\mathsf{i}] = \sqcap_{\overline{\mathcal{B}}}(\overline{\mathsf{L}}_1'[\mathsf{i}], \overline{\mathsf{L}}_2'[\mathsf{i}])$. If any element of the sequence $\overline{\mathsf{L}}_R$ corresponds to $\perp_{\overline{\mathcal{B}}}$, then the entire resulting list should be set to $\perp_{\overline{\mathcal{BR}}}$.

Lemma 4.10
$\sqcup_{\overline{\mathcal{BR}}}$ is the least upper bound operator.

Proof
Let $\overline{\mathsf{L}}$ be $\overline{\mathsf{L}} = \overline{\mathsf{L}}_1 \sqcup_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$. Then we have to prove the following two conditions:

1. $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}} \wedge \overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}$. Let us suppose that $\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2, \overline{\mathsf{L}}$ are lists of the same size (one of them could be padded with empty bricks inside it, but empty bricks do not interfere with order comparisons). Then, for $\overline{\mathsf{L}}_1$ to be smaller than $\overline{\mathsf{L}}$, it must be that each brick of $\overline{\mathsf{L}}_1$ is smaller (in the single brick order) than the corresponding brick of $\overline{\mathsf{L}}$. Let $[\overline{\mathsf{S}}_1]^{(\mathsf{m}_1,\mathsf{M}_1)}$ be the brick of $\overline{\mathsf{L}}_1$ in a generic position i, and $[\overline{\mathsf{S}}_2]^{(\mathsf{m}_2,\mathsf{M}_2)}$ be the brick of $\overline{\mathsf{L}}_2$ in the same position. The brick of $\overline{\mathsf{L}}$ in such position will be, by definition of $\sqcup_{\overline{\mathcal{BR}}}$, $[\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2]^{(\min(\mathsf{m}_1,\mathsf{m}_2),\max(\mathsf{M}_1,\mathsf{M}_2))}$. The brick of $\overline{\mathsf{L}}_1$ is smaller than the brick of $\overline{\mathsf{L}}$, because $\overline{\mathsf{S}}_1 \subseteq (\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2) \wedge \mathsf{m}_1 \geq \min(\mathsf{m}_1, \mathsf{m}_2) \wedge \mathsf{M}_1 \leq \max(\mathsf{M}_1, \mathsf{M}_2)$. The same goes for the brick of $\overline{\mathsf{L}}_2$. Thus, $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}} \wedge \overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}$.
2. $\overline{\mathsf{L}} \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}' \ \forall$ upper bound $\overline{\mathsf{L}}'$ of $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$. As before, suppose that $\overline{\mathsf{L}}$, $\overline{\mathsf{L}}_1$, $\overline{\mathsf{L}}_2$, and $\overline{\mathsf{L}}'$ have all the same size (otherwise we pad them with empty bricks using Algorithm 1). Since $\overline{\mathsf{L}}'$ is an upper bound of $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, this means that each brick of $\overline{\mathsf{L}}'$ is greater than the corresponding brick of both $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$. Let $[\overline{\mathsf{S}}_1]^{(\mathsf{m}_1,\mathsf{M}_1)}$ be the brick of $\overline{\mathsf{L}}_1$ in a generic position i, and $[\overline{\mathsf{S}}_2]^{(\mathsf{m}_2,\mathsf{M}_2)}$ be the brick of $\overline{\mathsf{L}}_2$ in the same position. Then, the corresponding brick of $\overline{\mathsf{L}}'$ (let it be $[\overline{\mathsf{S}}']^{(\mathsf{m}',\mathsf{M}')}$) must satisfy (to be an upper bound) the following requirements: (i) $\overline{\mathsf{S}}' \supseteq (\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2)$, (ii) $\mathsf{m}' \leq \min(\mathsf{m}_1, \mathsf{m}_2)$, (iii) $\mathsf{M}' \geq \max(\mathsf{M}_1, \mathsf{M}_2)$. The brick of $\overline{\mathsf{L}}$ in the same position is defined as $[\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2]^{(\min(\mathsf{m}_1,\mathsf{m}_2),\max(\mathsf{M}_1,\mathsf{M}_2))}$ and it is certainly smaller (or equal) than the brick of $\overline{\mathsf{L}}'$, for definition of $\leq_{\overline{\mathcal{B}}}$. Since this happens for every brick of $\overline{\mathsf{L}}'$ and $\overline{\mathsf{L}}$, it holds that $\overline{\mathsf{L}} \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}'$.

□

Lemma 4.11
$\sqcap_{\overline{\mathcal{BR}}}$ is the greatest lower bound operator.

Proof
The reasoning is symmetrical to that of the least upper bound (set intersection instead of union, min instead of max, and so on). In the special case where the glb corresponds to $\perp_{\overline{\mathcal{BR}}}$, it is immediate to prove the two conditions (since $\perp_{\overline{\mathcal{BR}}}$ is smaller than any other element of the domain, and if the glb is $\perp_{\overline{\mathcal{BR}}}$ it cannot exist any other valid lower bound). □

Lemma 4.12
$\overline{\mathcal{BR}}$ is a complete lattice.

Proof
A complete lattice is a partially ordered set in which all subsets have both a join and a meet. We already proved that the order $\leq_{\overline{\mathcal{BR}}}$ is a partial order (Theorem 4.9). We just need to prove that every subset of abstract elements of $\overline{\mathcal{BR}}$ have both a meet and a join. Let $L = \{\overline{\mathsf{L}}_1, \ldots, \overline{\mathsf{L}}_N\}$ be a set of abstract elements in $\overline{\mathcal{BR}}$. Their meet (greatest lower bound) is a list of bricks which has the same size n as that of the longest list in L and which bricks $\overline{\mathsf{B}}_i (\forall i \in [1; n])$ are defined as $\overline{\mathsf{B}}_i = [\bigcap_{j=1}^{N} \overline{\mathsf{S}}_{ij}]^{(\max_{j=1}^{N} m_{ij}, \min_{j=1}^{N} M_{ij})}$, where $[\overline{\mathsf{S}}_{ij}]^{(m_{ij}, M_{ij})}$ is the i-th brick of the list $\overline{\mathsf{L}}_j$. If one of the lists is shorter than n, it is augmented with empty bricks through Algorithm 1. If any of the resulting bricks is invalid, the glb becomes bottom.

Their join (least upper bound) is a list of bricks which has the same size n as that of the longest list in L (the other lists are padded with empty bricks through Algorithm 1 to have the same size) and which bricks $\overline{\mathsf{B}}_i (\forall i \in [1; n])$ are defined as $\overline{\mathsf{B}}_i = [\bigcup_{j=1}^{N} \overline{\mathsf{S}}_{ij}]^{(\min_{j=1}^{N} m_{ij}, \max_{j=1}^{N} M_{ij})}$, where $[\overline{\mathsf{S}}_{ij}]^{(m_{ij}, M_{ij})}$ is the i-th brick of the list $\overline{\mathsf{L}}_j$. Theorems 4.10 and 4.11 proved that $\sqcup_{\overline{\mathcal{BR}}}$ and $\sqcap_{\overline{\mathcal{BR}}}$ are the least upper bound and the greatest lower bound operators, respectively. □

Widening operator Let k_L, k_I and k_S be three constant integer values which will bound, respectively, the length of a bricks list, the indices range of a brick and the number of strings in the set of a brick. The widening operator is defined as follows:

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

We return the $\top_{\overline{\mathcal{BR}}}$ element of our domain in two cases: (i) if the two abstract values are not comparable with respect to our order ($\overline{\mathsf{L}}_1 \not\leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \wedge \overline{\mathsf{L}}_2 \not\leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1$), or (ii) if the length of one of the two lists is greater than the constant k_L ($\exists \mathsf{i} \in [1,2] : len(\overline{\mathsf{L}}_i) > \mathsf{k}_L$). Otherwise, we return $w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$. Now we have to define what the function w does. Let us assume that $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$ and that $len(\overline{\mathsf{L}}_1) = len(\overline{\mathsf{L}}_2) = \mathsf{n}$. If the two lists were not of the same length, we could always add a proper number of empty bricks inside the shorter list using Algorithm 1. The definition of w is thus the following one:

$$w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = [\overline{\mathcal{B}}_0^{\mathsf{new}}(\overline{\mathsf{L}}_1[0], \overline{\mathsf{L}}_2[0]); \overline{\mathcal{B}}_1^{\mathsf{new}}(\overline{\mathsf{L}}_1[1], \overline{\mathsf{L}}_2[1]); \ldots; \overline{\mathcal{B}}_{\mathsf{n}-1}^{\mathsf{new}}(\overline{\mathsf{L}}_1[\mathsf{n}-1], \overline{\mathsf{L}}_2[\mathsf{n}-1])]$$

where $\overline{\mathcal{B}}_i^{\mathsf{new}}(\overline{\mathsf{L}}_1[\mathsf{i}], \overline{\mathsf{L}}_2[\mathsf{i}])$ is defined by:

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

where $\mathsf{m} = \min(\mathsf{m}_{1i}, \mathsf{m}_{2i})$ and $\mathsf{M} = \max(\mathsf{M}_{1i}, \mathsf{M}_{2i})$.

Let us briefly explain why this widening operator is correct. First of all, the result of a widening between two values must be greater or equal than both values. In our domain, the result of the widening between $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ can be $\top_{\overline{\mathcal{BR}}}$ or $w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$. If it is $\top_{\overline{\mathcal{BR}}}$, $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \top_{\overline{\mathcal{BR}}}$ and $\overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \top_{\overline{\mathcal{BR}}}$ follows from the fact that $\top_{\overline{\mathcal{BR}}}$ is the top element of $\overline{\mathcal{BR}}$. In the other case, we know that $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$ or viceversa (for argument's sake, we assume that $\overline{\mathsf{L}}_1$ is the smaller value). Thus, the result of the widening is a new list in which each element $\overline{\mathcal{B}}_i^{\mathsf{new}}$ is the combination of $\overline{\mathsf{L}}_1[i]$ and $\overline{\mathsf{L}}_2[i]$. By definition of $\leq_{\overline{\mathcal{BR}}}$, to prove that $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \nabla_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$ and $\overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \nabla_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$ we just need to prove that $\overline{\mathsf{L}}_1[i] \leq_{\overline{\mathcal{B}}} \overline{\mathcal{B}}_i^{\mathsf{new}}$ and $\overline{\mathsf{L}}_2[i] \leq_{\overline{\mathcal{B}}} \overline{\mathcal{B}}_i^{\mathsf{new}} \forall i \in [0, \mathsf{n}-1]$, that is, that each brick of the result is greater or equal to the two corresponding bricks in $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$. By definition of $\overline{\mathcal{B}}_i^{\mathsf{new}}$ we have only three cases: (i) $\overline{\mathcal{B}}_i^{\mathsf{new}} = \top_{\overline{\mathcal{B}}}$, and so we have that $\overline{\mathsf{L}}_1[i] \leq_{\overline{\mathcal{B}}} \top_{\overline{\mathcal{B}}}$ and $\overline{\mathsf{L}}_2[i] \leq_{\overline{\mathcal{B}}} \top_{\overline{\mathcal{B}}}$ by definition of $\top_{\overline{\mathcal{B}}}$; (ii) $\overline{\mathcal{B}}_i^{\mathsf{new}} = [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(0,\infty)}$; in this case $\overline{\mathsf{L}}_1[i] = [\overline{\mathsf{S}}_{1i}]^{\mathsf{m}_{1i}, \mathsf{M}_{1i}} \leq_{\overline{\mathcal{B}}} [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(0,\infty)}$ since $\overline{\mathsf{S}}_{1i} \subseteq (\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}) \wedge 0 \leq \mathsf{m}_{1i} \wedge \mathsf{M}_{1i} \leq +\infty$. The same happens for $\overline{\mathsf{L}}_2[i]$; (iii) $\overline{\mathcal{B}}_i^{\mathsf{new}} = [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(\mathsf{m},\mathsf{M})}$ where $\mathsf{m} = \min(\mathsf{m}_{1i}, \mathsf{m}_{2i})$ and $\mathsf{M} = \max(\mathsf{M}_{1i}, \mathsf{M}_{2i})$. In this case we have $\overline{\mathsf{L}}_1[i] = [\overline{\mathsf{S}}_{1i}]^{\mathsf{m}_{1i}, \mathsf{M}_{1i}} \leq_{\overline{\mathcal{B}}} [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{\mathsf{m},\mathsf{M}}$ because $\overline{\mathsf{S}}_{1i} \subseteq (\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}) \wedge \mathsf{m} = \min(\mathsf{m}_{1i}, \mathsf{m}_{2i}) \leq \mathsf{m}_{1i} \wedge \mathsf{M}_{1i} \leq \mathsf{M} = \max(\mathsf{M}_{1i}, \mathsf{M}_{2i})$. The same happens for $\overline{\mathsf{L}}_2[i]$.

Then, we need the widening operator to be convergent. In other words, given an ascending chain $\overline{\mathsf{s}_\mathsf{n}}$, the sequence ($\overline{\mathsf{t}_{\mathsf{n}+1}} = \nabla_{\overline{\mathcal{BR}}}(\overline{\mathsf{t}_\mathsf{n}}, \overline{\mathsf{s}_\mathsf{n}})$) has to be ultimately stationary. In our case, 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 k_L,k_S, and k_I. After the list has reached k_L elements, the entire abstract value is approximated to $\top_{\overline{\mathcal{BR}}}$, stopping its possible growth altogether. If the range of a certain brick becomes larger than k_I, the range is approximated to $(0, +\infty)$, stopping the indices possible growth. Finally, if the strings set of a certain brick reaches k_S elements, the brick is approximated to $\top_{\overline{\mathcal{B}}}$, stopping its possible growth altogether.

Concretization function The concretization function maps an abstract element (i.e., a list of bricks) to a concrete element (i.e., a set of strings). Each brick represents a certain set of strings. The list of bricks thus represents all the strings built through the concatenation of strings which can be made from the bricks of the list (taken in the correct order). More formally, we define the strings represented by a single brick as:

$$\gamma_{\overline{\mathcal{B}}}(\overline{\mathsf{B}}) = \gamma_{\overline{\mathcal{B}}}([\overline{\mathsf{S}}]^{(m,M)}) = \bigcup_{j=m}^{M}(\underbrace{\overline{\mathsf{S}}\overline{\mathsf{S}}\ldots\overline{\mathsf{S}}}_{\text{j times}})$$

where $\underbrace{\overline{\mathsf{S}}\overline{\mathsf{S}}\ldots\overline{\mathsf{S}}}_{\text{j times}} = \overline{\mathsf{S}}^{j}$ stands for the concatenation between sets of strings (in particular, we concatenate $\overline{\mathsf{S}}$ to itself j times). To account for the case in which $j = 0$, we impose $\overline{\mathsf{S}}^{0} = \{\epsilon\}$.

Let us see an example to clarify this definition. Consider the brick $[\{a, b\}]^{(1,3)}$ and let $\overline{\mathsf{S}} = \{a, b\}$. Then, the concretization of such brick is the following one:

$$\begin{aligned}\gamma_{\overline{\mathcal{B}}}([\{a, b\}]^{(1,3)}) &= \overline{\mathsf{S}} \cup \overline{\mathsf{S}\mathsf{S}} \cup \overline{\mathsf{S}\mathsf{S}\mathsf{S}} = \\ &= \{a, b\} \cup \{aa, ab, ba, bb\} \cup \{aaa, aab, aba, abb, baa, bab, bba, bbb\} = \\ &= \{a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\}\end{aligned}$$

Note that, if min had been 0 instead of 1, the result would have been:

$$\gamma_{\overline{\mathcal{B}}}([\{a, b\}]^{(0,3)}) = \overline{\mathsf{S}}^{0} \cup \overline{\mathsf{S}}^{1} \cup \overline{\mathsf{S}}^{2} \cup \overline{\mathsf{S}}^{3} = \epsilon \cup \overline{\mathsf{S}} \cup \overline{\mathsf{S}\mathsf{S}} \cup \overline{\mathsf{S}\mathsf{S}\mathsf{S}}$$

The concretization function for lists of bricks is then the following one:

$$\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}}_0\overline{\mathsf{B}}_1\ldots\overline{\mathsf{B}}_{N-1}) = \{s : s \in \mathsf{K}^* \wedge s = b_0 + b_1 + \cdots + b_{N-1} \wedge \forall i \in [0, N-1] : b_i \in \gamma_{\overline{\mathcal{B}}}(\overline{\mathsf{B}}_i)\}$$

where "$+$" represents the operator of string concatenation.

Theorem 4.13
Let the abstraction function $\alpha_{\overline{\mathcal{BR}}}$ be defined by $\alpha_{\overline{\mathcal{BR}}} = \lambda \mathsf{Y}.\sqcap_{\overline{\mathcal{BR}}} \{\overline{\mathsf{B}} : \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}}) \subseteq \mathsf{Y}\}$.
Then $\langle \wp(\mathsf{S}), \subseteq\rangle \underset{\alpha_{\overline{\mathcal{BR}}}}{\overset{\gamma_{\overline{\mathcal{BR}}}}{\leftrightarrows}} \langle \overline{\mathcal{BR}}, \leq_{\overline{\mathcal{BR}}}\rangle$.

Proof
By Theorem 1.1 we only need to prove that $\gamma_{\overline{\mathcal{BR}}}$ is a complete meet morphism. Formally, we have to prove that $\gamma_{\overline{\mathcal{BR}}}(\bigsqcap_{\substack{\mathcal{BR}\\(\overline{\mathsf{B}})\in\overline{\mathsf{X}}}} \overline{\mathsf{B}}) = \bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}} \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}})$. For the sake of simplicity, we suppose that all list of bricks in $\overline{\mathsf{X}}$ contain n bricks.

$\gamma_{\overline{\mathcal{BR}}}(\bigsqcap_{\substack{\mathcal{BR}\\(\overline{\mathsf{B}})\in\overline{\mathsf{X}}}} \overline{\mathsf{B}}) =$

By definition of $\sqcap_{\overline{\mathcal{BR}}}$

$= \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}}') : \forall i \in [0..n-1] : \overline{\mathsf{B}}'[i] = [\bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}},\overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} \overline{\mathsf{S}}]^{(\max_{\overline{\mathsf{B}}\in\overline{\mathsf{X}},\overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} m, \min_{\overline{\mathsf{B}}\in\overline{\mathsf{X}},\overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} M)}$

By definition of $\gamma_{\overline{\mathcal{BR}}}$

$= \{b_0 + .. + b_{n-1} : \forall i \in [0..n-1] : i_1 = \max_{\overline{\mathsf{B}}\in\overline{\mathsf{X}},\overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} m, i_2 = \min_{\overline{\mathsf{B}}\in\overline{\mathsf{X}},\overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} M,$
$b_i \in \bigcup_{j=i_1}^{i_2} (\bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}},\overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} \overline{\mathsf{S}})^j\}$

By definition of $\cap, \min, \max$

$= \{b_0 + .. + b_{n-1} : \forall i \in [0..n-1] : \forall \overline{\mathsf{B}} \in \overline{\mathsf{X}} : \overline{\mathsf{B}}[i] = [\overline{\mathsf{S}}]^{(m,M)}, b_i \in \bigcup_{j=m}^{M} \overline{\mathsf{S}}^{j}\}$

By definition of $\cap$

$= \bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}}\{b_0 + .. + b_{n-1} : \forall i \in [0..n-1] : \overline{\mathsf{B}}[i] = [\overline{\mathsf{S}}]^{(m,M)}, b_i \in \bigcup_{j=m}^{M} \overline{\mathsf{S}}^{j}\}$

By definition of $\gamma_{\overline{\mathcal{BR}}}$

$= \bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}} \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}})$

□

Now that we presented the concretization function, we can prove that the normalization of a list of bricks does not change its concretization, i.e. the set of strings it represents.

Lemma 4.14 (Soundness of the normalization rules)
Given a normalization rule r_i ($i \in [1,5]$) and a list of bricks $\overline{\mathsf{L}}$, suppose that $\overline{\mathsf{L}}'$ is the list of bricks resulting from the application of r_i to $\overline{\mathsf{L}}$. Then, $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}) = \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}')$.

Proof
We will prove the theorem for one rule at a time.

- r_1: trivial, since it just removes empty bricks which represent the empty string, i.e., the neutral element of concatenation.
- r_2: let $\overline{\mathsf{B}_1} = [S_1]^{(1,1)}$ and $\overline{\mathsf{B}_2} = [S_2]^{(1,1)}$ be the two bricks which Rule 2 merges. The first brick represents the strings in S_1 (since its indices are both 1), while the second represents, for the same reasons, the strings in S_2. The concatenation of these two bricks, then, represents the strings set $S_1\,S_2$ (remember that $S\,T$ represents the concatenation between the two strings sets S and T, i.e. the set containing all strings which can be obtained by concatenating a string from S and a string from T, in this order). Rule 2 transforms these two bricks in $\overline{\mathsf{B}}' = [S_1\,S_2]^{(1,1)}$, which represents exactly the same set of strings as the two original bricks, i.e. $S_1\,S_2$.
- r_3: let $\overline{\mathsf{B}} = [\mathsf{S}]^{(m,m)}$ be the brick which Rule 3 modifies. Its concretization is $\bigcup_{j=m}^{m}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}}) = (\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{m times}}) = \mathsf{S}^m$. Rule 3 transforms such brick in $\overline{\mathsf{B}}' = [\mathsf{S}^m]^{(1,1)}$, which concretization is S^m, exactly the same as the original one.
- r_4: let $\overline{\mathsf{B}_1} = [\mathsf{S}]^{(m_1,M_1)}$ and $\overline{\mathsf{B}_2} = [\mathsf{S}]^{(m_2,M_2)}$ be the two bricks which Rule 4 merges. Their concretization is, respectively, $\bigcup_{j=m_1}^{M_1}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$ and $\bigcup_{j=m_2}^{M_2}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$. The concatenation of these two bricks represents the concatenation of their concretizations: $C_1 = \{(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{m_1\text{ times}}),\ldots,(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{M_1\text{ times}})\}$ concatenated to $C_2 = \{(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{m_2\text{ times}}),\ldots,(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{M_2\text{ times}})\}$. The result is the set $\{\mathsf{S}_1\mathsf{S}_2 : \mathsf{S}_1 \in C_1 \wedge \mathsf{S}_2 \in C_2\} = \{\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{j_1\text{ times}}\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{j_2\text{ times}} : j_1 \in [m_1,M_1] \wedge j_2 \in [m_2,M_2]\} = \{\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}} : j \in [m_1+m_2, M_1+M_2]\}$. Rule 4 merges these two bricks into the single brick $\overline{\mathsf{B}} = [\mathsf{S}]^{(m_1+m_2,M_1+M_2)}$, which concretization is $\bigcup_{j=(m_1+m_2)}^{M_1+M_2}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$, exactly the same of the original one.
- r_5: let $\overline{\mathsf{B}} = [\mathsf{S}]^{(m,M)}$ be the brick which is split by Rule 5, where $m \geq 1 \wedge M \neq m$. Its concretization is $\bigcup_{j=m}^{M}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$. Rule 5 transforms such brick in the concatenation of the two bricks $\overline{\mathsf{B}_1} = [\mathsf{S}^m]^{(1,1)}$ and $\overline{\mathsf{B}_2} = [\mathsf{S}]^{(0,M-m)}$. Their concretizations are, respectively, $C_1 = \mathsf{S}^m$ and $C_2 = \bigcup_{j=0}^{M-m}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$. The concatenation of these two bricks produces the set of strings $\{\mathsf{S}_1\mathsf{S}_2 : \mathsf{S}_1 = \mathsf{S}^m \wedge \mathsf{S}_2 \in C_2\} = \{\mathsf{S}^m\,\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}} : j \in [0, M-m]\} = \{\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}} : j \in [0+m, M-m+m]\} = \bigcup_{j=m}^{M}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$.

□

Gaining more precision Working with normalized values is important to guarantee the convergence of operators like the least upper bound and the widening. However, normalizing values after each operation is costly and, worse than that, it entails a big loss of precision (which we documented while analysing our case studies). For example, the result of the $\overline{\mathcal{BR}}$ domain on the second case study

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

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

(prog2), when normalizing values after *each* operation, is $\top_{\overline{\mathcal{BR}}}$: we are not able to track any kind of information on the program. For these two reasons (performance and, most importantly, precision), we choose to normalize abstract values only *after* executing the least upper bound operator or the widening operator. Any other operation (regarding both the abstract semantics and the lattice) will not be followed by a normalization step.

Semantics Table VI defines the abstract semantics on $\overline{\mathcal{BR}}$. Let us explain in detail the semantics of each operator.

When a constant string value is evaluated (`new String(str)`), the semantics returns a single brick containing exactly that string with $[1, 1]$ as indices.

For the concatenation of two strings, we rely on the $\overline{\mathit{concatList}}$ function that concatenates two lists of bricks.

To define the semantics of $\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}$, we first normalize the abstract value in input (we can do that since we know, by Lemma 4.14, that the normalization does not change the set of represented strings). Remember that, in a normalized list of bricks, each brick is in the form $[\overline{\mathsf{T}}]^{(0,max>0)}$ or $[\overline{\mathsf{T}}]^{(1,1)}$. If the first brick of the normalized abstract value $\overline{\mathsf{b}}'$ has the form $[\overline{\mathsf{T}}]^{(0,max>0)}$, then we have too much uncertainty on how the string begins: we cannot compute a substring based on start and end indices. Instead, if the first brick has the form $[\overline{\mathsf{T}}]^{(1,1)}$ then we are sure that the string will begin with any of the strings in $\overline{\mathsf{T}}$. If all the strings in $\overline{\mathsf{T}}$ are long enough ($\mathit{len}(\overline{\mathsf{t}}) \geq \mathsf{e}\ \forall \overline{\mathsf{t}} \in \overline{\mathsf{T}}$) we can pack all the possible substrings in a new abstract value, which we will return.

The semantics of $\texttt{contains}_{\mathsf{c}}$ returns true iff the character c appears in all the strings of a certain brick with minimal index $min \geq 1$. It returns false iff we are sure that c does not appear in any string of any brick of the abstract value. Otherwise, we have to return $\top_{\mathsf{B}}$.

Theorem 4.15 (Soundness of the abstract semantics)
$\overline{\mathbb{S}_{\overline{\mathcal{BR}}}}$ and $\overline{\mathbb{B}_{\overline{\mathcal{BR}}}}$ are a sound overapproximation of $\mathbb{S}$ and $\mathbb{B}$, respectively. Formally, $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathbb{S}_{\mathcal{BR}}}\llbracket \mathsf{s} \rrbracket(\overline{\mathsf{L}})) \supseteq \{\mathbb{S}\llbracket \mathsf{s} \rrbracket(\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})\}$ and $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{B}_{\mathcal{BR}}}\llbracket \mathsf{s} \rrbracket(\overline{\mathsf{L}})) \geq_{\mathsf{B}} \{\mathbb{B}\llbracket \mathsf{s} \rrbracket(\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})\}$.

Proof
We prove the soundness separately for each operator.

- $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathbb{S}_{\mathcal{BR}}}\llbracket \texttt{new String(str)} \rrbracket()) \supseteq \{\mathbb{S}\llbracket \texttt{new String(str)} \rrbracket()\}$ follows immediately from the definition of $\overline{\mathbb{S}_{\mathcal{BR}}}\llbracket \texttt{new String(str)} \rrbracket()$ and of $\gamma_{\overline{\mathcal{BR}}}$.
- Consider the binary operator concat. Let $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ be two lists of bricks. We have to prove that $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathbb{S}_{\mathcal{BR}}}\llbracket \mathsf{concat} \rrbracket(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)) \supseteq \{\mathbb{S}\llbracket \mathsf{concat} \rrbracket(\mathsf{c}_1, \mathsf{c}_2) : \mathsf{c}_1 \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1) \wedge \mathsf{c}_2 \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_2)\}$. Let s be an element in $\{\mathbb{S}\llbracket \mathsf{concat} \rrbracket(\mathsf{c}_1, \mathsf{c}_2) : \mathsf{c}_1 \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1) \wedge \mathsf{c}_2 \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_2)\}$. By definition of $\mathbb{S}$, this means that there exist two strings $\mathsf{c}_1, \mathsf{c}_2$ such that $\mathsf{s} = \mathsf{c}_1 + \mathsf{c}_2$ and that $\mathsf{c}_1 \in \gamma(\overline{\mathsf{L}}_1) \wedge \mathsf{c}_2 \in \gamma(\overline{\mathsf{L}}_2)$. On the other hand, $\overline{\mathbb{S}_{\mathcal{BR}}}\llbracket \mathsf{concat} \rrbracket(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$ produces a new list of bricks $\overline{\mathsf{L}}$ which concatenates the two lists in input by definition of $\overline{\mathbb{S}_{\overline{\mathcal{BR}}}}$ By the definition of $\gamma_{\overline{\mathcal{BR}}}$ and the associative property of the concatenation between strings, we can say that the

strings belonging $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$ are all the strings obtained through the concatenation of one string belonging to $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1)$ and another belonging to $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_2)$. Then, surely s belongs to $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$.

- Consider the unary operator $\mathtt{substring}_{\mathtt{b}}^{\mathtt{e}}$ and let $\overline{\mathsf{L}}$ be a (normalized) list of bricks. Consider the following cases:
  - if $\overline{\mathsf{L}}[0] = [\overline{\mathsf{T}}]^{1,1} \wedge \forall \overline{\mathsf{t}} \in \overline{\mathsf{T}} : \mathit{len}(\overline{\mathsf{t}}) \geq \mathtt{e}$, then we have that $\gamma_{\overline{\mathcal{B}}}(\overline{\mathsf{L}}[0]) = \overline{\mathsf{T}}$. Thus, all the strings in $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$ have as prefix one of the strings of $\overline{\mathsf{T}}$, by definition of $\gamma_{\overline{\mathcal{BR}}}$. Moreover, by hypothesis all strings of $\overline{\mathsf{T}}$ are longer than $\mathtt{e}$ characters. Then, a string belonging to $\{\mathbb{S}[\![\mathtt{substring}_{\mathtt{b}}^{\mathtt{e}}]\!](\mathtt{c}) : \mathtt{c} \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})\}$ is certainly a substring of one string of $\overline{\mathsf{T}}$, from the $\mathtt{b}$-th character to the $\mathtt{e}$-th character, by definition of $\mathbb{S}$. This corresponds exactly to $\gamma_{\mathcal{BR}}(\mathbb{S}_{\overline{\mathcal{BR}}}[\![\mathtt{substring}_{\mathtt{b}}^{\mathtt{e}}]\!](\overline{\mathsf{L}}))$, since the abstract semantics applied to $\overline{\mathsf{L}}$ produces a single brick containing the substrings of all strings in $\overline{\mathsf{T}}$, from the $\mathtt{b}$-th character to the $\mathtt{e}$-th character.
  - otherwise, the abstract semantics returns $\top_{\overline{\mathcal{BR}}}$, that soundly approximates any possible result of the concrete semantics.
- Consider the unary operator $\mathtt{contains}_{\mathtt{c}}$ and let $\overline{\mathsf{L}}$ be a list of bricks. Regarding the character $\mathtt{c}$, we have three possible cases:
  - if $\exists \overline{\mathsf{b}} \in \overline{\mathsf{L}} : \overline{\mathsf{b}} = [\overline{\mathsf{T}}]^{m,M} \wedge 1 \leq m \leq M \wedge (\forall \overline{\mathsf{t}} \in \overline{\mathsf{T}} : \mathsf{c} \in \mathit{char}(\overline{\mathsf{t}}))$, this means that there exists at least one brick whose strings all contain the character $\mathtt{c}$. Let $\overline{\mathsf{b}}$ be this brick. Then, all the strings belonging to $\gamma_{\overline{\mathcal{B}}}(\overline{\mathsf{b}})$ contain the character $\mathtt{c}$, since its minimum index is ≥ 1. $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$ concatenates all the concretizations of its bricks, so each string belonging to this concretization surely contains the character $\mathtt{c}$. The result of the concrete semantics is, then, always $\mathtt{true}$. Since $\mathbb{B}_{\overline{\mathcal{BR}}}[\![\mathtt{contains}_{\mathtt{c}}]\!](\overline{\mathsf{L}}) = \mathsf{true}$, the abstract semantics is a sound approximation of the concrete semantics.
  - if $\forall [\overline{\mathsf{T}}]^{\mathsf{m},\mathsf{M}} \in \overline{\mathsf{L}}, \forall \overline{\mathsf{t}} \in \overline{\mathsf{T}} : \mathsf{c} \notin \mathit{char}(\overline{\mathsf{t}})$, this means that no brick in the list $\overline{\mathsf{L}}$ has a string containing the character $\mathtt{c}$. Then, no concrete string in $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$ contains such character, and for this reason the concrete semantics always returns $\mathtt{false}$. Since $\mathbb{B}_{\overline{\mathcal{BR}}}[\![\mathtt{contains}_{\mathtt{c}}]\!](\overline{\mathsf{L}}) = \mathsf{false}$, this precisely approximates the results of the concrete semantics.
  - otherwise, the abstract semantics on $\overline{\mathsf{L}}$ returns $\top_{\mathsf{B}}$, and the property is immediately proven, since $\top_{\mathsf{B}}$ is a superset of any set of boolean values.

□

Running Example The results of the analysis of the two running examples using $\overline{\mathcal{BR}}$ are depicted in Figures 5(a) and 5(b).

For `prog1`, at line 1 we represent `query` with a single brick with a singleton set (containing $\mathtt{s}_1$, the string associated to `query`) and indices $\min = \max = 1$. The variable `l` has an unknown value, so it is associated to $\top_{\overline{\mathcal{BR}}}$. At line 4 we concatenate the value of `query` to $\mathtt{s}_2$, `l` and $\mathtt{s}_3$ and we obtain a list of four bricks: the first two are made up by a singleton set (containing, respectively, $\mathtt{s}_1$ and $\mathtt{s}_2$) and indices $\min = \max = 1$, the third one is $\top_{\overline{\mathcal{BR}}}$ (because of `l`), and the fourth one is made up by a singleton set (containing $\mathtt{s}_3$) and indices $\min = \max = 1$. This means that we know that, just after line 4, the string associated to `query` starts with $\mathtt{s}_1 + \mathtt{s}_2$, then it has an unknown part, and then it ends with $\mathtt{s}_3$. Then, we have to compute the lub between the values of `query` after lines 1 and 4. To do this, firstly we use Algorithm 1 to make the two lists have the same size: the algorithm adds three empty bricks at the end of the bricks list of the abstract value at line 1, thus maintaining the correspondence between $[\{\mathtt{s}_1\}]^{1,1}$ in the two lists. The result of the lub is, again, a list of four bricks: the first one is made up by a singleton set (containing $\mathtt{s}_1$) and indices $\min = \max = 1$, the second one is made up by another singleton set (containing $\mathtt{s}_2$) and indices $\min = 0, \max = 1$, the third one is $\top_{\overline{\mathcal{BR}}}$ (because of `l`) and the last one is made up by a singleton set (containing $\mathtt{s}_3$) and indices $\min = 0, \max = 1$. This means that we know that, just after line 5, the string associated to `query` surely starts with $\mathtt{s}_1$, then it *could* continue with $\mathtt{s}_2$, then it has an unknown part and then it *could* end with $\mathtt{s}_3$. At line 7, the abstract value of the variable `per` is composed by a single brick with a singleton set (containing $\mathtt{s}_4$, the string associated to `per`) and indices $\min = \max = 1$. Finally,

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

(a) Analysis of prog1

#I	Var	$\overline{\mathcal{BR}}$
1	x	$[\{``a"\}]^{1,1}$
3	x	$[\{``0"\}]^{0,n}[\{``a"\}]^{1,1}[\{``1"\}]^{0,n}$
4	x	$[\{``0"\}]^{0,+\infty}[\{``a"\}]^{1,1}[\{``1"\}]^{0,+\infty}$

(b) Analysis of prog2

Figure 5. The results of $\overline{\mathcal{BR}}$

at line 8 there is another concatenation. The bricks of the abstract value associated to query after line 8 are seven: (i) the first brick represents the string $\mathtt{s}_1$, (ii) the second brick could be the empty string ϵ or $\mathtt{s}_2$, (iii) the third brick corresponds to the (unknown) input l, (iv) the fourth brick could be the empty string ϵ or $\mathtt{s}_3$, and (v) the last three bricks represent the concatenation of $\mathtt{s}_5$, $\mathtt{s}_4$, and $\mathtt{s}_6$. 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, $\mathtt{s}_1 + \mathtt{s}_3 + \mathtt{s}_5 + \mathtt{s}_4 + \mathtt{s}_6$, which cannot be computed in any execution of the analyzed code.

For prog2, after line 1 the abstract value associated to x is a single brick with a singleton set (containing "a") and indices $\min = \max = 1$. After the first iteration of the loop, the result of the concatenation is made up by three bricks, all of them with a singleton set (containing, respectively, "0", "a" and "1") and indices $\min = \max = 1$. To compute the least upper bound between this value and the value of x *before* the loop ($[\{``a"\}]^{1,1}$) we first execute Algorithm 1, obtaining the new list $E[\{``a"\}]^{1,1}E$ instead of just $[\{``a"\}]^{1,1}$. The result of the lub is then the abstract value $[\{``0"\}]^{0,1}[\{``a"\}]^{1,1}[\{``1"\}]^{0,1}$. The normalization step does not change this abstract value. Starting from this value, we execute the second iteration, and we obtain $[\{``0"\}]^{1,1}[\{``0"\}]^{0,1}[\{``a"\}]^{1,1}[\{``1"\}]^{0,1}[\{``1"\}]^{1,1}$. To compute the least upper bound between the values after the first and second iterations (we do not know how many iterations the loop will do), we apply Algorithm 1 on the shorter list, obtaining the new list $E[\{``0"\}]^{0,1}[\{``a"\}]^{1,1}[\{``1"\}]^{0,1}E$. The result of the lub is then the abstract value $[\{``0"\}]^{0,1}[\{``0"\}]^{0,1}[\{``a"\}]^{1,1}[\{``1"\}]^{0,1}[\{``1"\}]^{0,1}$, which, after the normalization step, becomes $[\{``0"\}]^{0,2}[\{``a"\}]^{1,1}[\{``1"\}]^{0,2}$. Following the same reasoning, after the third iteration we obtain $[\{``0"\}]^{1,1}[\{``0"\}]^{0,2}[\{``a"\}]^{1,1}[\{``1"\}]^{0,2}[\{``1"\}]^{1,1}$ which becomes $[\{``0"\}]^{0,3}[\{``a"\}]^{1,1}[\{``1"\}]^{0,3}$ after the lub with the value of the previous iteration and after the normalization step. We can see that, after each iteration, we obtain an abstract value which first and last bricks have an augmented range with respect to the value in the previous iteration: $\min$ is always zero, but $\max$ increases by one at each iteration. The convergence of the analysis is obtained through to the use of the widening operator, which, when a brick's indices range reaches the threshold k_I, forces the range of the brick to $\min = 0, \max = +\infty$. Since k_I is a constant value, we will certainly reach it after a finite number of iterations. Therefore, after the loop, we associate x to $[\{``0"\}]^{0,+\infty}[\{``a"\}]^{1,1}[\{``1"\}]^{0,+\infty}$. The result is almost optimal: the imprecision is due to the fact the number of occurrences of 0s and 1s are not restricted to be the same. For example, $0a11$ is a concrete value represented by our resulting abstraction, but we know that this string can never be produced by the program prog2.

4.4. String Graphs

In the first domain ($\overline{\mathcal{CI}}$) the only focus of the approximation was character inclusion. In the next two domains ($\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$) we also considered order, but limited at the beginning (prefix) or at the end (suffix) of the string. In the $\overline{\mathcal{BR}}$ domain we considered (like in $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$) both inclusion and order among characters, but this time it was not limited to the beginning or the end of the string. $\overline{\mathcal{BR}}$ approximates a string with a list of bricks, where each brick represents a set of strings. The precision of this domain is definitely better than that of the previous ones, as it was made clear by the analysis of `prog1` and `prog2`. We obtained very good results analysing such programs, even though there is still a little room for improvements. The new abstract domain we are going to present in this Section tracks a kind of information similar to the one tracked by $\overline{\mathcal{BR}}$ (inclusion and order), but equipped with more precise lattice and semantics operators. This domain exploits type graphs [25], a data structure which represents tree automata, and adapts them to represent set of strings. Type graphs were introduced in 1992 by Janssens & Bruynooghe, when they developed a method for obtaining descriptions of possible values of program variables (extended modes or a kind of type information). Their method was based upon a framework for abstract interpretation. Many of the concepts we are going to present about string graphs come from the original definition of type graphs, and we refer the interested reader to [25] for more details about them.

Domain definition A string graph $\overline{\mathsf{T}}$ is a triple $(\overline{\mathsf{N}}, \overline{\mathsf{A}}_F, \overline{\mathsf{A}}_B)$ where $\overline{\mathsf{T}}_r = (\overline{\mathsf{N}}, \overline{\mathsf{A}}_F)$ 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 $\overline{\mathsf{T}}_r$. Ancestors and descendants are defined in the usual way. The backward arcs $(\overline{\mathsf{n}}, \overline{\mathsf{m}})$ in $\overline{\mathsf{A}}_B$, have the property that $\overline{\mathsf{m}}$ belongs to the ancestors of $\overline{\mathsf{n}}$. A forward path is a path composed of forward arcs. The depth of a node $\overline{\mathsf{n}}$, denoted by $\overline{depth}(\overline{\mathsf{n}})$, is the length of the shortest path from the root of the type graph to $\overline{\mathsf{n}}$. 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 $\mathsf{k} = \overline{outdegree}(\overline{\mathsf{n}})$ where $\overline{outdegree}$ is a function that, given a node, returns the number of its sons. We also define the $\overline{indegree}$ function, which, given a node, returns the number of its predecessors. The root of the tree (i.e., the only node with no incoming forward arcs) is called $\overline{\mathsf{n}}_0$.

Each node $\overline{\mathsf{n}} \in \overline{\mathsf{N}}$ of a string graph has a label, denoted by $\overline{lb}(\overline{\mathsf{n}})$, indicating the kind of term it describes. The nodes are divided into three classes:

- *Simple nodes* have a label from the set $\{max, \bot_{\overline{\mathcal{SG}}}, \epsilon\} \cup \mathsf{K}$. This means that the leaves of string graphs trees can represent (i) all possible strings, K^* (if the node has label max), (ii) no strings, $\emptyset$ (if the node has label $\bot_{\overline{\mathcal{SG}}}$), (iii) the empty string (if the node has label ϵ), and (iv) a string made by a single character taken from the alphabet K, respectively.
- *Concat nodes* are labelled with the functor $concat/k$ (with the obvious meaning of string concatenation) and have outdegree k with $k > 0$;
- *OR nodes* have the label OR and an outdegree k.

The graphical representation of string graphs is straightforward. The nodes of a string graph are represented by their labels and every node is encircled. The direction of the arc is indicated by its arrow: forward arcs are drawn downwards, backward arcs upwards. The root of the string graph is the topmost node. An example is depicted in Figure 6. The root of the string graph is an OR-node with two sons: (i) a simple node (b) , and (ii) a $concat$-node with two sons of its own (a simple node (a), and the root (with the use of a backward arc)). This string graph represents an infinite set of strings, that is the set of strings which start with an indefinite number of a (even zero) and surely end with a b, that is, $\{b, ab, aab, aaab, \dots\} = a^*b$.

The structure of the string graph together with the labels of its nodes determines the set of represented strings. The set of finite strings represented by a node $\overline{\mathsf{n}}$ in the string graph $\overline{\mathsf{T}}$ is said to be the *denotation* of the node $\overline{\mathsf{n}}$, $\mathbb{D}(\overline{\mathsf{n}})$.

Definition 4.1
The denotation $\mathbb{D}(\overline{\mathsf{n}})$ of a node $\overline{\mathsf{n}}$ in a string graph is defined as follows:

```
**function** $\mathbb{D}(\overline{\mathsf{n}})$
```

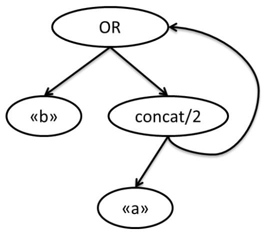


Figure 6. An example of string graph

```
    **if** $lb(\overline{n}) = max$ **then**
        **return** $K^*$
    **else if** $lb(\overline{n}) = \bot$ **then**
        **return** $\emptyset$
    **else if** $lb(\overline{n}) \in K \vee lb(\overline{n}) = \epsilon$ **then**
        **return** $\{lb(\overline{n})\}$
    **else if** $lb(\overline{n}) = concat/k$ and $\overline{n}/1, \ldots, \overline{n}/k$ are its sons **then**
        **return** $\{concat(t_1, \ldots, t_k) : t_i \text{ is finite } \wedge t_i \in \mathbb{D}(\overline{n}/i)\ \forall i \in [1, k]\}$
    **else**
        **return** $\bigcup_{i=1}^{k} \mathbb{D}(\overline{n}/i)$, as $lb(\overline{n}) = OR$ and $\overline{n}/1, \ldots, \overline{n}/k$ are its sons
    **end if**
**end function**
```

The order of the sons of a concat node is important because string concatenation is not commutative, whereas the order of the sons of an OR-node is irrelevant. $\mathbb{D}(\overline{n})$ can be $\emptyset$ or a (finite or infinite) set of finite strings. With $\overline{n}_0$ the root of string graph $\overline{T}$, we use $\mathbb{D}(\overline{T})$ as a synonym for $\mathbb{D}(\overline{n}_0)$.

Note that several distinct string graphs can have the same denotation. The existence of superfluous nodes and arcs makes some operators, such as $\leq$, quite complex and inefficient. To reduce this variety of string graphs, we impose some additional restrictions, which correspond to the definition of *compact type graphs* in [25] (where you can also find a compaction algorithm). For example, one of these restrictions is that an OR-node must have strictly more than one son and each son must not be a max-node. The denotation of the string graph is preserved when carrying out a compaction, i.e. the set of represented strings does not change.

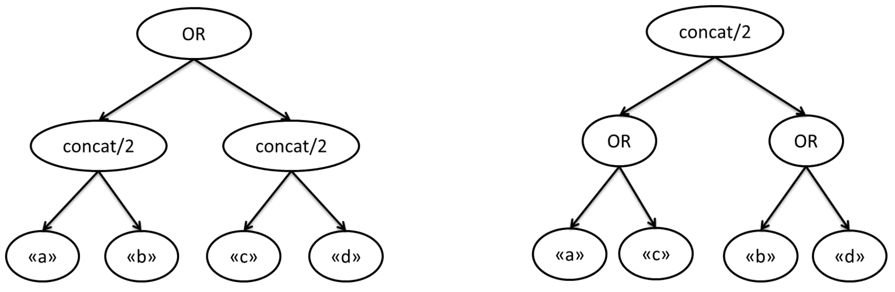
Notice also that compact string graphs are not the most economical representation. Nodes in different branches can have the same denotation. In particular, different sons of an OR node may have overlapping, even identical denotations. This makes testing whether a particular string is in the denotation of a compact string graph and the comparison of the denotations of two string graphs inefficient, so we impose a further restriction which will result in the definition of normal string graphs. Such restriction limits the expressive power of the string graphs but is necessary to achieve efficient operations. First, we introduce two functions, $prnd$ and $prlb$. The function $prnd(\overline{n})$ denotes the set of *principal nodes* of a node $\overline{n}$, and $prlb(\overline{n})$ its set of *principal labels*.

$$prnd(\overline{n}) = \begin{cases} \bigcup_{i=1}^{k} prnd(\overline{n}/i) & \text{if } lb(\overline{n}) = OR \wedge k = \overline{outdegree}(\overline{n}) \\ \overline{n} & \text{else} \end{cases}$$

$$prlb(\overline{n}) = \{lb(\overline{n}_j) : \overline{n}_j \in prnd(\overline{n})\}$$

Two sets of principal labels are *overlapping* if their intersection is not empty.

Definition 4.2 (Principal label restriction)
The principal label restriction states that each pair of sons of an OR-node must have non-overlapping sets of principal labels.



(a) Compact string graph before normalization (b) Normal string graph

Figure 7. An example of string graphs normalization

Normal string graphs are compact string graphs satisfying the principal label restriction. In [24] you can find the definition of a normalization algorithm, $normalize(\overline{\mathsf{T}})$, which takes in input a *compact* type graph and returns in output the corresponding *normal* type graph. Adapting it to string graphs is straightforward. The principal label restriction limits the expressiveness of string graphs: string graphs violating this restriction sometimes have to be replaced by a string graph denoting a larger set of strings. An example of compact string graph before and after normalization is depicted in Figure 7. The string graph in Fig. 7(a) does not satisfy the principal label restriction, since the two sons of the root node have the same label $concat/2$; its denotation is $\{ab, cd\}$. The string graph in Fig. 7(b) is normal; its denotation is $\{ab, ad, cb, cd\}$, a larger set than $\{ab, cd\}$. In fact, the normalization process makes us lose the information that, when the first character of the string is a, then the second is always b (and the same for c and d).

Normal string graphs must also satisfy, besides the principal label restriction, other four restrictions (not present in the original definition of normal *type* graphs), which we are now going to introduce.

Rule 1 Given a node $\overline{\mathsf{n}}$ with label $concat/1$ and $\overline{\mathsf{n}}/1$ as successor, replace $\overline{\mathsf{n}}$ with $\overline{\mathsf{n}'} = \overline{\mathsf{n}}/1$. Any backward arc $(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ should be replaced with the arc $(\overline{\mathsf{m}}, \overline{\mathsf{n}'})$. This rule simplifies some naïve occurrences of the functor $concat/k$. In fact, when $concat$ has only one son ($k = 1$), the result of its application is the argument itself. We thus discard every $concat/1$ node, replacing it with its argument.

Rule 2 Given a node $\overline{\mathsf{n}}$ with label $concat/k$ such that $\overline{\mathsf{n}}/i = max\ \forall i \in [1, k]$, replace $\overline{\mathsf{n}}$ with $\overline{\mathsf{n}}' = max$. This rule simplifies a node with label $concat/k$ and which successors $\overline{\mathsf{n}}/i$ all have the label max. In fact, the concatenation of all possible strings with all possible strings gives us all possible strings, again.

Rule 3 Given a node $\overline{\mathsf{n}}$ with label $concat/k$ such that $\exists i : \overline{\mathsf{n}}/i = concat/k_1 \wedge \overline{\mathsf{n}}/(i+1) = concat/k_2$, $indegree(\overline{\mathsf{n}}/i) = 1$ and $indegree(\overline{\mathsf{n}}/(i+1)) = 1$, replace $\overline{\mathsf{n}}/i$ and $\overline{\mathsf{n}}/(i+1)$ with a single new node $\overline{\mathsf{n}'} = concat/(k_1 + k_2)$ whose sons are

$$\overline{\mathsf{n}}'/j = \begin{cases} (\overline{\mathsf{n}}/i)/j & \text{if } j \le k_1 \\ (\overline{\mathsf{n}}/(i+1))/(j - k_1) & \text{otherwise} \end{cases}$$

where $j \in [1, k_1 + k_2]$. This rule merges two successive sons of a $concat$-node, which labels are both $concat$. In fact, if we concat some characters obtaining the string s_1, then we concat some other characters obtaining the string s_2, and finally we concat s_1 and s_2, we obtain the same result as concatenating *all* the characters in the first place.

Rule 4 Given a node $\overline{\mathsf{n}}$ with label $concat/k$ such that $\exists i : \overline{\mathsf{n}}/i = concat/k_1 \wedge \overline{indegree}(\overline{\mathsf{n}}/i) = 1$, replace $\overline{\mathsf{n}}/i$ with k_1 nodes such that $\overline{\mathsf{n}}/(i + j - 1) = (\overline{\mathsf{n}}/i)/j\ \forall j \in [1, k_1]$. All the sons of $\overline{\mathsf{n}}$ with index $> i$ change index, which gets augmented of $k_1 - 1$ (i.e., the generic index k becomes $k + k_1 - 1$). This rule imposes that the sons of a $concat$-node must be simple nodes (leaves), OR-nodes or $concat$-nodes with in-degree > 1. In fact, if a $concat$-node (T_1) has a *concat* son (T_2) with indegree $= 1$, we replace T_2 with all its sons, thus increasing the arity of T_1.

We can prove that such normalization rules do not affect the expressiveness of the string graphs. In fact, the denotation of a string graph does not change after the application of one of the four normalization rules.

Lemma 4.16 (Soundness of the normalization rules)
Given a normalization rule r_i ($i \in [1, 4]$) and a string graph $\overline{\mathsf{T}}$, suppose that $\overline{\mathsf{T}}'$ is the string graph resulting from the application of r_i to $\overline{\mathsf{T}}$. Then, $\mathbb{D}(\overline{\mathsf{T}}') = \mathbb{D}(\overline{\mathsf{T}})$.

Proof
We refer to [6] for the complete proof of this theorem. □

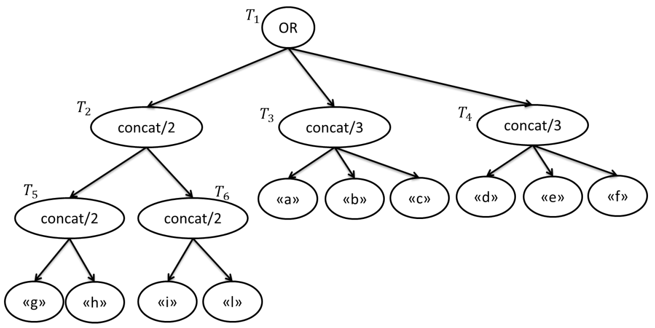
In Figure 8 we can see an example of the normalization process. First of all we apply rule r_3 to T_2 and its sons T_5 and T_6: since T_5 and T_6 are two consecutive sons of a *concat*-node and they are both *concat*-node themselves, we merge them in a single *concat*-node with, as sons, all the sons of T_5 followed by all the sons of T_6. Now we can apply rule r_1 to T_2: since it is a *concat*-node with only one son (T_7), we replace it with such son. Finally, we must apply the principal label restriction because two sons (T_3 and T_4) of the root OR-node have the same label ($concat/3$). We merge such sons in only one, moving the choice (represented by the OR) "downward" the tree (i.e., instead of choosing between the two *concat*-nodes, we choose at the level of their sons, one by one). The string graph in Figure 8(d) is the fixpoint of the application of the normalization rules; in fact we cannot apply any more rules to it. Note that the denotation has increased, being $\{ghil, abc, abf, aec, aef, dbc, dbf, dec, def\}$ instead of the original $\{ghil, abc, def\}$; this is caused by the application of the principal label restriction.

The abstract domain $\overline{\mathcal{SG}}$ is then defined as $\overline{\mathcal{SG}} = \overline{\mathsf{NSG}}$, where $\overline{\mathsf{NSG}}$ is the set of all normal string graphs, i.e., compact string graphs which satisfy the principal label restriction and the additional rules 1-4 stated above.

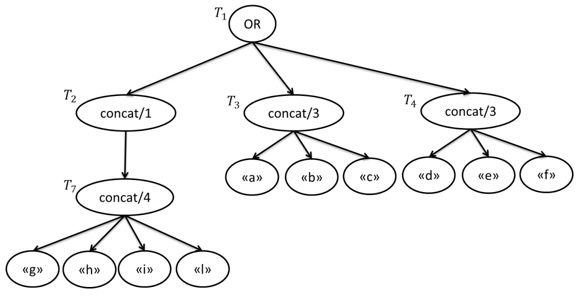
Partial order To define the partial order of the domain we can exploit the algorithm defined in [25] for computing $\leq(\overline{\mathsf{n}}, \overline{\mathsf{m}}, \emptyset)$. The algorithm compares $\mathbb{D}(\overline{\mathsf{n}})$ with $\mathbb{D}(\overline{\mathsf{m}})$ and returns true if $\mathbb{D}(\overline{\mathsf{n}}) \subseteq \mathbb{D}(\overline{\mathsf{m}})$, which is exactly what we need. In particular, 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 formal definition of the algorithm is the following:

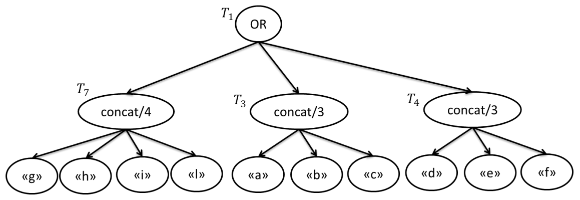
```
function $\leq(\overline{\mathsf{n}}, \overline{\mathsf{m}}, \mathsf{S}^C)$
    if $(\overline{\mathsf{n}}, \overline{\mathsf{m}}) \in \mathsf{S}^C$ then
        return $true$
    else if $lb(\overline{\mathsf{m}}) = max$ then
        return $true$
    else if $lb(\overline{\mathsf{n}}) = lb(\overline{\mathsf{m}}) = concat/k \wedge k > 0$ then
        return $\forall i \in [1, k] : \leq(\overline{\mathsf{n}}/i, \overline{\mathsf{m}}/i, \mathsf{S}^C \cup \{(\overline{\mathsf{n}}, \overline{\mathsf{m}})\})$
    else if $lb(\overline{\mathsf{n}}) = lb(\overline{\mathsf{m}}) = OR$ where $k = outdegree(\overline{\mathsf{n}})$ then
        return $\forall i \in [1, k] : \leq(\overline{\mathsf{n}}/i, \overline{\mathsf{m}}, \mathsf{S}^C \cup \{(\overline{\mathsf{n}}, \overline{\mathsf{m}})\})$
    else if $lb(\overline{\mathsf{m}}) = OR \wedge \exists m_d \in prnd(\overline{\mathsf{m}}) : lb(\overline{\mathsf{m_d}}) = lb(\overline{\mathsf{n}})$ then
        return $\leq(\overline{\mathsf{n}}, \overline{\mathsf{m_d}}, \mathsf{S}^C \cup \{(\overline{\mathsf{n}}, \overline{\mathsf{m}})\})$
    else
        return $lb(\overline{\mathsf{n}}) = lb(\overline{\mathsf{m}})$
    end if
end function
```



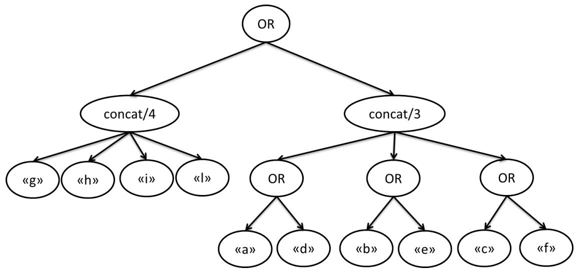
(a) String graph before the application of normalization rules



(b) Application of rule r_3 to T_2 and its sons, T_5 and T_6



(c) Application of rule r_1 to T_2 and its only son T_7



(d) Application of principal label restriction to T_3 and T_4

Figure 8. A complete example of string graphs normalization

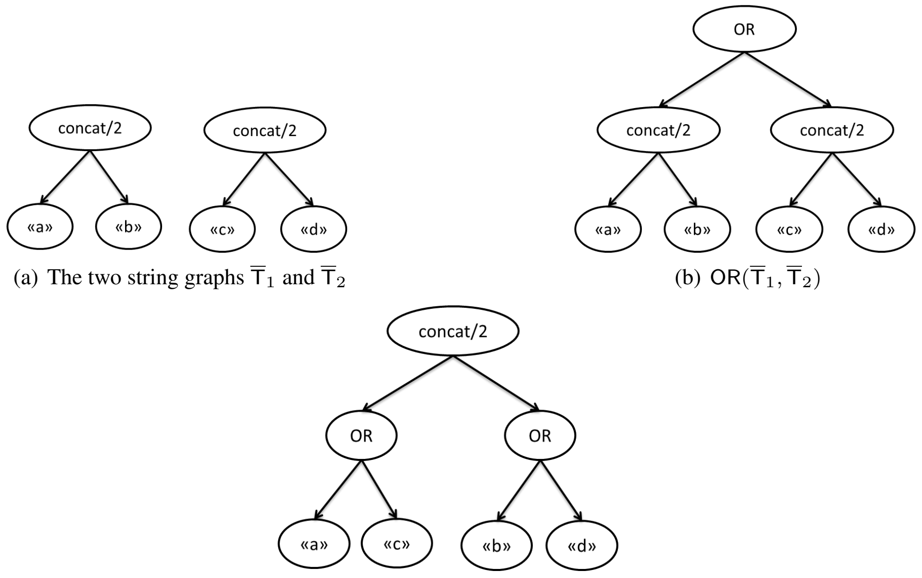
Given two string graphs $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$, to check if $\overline{\mathsf{T}}_1 \leq_{\overline{\mathcal{SG}}} \overline{\mathsf{T}}_2$ we will compute $\leq (\overline{\mathsf{n}}_0, \overline{\mathsf{m}}_0, \emptyset)$ where $\overline{\mathsf{n}}_0$ is the root of $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{m}}_0$ is the root of $\overline{\mathsf{T}}_2$. The order is then:

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

where $\overline{root}(\overline{\mathsf{T}})$ is the root element of the tree defined in $\overline{\mathsf{T}}$.

The bottom element $\bot_{\overline{\mathcal{SG}}}$ is a string graph made by one node, a $\bot$-node that represents $\emptyset$. The top element $\top_{\overline{\mathcal{SG}}}$ is a string graph made by only one node, a max-node that represents K^*.

Least upper bound and greatest lower bound The least upper bound between two string graphs $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$ can be computed by creating a new string graph $\overline{\mathsf{T}}$ whose root is an OR-node whose sons



(a) The two string graphs $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$

(b) $\mathsf{OR}(\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2)$

(c) Result of the least upper bound: $\overline{normStringGraph}(\mathsf{OR}(\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2))$

Figure 9. Computation of the lub

are $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$. Then we apply the compaction plus normalization algorithm that will transform $\overline{\mathsf{T}}$ in a normal string graph:

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

An example is depicted in Figure 9.

The greatest lower bound operator behaves like the glb between type graphs, which is described in the appendix of [25]. The authors present an algorithm, $intersection(\overline{\mathsf{n}}_1, \overline{\mathsf{n}}_2)$, which computes the type graph $\overline{\mathsf{T}}'$, whose denotation is the intersection of the denotations of the type graphs with roots $\overline{\mathsf{n}}_1$ and $\overline{\mathsf{n}}_2$. Their strategy to deal with this kind of problem is to leave the old type graphs unchanged and to construct the new type graph step by step. The initialization creates the root $\bar{\mathsf{l}}_0$ of $\overline{\mathsf{T}}'$ whose required denotation is defined in terms of the nodes $\overline{\mathsf{n}}_1$ and $\overline{\mathsf{n}}_2$. At this point the root $\bar{\mathsf{l}}_0$ is called an *unexpanded leaf*. They define the function *is* which associates at every step in the construction of $\overline{\mathsf{T}}'$ with each node in $\overline{\mathsf{T}}'$ a set of nodes from the given type graphs such that the second function on the nodes of $\overline{\mathsf{T}}'$, $\mathbb{D}$-is, specifies for each node $\bar{\mathsf{l}}$ of $\overline{\mathsf{T}}'$ its intended denotation.

$$\mathbb{D} - is(\bar{\mathsf{l}}) = \bigcap_{\overline{\mathsf{n}} \in is(\bar{\mathsf{l}})} \mathbb{D}(\overline{\mathsf{n}})$$

Each step extends $\overline{\mathsf{T}}'$ without decreasing the denotation of its nodes. This is done by transforming one of the unexpanded leaves $\bar{\mathsf{l}}$ of $\overline{\mathsf{T}}'$ into a usual node (after the transformation, $\bar{\mathsf{l}}$ is called a *safe node*), and new unexpanded leaves may be added as sons of $\bar{\mathsf{l}}$. The nodes of $\overline{\mathsf{T}}'$, in each step of its construction, belong either to S^{ul}, the set of unexpanded leaves, or to S^{sn}, the set of safe nodes.

Lemma 4.17
The abstract domain $\overline{\mathcal{SG}}$ is a complete lattice, $\sqcup_{\overline{\mathcal{SG}}}$ is the least upper bound operator and $\sqcap_{\overline{\mathcal{SG}}}$ is the greatest lower bound operator.

Proof
Since string graphs are just a particular case of type graphs, we refer to [24, 25] for the complete proof of this lemma. □

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

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

Widening operator For the widening operator, we can exploit the one defined in [40]. The widening operator is always applied to an old graph g_{old} and a new graph g_{new} to produce a new graph g_{res}. The main idea behind the widening operator of [40] for type graphs is to consider two graphs:

$$g_0 = g_{old} \text{ and } g_n = (g_{old} \sqcup g_{new})$$

and exploit the topology of the graphs to guess where g_n is growing compared to g_0. The key notion is the concept of topological clash which occurs in situations where: (i) an OR-node v_0 in g_0 corresponds to an OR-node v_n in g_n where $prlb(v_0) \neq prlb(v_n)$, or (ii) an OR-node v_0 in g_0 corresponds to an OR-node v_n in g_n where $depth(v_0) < depth(v_n)$. In these cases the widening operator tries to prevent the graph from growing by introducing a cycle in g_n. Given a clash (v_0, v_n), the widening searches for an ancestor v_a to v_n such that $prlb(v_n) \subseteq prlb(v_a)$. If such an ancestor is found and if $v_a \geq v_n$, a cycle can be introduced.

When no ancestor with a suitable $prlb$-set can be found, the widening operator simply allows the graph to grow. Termination will be guaranteed because this growth necessarily adds along the branch of a $prlb$-set which is not a subset of any existing $prlb$-set in the branch. This case happens frequently in early iterations of the fixpoint. Letting the graph grow in this case is of great importance to recover the structure of the type in its entirety.

The last case to consider appears when there is an ancestor v_a with a suitable $prlb$-set, but $v_a \geq v_n$ is false. In this case, introducing a cycle would produce a graph g_{res} whose denotation may not include the denotation of g_n, and hence the widening cannot perform cycle introduction. Instead, the operation replaces v_a by a new OR-node which is an upper bound to v_a and v_n but decreases the overall size of the graph. The widening is then applied again on the resulting graph.

In conclusion, such widening operator can be viewed as a sequence of transformations on g_n which are of two types: cycle introduction and node replacement, until no more topological clashes can be resolved.

Concretization and abstraction functions The concretization function is simply defined by $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}) = \mathbb{D}(\overline{\mathsf{T}})$. Let the abstraction function $\alpha_{\overline{\mathcal{SG}}}$ be defined by $\alpha_{\overline{\mathcal{SG}}} = \lambda \mathsf{Y}. \sqcap_{\overline{\mathcal{SG}}} \{\overline{\mathsf{B}} : \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{B}}) \subseteq \mathsf{Y}\}$. These two functions form a Galois connection, i.e. $\langle \wp(\mathsf{S}), \subseteq \rangle \underset{\alpha_{\overline{\mathcal{SG}}}}{\overset{\gamma_{\overline{\mathcal{SG}}}}{\leftrightarrows}} \langle \overline{\mathcal{SG}}, \leq_{\overline{\mathcal{SG}}} \rangle$. See [24, 25] for the proof of this assertion.

Semantics Table VII defines the abstract semantics on $\overline{\mathcal{SG}}$. Let us discuss in detail the semantics of each operator.

The evaluation of a string (made by k characters) returns a *concat*-node with all the characters that compose the string as sons.

When we concatenate two strings, we create a new string graph, whose root is a *concat*-node with two sons. The two sons are the roots of the two input abstract values. Then we need to normalize the result, to be sure that it is a normal string graph.

The semantics of $\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}$ returns a precise value only if the root is a *concat*-node whose first e sons are characters. In fact, if the root of the string graph is a *concat*-node and its first `endIndex` sons are simple nodes (leaves), then we can return the exact substring. Otherwise, we return $\top_{\overline{\mathcal{SG}}}$.

The semantics of $\texttt{contains}_{\mathsf{c}}$ returns false iff we are sure that the character c does not appear in the string, that is, there is no simple node labelled with such character and there is no *max*-node. We can return true iff we find in the string graph a *concat*-node $\overline{\mathsf{m}}$ containing a son with label c, and the path from the root to $\overline{\mathsf{m}}$ does not contain any *OR*-node. Otherwise, we will have to return $\top_{\overline{\mathcal{SG}}}$.

Theorem 4.18 (Soundness of the abstract semantics)
$\overline{\mathbb{S}_{\mathcal{SG}}}$ and $\overline{\mathbb{B}_{\mathcal{SG}}}$ are a sound overapproximation of $\mathbb{S}$ and $\mathbb{B}$, respectively. Formally, $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{S}_{\mathcal{SG}}}[\![\mathsf{s}]\!](\overline{\mathsf{T}})) \supseteq \{\mathbb{S}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})\}$ and $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{B}_{\mathcal{SG}}}[\![\mathsf{s}]\!](\overline{\mathsf{T}})) \geq_{\mathsf{B}} \{\mathbb{B}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})\}$.

Proof
We prove the soundness separately for each operator.

- $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{new String(str)}]\!]()) \supseteq \{\mathbb{S}[\![\texttt{new String(str)}]\!]()\}$ follows immediately from the definition of $\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{new String(str)}]\!]()$ and of $\gamma_{\overline{\mathcal{SG}}}$.
- Consider the binary operator `concat`. Let $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$ be two string graphs. $\{\mathbb{S}[\![\texttt{concat}]\!](\mathsf{c}_1, \mathsf{c}_2) : \mathsf{c}_1 \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_1) \wedge \mathsf{c}_2 \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_2)\}$ contains strings which are the concatenation of one string from $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_1)$ and one from $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_2)$ by definition of $\mathbb{S}$. Let s be one of these strings. s belongs to $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{concat}]\!](\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2))$, since $\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{concat}]\!](\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2)$ produces a new string graph which has a concat-node as root and the two original string graphs as sons [§], and the concretization of such string graph is $\{concat(t_1, t_2) : t_i \text{ is finite } \wedge t_i \in \mathbb{D}(\overline{\mathsf{root}}/i)\ \forall i \in [1, 2]\}$ by definition of $\gamma_{\mathcal{SG}}$.
- Consider the unary operator $\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}$ and let $\overline{\mathsf{T}}$ be a string graph. Consider the two following cases:
  - if $\overline{root}(\overline{\mathsf{T}}) = \mathsf{concat/k} \wedge \forall \mathsf{i} \in [0, \mathsf{e} - 1] : \overline{lb}(\overline{root}(\overline{\mathsf{T}})/\mathsf{i}) \in \mathsf{K}$, then the root of $\overline{\mathsf{T}}$ is a concat-node and its first e sons are all simple characters. In this case, all strings belonging to $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})$ will start with the concatenation of these characters, by definition of $\mathbb{D}(\overline{\mathsf{T}})$. This prefix is also certainly longer than e characters. Then, a string belonging to $\{\mathbb{S}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})\}$ is composed by the concatenation of all the characters of the nodes from $\overline{root}(\overline{\mathsf{T}})/\mathsf{b}$ to $\overline{root}(\overline{\mathsf{T}})/(\mathsf{e} - 1)$ by definition of $\mathbb{S}$. This corresponds exactly to $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\overline{\mathsf{T}}))$, since the abstract semantics applied to $\overline{\mathsf{T}}$ produces a string graph whose root is a concat-node and which sons are the nodes from $\overline{root}(\overline{\mathsf{T}})/\mathsf{b}$ to $\overline{root}(\overline{\mathsf{T}})/(\mathsf{e} - 1)$.
  - otherwise, the abstract semantics returns $\top_{\overline{\mathcal{SG}}}$, that approximates any possible value of the concrete semantics.
- Consider the unary operator $\texttt{contains}_{\mathsf{c}}$ and let $\overline{\mathsf{T}}$ be a string graph. Regarding the character c, we have three cases:
  - if $\exists \overline{\mathsf{m}} \in \overline{\mathsf{T}} : \overline{\mathsf{m}} = \mathsf{concat/k} \wedge OR \notin \overline{path}(\overline{\mathsf{root}}, \overline{\mathsf{m}}) \wedge \exists \mathsf{i} : \overline{lb}(\overline{\mathsf{m}}/\mathsf{i}) = \mathsf{c}$, this means that there exists a concat-node in $\overline{\mathsf{T}}$ that (i) has a son with the character c as label, and (ii) the path from the root to such node does not contain OR nodes. Then, by definition of $\mathbb{D}(\overline{\mathsf{T}})$, the character c belongs to all strings in $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})$, and then the result of the concrete semantics is always `true`. Since $\overline{\mathbb{B}_{\mathcal{SG}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\overline{\mathsf{T}}) = \texttt{true}$ by the definition of the abstract semantics, the abstract semantics soundly approximates the concrete semantics.
  - if $\nexists \overline{\mathsf{n}} \in \overline{\mathsf{T}} : \overline{lb}(\overline{\mathsf{n}}) = \mathsf{max} \vee \overline{lb}(\overline{\mathsf{n}}) = \mathsf{c}$, this means that no node of the string graph has label c or max. Then, the character c cannot be contained in any of the concrete strings corresponding to the abstract state $\overline{\mathsf{T}}$ and for this reason the concrete semantics always

[§]The string graph is also normalized, but the normalization can only increase the concretization of an abstract state, thus we can ignore it: if a string belongs to the concretization of a not-normal string graph, it will surely belong also to its normalized version.

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

(a) Analysis of `prog1`

#I	Var	$\overline{\mathcal{SG}}$
1	x	$\mathsf{concat}[\text{“}a\text{”}]$
3	x	$\mathsf{OR}[\text{“}a\text{”}; \mathsf{concat}[\text{“0”}; \text{“}a\text{”}; \text{“1”}]]$
4	x	$\mathsf{OR}_1[\text{“}a\text{”}; \mathsf{concat}[\text{“0”}; \mathsf{OR}_1; \text{“1”}]]$

(b) Analysis of `prog2`

Figure 10. The results of $\overline{\mathcal{SG}}$

returns `false`. Since $\overline{\mathbb{B}}_{\mathcal{SG}}[\![\texttt{contains}_\mathsf{c}]\!](\overline{\top}) = \texttt{false}$ by the definition of the abstract semantics, the abstract semantics soundly approximates the concrete semantics.

– otherwise, the abstract semantics on $\overline{\top}$ returns $\top_\mathtt{B}$, and this soundly approximates any possible result of the concrete semantics.

□

Running Example The results of the analysis of the two running examples through string graphs are depicted in Figures 10(a) and 10(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 the string s. The symbol $+$ represents, as usual, string concatenation, while ; is used to separate different sons of a node.

For `prog1`, at line 1 we represent `query` with a string graph made by a $concat$-node with all the characters of s_1 as sons. The `l` variable (line 3) corresponds simply to a max-node, since we do not know its value. At line 4 we concatenate the current value of `query` with s_2, `l` and s_3: the abstract value of `query` then is a $concat$ node with, as sons, all the characters of s_1, followed by all the characters of s_2, followed by a max-node, followed by all the characters of s_3. Since the value of `l` is unknown, we must compute the least upper bound between the values of `query` after line 1 and 4. We obtain a string graph made by an OR-node with the two input string graphs as sons. Then, at line 6 we associate the variable `per` to the abstraction of s_4. Finally, at line 8 we concatenate `query` to s_5, `per` and s_6: we obtain a string graph which is made by a $concat$ node as root, and, as sons, the string graph associated to `query` at line 5 and then all the characters of s_5, s_4 and s_6, one after the other. The resulting string graph for `query` represents exactly the two possible outcomes of the procedure.

For `prog2`, after line 1 we represent `x` with a $concat$ node with just one son, containing an `a` character. After the first iteration of the loop, line 3, the abstract value associated to `x` is a $concat$ node with three sons, `0`, `a` and `1`. The least upper bound between the two abstract values (before entering the loop and after the first iteration) is an OR-node with two sons: one is an `a` character, the other is the value of `x` after the first iteration. Since we have not reached convergence, we must compute the value of after the second iteration also. In this domain, though, computing the least upper bound of the values of the first n iterations is not sufficient to reach convergence, since we always add some new branch to the string graph. We need to use the widening operator and the result (after reaching convergence) is as follows: $\mathsf{OR}_1[\text{“}a\text{”}; \mathsf{concat}[\text{“0”}; \mathsf{OR}_1; \text{“1”}]]$. The string graph root is an OR-node with two sons: an `a` character and a $concat$ node with three sons. The first and last sons are, respectively, a `0` character and a `1` character. The second son is, instead, the root OR-node, thanks to the use of a backward arc. 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 $0^n \dots 1^n$.

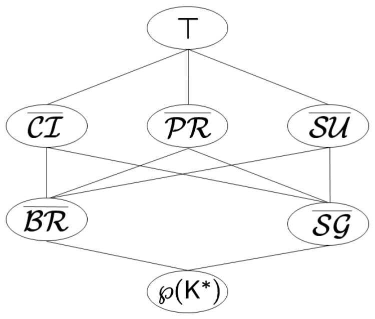

⊤
$\overline{\mathcal{CI}}$
$\overline{\mathcal{PR}}$
$\overline{\mathcal{SU}}$
$\overline{\mathcal{BR}}$
$\overline{\mathcal{SG}}$
$\wp(\mathsf{K}^*)$

Figure 11. The hierarchy of abstract domains

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 slightly more accurate.

4.5. Discussion: Relations Between the Five Domains

The abstract domains we introduced track different types of information. Let us discuss the relations between the five domains.

Intuitively, there are two axes on which the analysis of string values can work: the characters contained in a string, and their position inside the string. It is easy to see that $\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$ 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 [6] 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.

Combining these results, we obtain the lattice depicted in Figure 11, 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 values of strings 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.

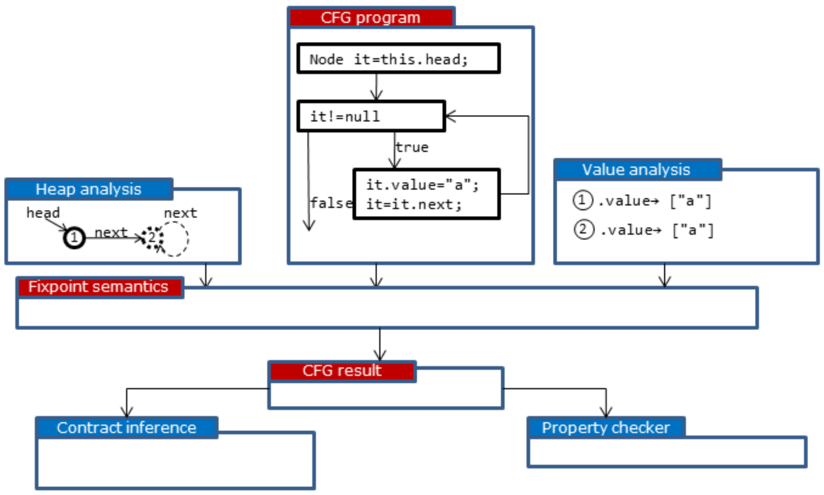


Figure 12. The structure of Sample

Table VIII. Results of `caseStudy1`

Abstract domain	**Value of `q`**
$\overline{\mathcal{CI}}$	$(\{E, e, s, *, T, F, a, M, , L, C, r, R, O, S, d\}, \{E, e, s, *, n, T, = , t, u, F, a, M, I, , L, C, H, W, r, R, O, S, d\})$
$\overline{\mathcal{PR}}$	"SELECT * FROM ADDRESS"
$\overline{\mathcal{SU}}$	$\top_{\overline{\mathcal{SU}}}$
$\overline{\mathcal{BR}}$	[{"SELECT * FROM address"}]$^{(1,1)}$[{"WHERE studentId="}]$^{(0,1)}$
$\overline{\mathcal{SG}}$	OR [concat["SELECT * FROM addressWHERE studentId="] , concat["SELECT * FROM address"]]

5. EXPERIMENTAL RESULTS

We developed a preliminary implementation of all the abstract domains formalized in Section 4 in Sample [¶] (Static Analyzer of Multiple Programming LanguagEs) [15, 16, 17, 44]. Sample is a generic analyzer of object-oriented programs that is parametric on a value (e.g., numerical) domain, a heap abstraction, and on the property of interest or an engine to infer annotation (e.g., pre- and post- conditions). Figure 12 depicts the structure of Sample. The string analyses are plugged as value analyses.

Notice that the results we have reported on the two motivating examples introduced in Section 1.1 are obtained through this implementation.

We discuss now the application of our analysis to two other case studies. Figure 13 reports their code. In particular, `caseStudy1` is an interesting example cited in [4] as motivation for their work. The code creates a SQL query by first assigning a constant value ("SELECT * FROM address") to the string `q` and then concatenating it with another constant string ("WHERE studentId="), but only if some condition (unknown at compile time) holds. In Table VIII we report the results of the analysis of this case study with all our five domains. With $\overline{\mathcal{CI}}$ we get that (i) all the characters of the string "SELECT * FROM address" will *certainly* be contained in `q` at the end of the program, and (ii) all the characters of the string "WHERE studentId=" (in addition to those of "SELECT * FROM address") *could* be contained in `q` at the end of the program. $\overline{\mathcal{PR}}$ tells us that `q` will certainly start with the string "SELECT * FROM address", while $\overline{\mathcal{SU}}$ is not able to give us any information, since its result is $\top_{\overline{\mathcal{SU}}}$. $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$, instead, infer the same information (even though encoded in

[¶]`http://www.pm.inf.ethz.ch/research/semper/Sample`

```
var q : String = "SELECT * FROM address";
if (i != 0)
        q = q + "WHERE studentId="
```

(a) The first case study, `caseStudy1`

```
var sql1 : String = "";
var sql2 : String = "";
sql1 = "SELECT";
sql1 = sql1 + " " + l;
sql1 = sql1 + " " + "FROM";
sql1 = sql1 + " " + l;
sql2 = "UPDATE";
sql2 = sql2 + " " + l;
sql2 = sql2 + " " + "SET";
sql2 = sql2 + " " + l + " = " + l;
```

(b) The second case study, `caseStudy2`

Figure 13. Two additional case studies

Table IX. Results of `caseStudy2`

Abstract domain	**Value of `sql1`**	**Value of `sql2`**
$\overline{\mathcal{CI}}$	$(\{E, T, F, M, L, ' \ ', C, R, O, S\}, \mathsf{K})$	$(\{E, T, =, ' \ ', U, A, P, D, S\}, \mathsf{K})$
$\overline{\mathcal{PR}}$	"SELECT"	"UPDATE"
$\overline{\mathcal{SU}}$	$\top_{\overline{\mathcal{SU}}}$	$\top_{\overline{\mathcal{SU}}}$
$\overline{\mathcal{BR}}$	$[\{\text{“SELECT”}\}]^{(1,1)}[\{\text{“ ”}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}$ $[\{\text{“ ”}\}]^{(1,1)}[\{\text{“FROM”}\}]^{(1,1)}[\{\text{“ ”}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}$	$[\{\text{“UPDATE”}\}]^{(1,1)}[\{\text{“ ”}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}[\{\text{“ ”}\}]^{(1,1)}[\{\text{“SET”}\}]^{(1,1)}$ $[\{\text{“ ”}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}[\{\text{“ = ”}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}$
$\overline{\mathcal{SG}}$	concat[“SELECT ” ; max ; “ FROM ” ; max]	concat[“UPDATE ” ; max ; “ SET ” ; max ; “ = ” ; max]

different ways), which also corresponds *exactly* to the outcome of the program: `q` could have value "SELECT * FROM addressWHERE studentId=" or "SELECT * FROM address". From this result we can discover the bug hidden in the program: there is a space missing between "address" and "WHERE", which will make the SQL query to sometimes fail at runtime (when $\texttt{i} \neq \texttt{0}$). Note that the resulting bricks list is made by just two bricks, while the resulting string graph is composed by 61 nodes (1 OR, 2 concat, 37+21 simple nodes).

`caseStudy2` is inspired from an example in [2]. This program creates two strings `sql1` and `sql2` (which will be executed as SQL queries) by successive concatenations: each statement concatenates the previous value of the string variable with some other string (sometimes coming from another variable). The variable `l` is used in these concatenations, but we do not know the value of such variable at compile time since it is an input. In Table IX we report the results of the analysis on this case study with all our five domains. $\overline{\mathcal{CI}}$ discovers that (i) the string `sql1` surely contains all the characters of "SELECT", " " and "FROM", but it could contain any character of the alphabet K (because of the concatenation with `l`), and (ii) the string `sql2` surely contains all the characters of "UPDATE", " ", "SET" and " = ", but it could contain any character of the alphabet K (because of the concatenation with `l`). $\overline{\mathcal{PR}}$ tells us that `sql1` starts with "SELECT" and `sql2` starts with "UPDATE", while, as in the previous example, $\overline{\mathcal{SU}}$ is not able to track any information about the resulting values of the two strings. As it happened in `caseStudy1`, $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$ infer both the same information, which also corresponds *exactly* to the outcome of the program: `sql1` starts with "SELECT ", then there is an unknown part (due to `l`), then it continues with " FROM " and it finally ends with another unknown part. This information is encoded through 7 bricks (in the resulting bricks list) and 16 nodes (in the resulting string graph). Note that, if we normalized the result of

$\overline{\mathcal{BR}}$, we would reduce the number of bricks in the list to 4. The other string variable, `sql2`, starts with "UPDATE ", then there is an unknown part, then it continues with " SET " followed by another unknown part, then " = " and finally the last unknown part. This information is encoded through 9 bricks (which could be reduced to 6 with a normalization step) and 19 nodes. Note that, on simple programs, the $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$ domains tend to produce the same results.

In order to check the scalability and performances of our analysis, we will exploit some Scala standard libraries. The preliminary experimental results point out that $\overline{\mathcal{CI}}$ and $\overline{\mathcal{PR}} \times \overline{\mathcal{SU}}$ are quite efficient, $\overline{\mathcal{BR}}$ is slightly slower but still fast, while $\overline{\mathcal{SG}}$'s is the slowest domain of the framework. In fact, the analyses using $\overline{\mathcal{CI}}$ and $\overline{\mathcal{PR}} \times \overline{\mathcal{SU}}$ lasted just a fraction of second, using $\overline{\mathcal{BR}}$ a little more (always remaining below the second, though), while with $\overline{\mathcal{SG}}$ the analysis lasts some seconds, especially when the code is not trivial (e.g., with string concatenations inside loops).

6. RELATED WORK

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 useful string properties. In fact, 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, often in combination with user inputs, strings are sent to their respective processors. However, strings are usually not evaluated for their validity or security despite the potential usefulness of such metrics. For these reasons, string analysis has been widely studied.

Hosoya and Pierce [23] designed a statically typed processing language (called XDuce and based on the theory of finite tree automata) for building XML documents. Its sound type system ensures that dynamically generated documents conform to "templates" defined by the document types. This work differs a lot from ours, since, first of all, they use type systems instead of abstract interpretation. Moreover, they are focused on building XML documents, while our focus is on collecting possible values of generic string variables. Lastly, 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.* [41]. They presented an automata-based approach for the verification of string operations in PHP programs based on symbolic string analysis. They encode the set of string values that string variables can take as deterministic finite automaton (DFA): the language accepted by the DFA corresponds to the values that the corresponding string variable can assume at that program point. Using this technique, it is possible to automatically verify the sanitization of a string, showing that attacks are not possible. The information tracked by this analysis is fixed, and it is specific for PHP programs. However, in 2011 they proposed a unifying framework [43] of their previous works, i.e. an abstraction lattice which can be tuned to provide various trade-offs between precision and performance. The framework is based on the *regular abstraction* [42], a relational analysis in which values of string variables are represented as multi-track DFA (each track corresponds to a specific string variable). As the number of variables increases, such relational analysis becomes intractable, so they add two other string abstraction (*relation abstraction* and *alphabet abstraction*) to improve the scalabity of their approach. They also propose a heuristic to choose a particular point in their abstraction lattice, depending on the program and property to be verified. The *alphabet abstraction* can be seen as a more complex version of $\overline{\mathcal{CI}}$, since it also keeps track of the position of characters; such abstraction must be applied to automata, thus obtaining more convoluted operations than in our domain $\overline{\mathcal{CI}}$.

Tabuchi *et al.* [38] presented a type system based on regular expressions. It is focused on a minimal λ-calculus supporting concatenation and pattern matching. This calculus established a theoretical foundation of using regular expressions as types of strings in text processing languages. Also in this case (as in XDuce), the approach is very different from ours, since it employs type system. The only resemblance regards the use of regular expressions, which we use in the $\overline{\mathcal{BR}}$ domain.

Thiemann [39] introduced another type system for string analysis (based on context-free grammars) and presented a type inference algorithm based on Earley's parsing algorithm. It was not discussed how to deal with string operations other than concatenation (while in this article we show the semantics of various other string operations). His analysis is more precise than those based on regular expressions, but his type inference algorithm is incomplete (though sound). Also, the analysis is tuned at a fixed level of precision.

Context-free grammars are also the basis of the analysis of Christensen, Møller and Schwartzbach [4]. This analysis (implemented in a tool called JSA, Java String Analyzer) is tuned at a fixed level of abstraction and it statically determines the values of string expressions in Java programs. This work has considerable similarities with the $\overline{\mathcal{SG}}$ domain because type graphs are closely related to context free grammars. However, they generally obtain a regular grammar which *contains* the reference grammar, but they are not the same grammar. In the second running example of this article, `prog2`, $\overline{\mathcal{SG}}$ domain reaches a better precision than theirs, being able to model precisely the reference grammar ($S \to$ "a" | "0" S "1") without the need of any kind of approximation. Moreover, they precisely abstract only the concatenation operation, while for other string operators they use less precise automata operations or character set approximations; our work deals precisely also with other operators and can be easily extended to as many as needed. Møller published many other papers concerning abstractions for string analysis, but every one of them is strictly focused on some particular case ([34] on a set of HTML pages, [3, 29, 37] on XML documents, [35] on XSLT, [1] on XHTML, [26, 30, 28, 36] on type checking), without producing a unifying framework, while we aim at a higher generality.

To statically check the properties of Web pages generated dynamically by a server-side program, Minamide [32] developed a static program analysis that approximates the string output of a program with a context-free grammar. His analysis is based on the Java string analyzer (JSA) of [4], but the novelty of his analysis is the application of finite-state-automata transducers to revise the flow equations due to string-update operations embedded in the program, reaching a simpler and more precise analyzer than [4]. This work is specific for HTML pages. Even though the obtained results are similar to ours, his work suffers from some other limitations: after extracting from the program the corresponding grammar with operation productions, such grammar must be transformed into a context-free one. This restricts the string operations supported by the framework (to those which transform a context-free grammar into another context-free grammar) and it imposes that no string operation must occur in a cycle of production. Finally, the validity check between the reference grammar and the context-free grammar is very costly and can be done only when the nesting depth of the elements in the generated document is bounded.

A combination of grammars and abstract interpretation was studied by Cousot and Cousot in [11], where they showed that set constraint solving of a particular program `P` could be understood as an abstract interpretation over a finite domain of tree grammars, constructed from `P`. However, their work is at a very high level and their concern is not the approximation of string variables, so no string operators are considered in such article.

Abstract interpretation specifically focused on string analysis can be found in Choi *et al.* [2], where they used standard abstract-interpretation techniques with heuristic widening to devise a string analyzer that handles heap variables and context sensitivity. They selected a restricted subset of regular expressions as abstract domain (which results in limited loss of expressibility). Our $\overline{\mathcal{BR}}$ domain is similar to this work, and, even though most of the lattice operators are different, we obtain the same result on the second running example (0^*a1^*). $\overline{\mathcal{SG}}$ domain, instead, is more precise than their domain. In fact, on the second running example (`prog2`) the string graphs are able to produce exactly the reference grammar ($0^n a 1^n$), while their result does not constrain the number of 0s and 1s to be the same.

Kim and Choe [27] introduced recently another approach to string analysis based on abstract interpretation. They abstract strings with pushdown automata (PDA). The result of their 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.

Doh *et al.* [14] reported the "abstract parsing" technique, which statically analyzes string values from programs. They combine LR(k)-parsing technology and data-flow analysis to analyze, in advance of execution, the documents dynamically generated by a program. Based on the document languages context-free reference grammar and the programs control structure, the analysis predicts how the documents will be generated and parses the predicted documents. Their technique is quite precise, but the level of abstraction is fixed.

Given this context, our work is the first one (together with [43], published at the same time as [7]) 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.

7. CONCLUSION AND FUTURE WORK

String analysis is a static analysis technique that determines the string values that a variable can hold at specific points in a program. This information is often useful to help program understanding, to detect and fix programming errors and security vulnerabilities, and to solve certain program verification problems.

In this article we approached string analysis using the abstract interpretation framework. In particular, we focused on the construction of various abstract domains. We chose some string operators and we defined their concrete and abstract semantics. We created five domains: $\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$, $\overline{\mathcal{SU}}$, $\overline{\mathcal{BR}}$, $\overline{\mathcal{SG}}$. The first three domains ($\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$, $\overline{\mathcal{SU}}$) are quite simple and the information we can trace with them is limited. However, they are not computationally expensive (the prefix and suffix in particular) and they do not need to define a widening operator. The last two domains ($\overline{\mathcal{BR}}$, $\overline{\mathcal{SG}}$) are certainly more complex, and they let us trace more interesting patterns. The lattices of such domains are infinite and do not satisfy ACC; thus, we define a widening operator. Even though these two domains are both quite precise, $\overline{\mathcal{SG}}$ seems to be the *most* precise domain of our framework (and, for the usual trade-off between performance and precision, the most costly).

As a string can be seen also as an array of characters, as a future work we plan to generalize our analysis in order to manage arrays of any base type (not only characters), combining it with domains which abstract relevant properties of such base types.

REFERENCES

1. Brabrand C., Møller A., Schwartzbach M.I. 2001. *Static Validation of Dynamically Generated HTML.* In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01. Pages 221 - 231.
2. Choi T., Lee O., Kim H., and Doh K. 2006. *A practical string analyzer by the widening approach.* In Proceedings of APLAS '06. Pages 374-388. Springer.
3. Christensen A.S., Møller A, Schwartzbach M.I. 2002. *Static Analysis for Dynamic XML.* Technical Report RS-02-24. Presented at Programming Language Technologies for XML (PLAN-X) 2002.
4. Christensen A., Moller A., and Schwartzbach M. 2003. *Precise analysis of string expressions.* In Proceedings of SAS '03. Pages 1-18. Springer-Verlag.
5. Cortesi A. and Zanioli M. 2011. *Widening and narrowing operators for abstract interpretation.* In Computer Languages, Systems and Structures. Volume 37(1). Pages 2442. Elsevier.
6. Costantini G. 2010. *Abstract domains for static analysis of strings.* Master's thesis, Ca' Foscari University of Venice.
7. Costantini G., Ferrara P., and Cortesi A. 2011. *Static analysis of string values.* In Proceedings of 13th International Conference on Formal Engineering Methods, ICFEM '11. Volume 6991 of LNCS. Pages 505-521. Springer.
8. Cousot P. and Cousot R. 1977. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '77. Pages 238-252. ACM.
9. Cousot P. and Cousot R. 1979. *Systematic design of program analysis frameworks.* In Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '79. Pages 269-282. ACM.
10. Cousot P., Cousot R. 1992. *Abstract interpretation and application to logic programs.* Journal of Logic Programming, 13(23):103179, 1992.
11. Cousot P., Cousot R. 1995. *Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation.* In FPCA 1995. Pages 170 - 181.
12. Cousot P., Cousot R., Feret J., Mauborgne L., Mine A., Monniaux D., and Rival X. 2005. *The ASTREE analyzer.* In Proceedings of the 14th European conference on Programming Languages and Systems, ESOP '05. Pages 21-30. Springer-Verlag.

13. Cousot P. and Halbwachs N. 1978. *Automatic discovery of linear restraints among variables of a program*. In Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '78. Pages 84-96. ACM.
14. Doh K., Kim H., and Schmidt D. 2009. *Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology*. In Proceedings of the 16th International Symposium on Static Analysis, SAS '09. Pages 256 - 272. Springer-Verlag.
15. Ferrara P. 2010. *Static type analysis of pattern matching by abstract interpretation*. In Proceedings of the 12th IFIP WG 6.1 international conference and 30th IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems, FORTE/FMOODS '10. Pages 186-200. Springer-Verlag.
16. Ferrara P., Fuchs R., and Juhasz U. 2012. *TVAL+ : TVLA and Value Analyses Together*. In Proceedings of the 10th International Conference on Software Engineering and Formal Methods, SEFM '12. Springer-Verlag.
17. Ferrara P. and Müller P. 2012. *Automatic inference of access permissions*. In Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation, VMCAI '12. Pages 202-218. Springer-Verlag.
18. Gould C., Su Z., and Devanbu P. 2004. *Static checking of dynamically generated queries in database applications*. In Proceedings of the 26th International Conference on Software Engineering, ICSE '04. Pages 645-654. IEEE Computer Society.
19. Granger P. 1991. *Static analysis of linear congruence equalities among variables of a program*. In Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1, TAPSOFT '91. Pages 169 - 192. Springer-Verlag.
20. Gulwani S. 2011. *Automating string processing in spreadsheets using input-output examples*. In Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11. Pages 317-330. ACM.
21. Halder R. and Cortesi A. 2010. *Obfuscation-based analysis of sql injection attacks*. In Proceedings of the The IEEE symposium on Computers and Communications, ISCC '10. Pages 931-938. IEEE Computer Society.
22. Hooimeijer P. and Veanes M. 2011. *An evaluation of automata algorithms for string analysis*. In Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation, VMCAI '11. Pages 248-262. Springer-Verlag.
23. Hosoya H. and Pierce B. 2003. *Xduce: A statically typed xml processing language*. ACM Transactions on Internet Technology (TOIT). Volume 3, Issue 2. Pages 117 - 148. ACM.
24. Janssens G. and Bruynooghe M. 1990. *Deriving descriptions of possible values of program variables by means of abstract interpretation: Definitions and proofs*. Technical Report CW-107, Computer Science Dept., K.U. Leuven.
25. Janssens G. and Bruynooghe M. 1992. *Deriving description of possible values of program variables by means of abstract interpretation*. Journal of Logic Programming. Volume 13, Issue 2-3. Pages 205 - 258. Elsevier.
26. Jensen S.H., Møller A., Thiemann P. 2009. *Type Analysis for JavaScript*. In Proceedings of the 16th International Static Analysis Symposium, SAS '09. Volume 5673. Springer-Verlag.
27. Kim S.-W. and Choe K.-M. 2011. *String analysis as an abstract interpretation*. In Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation, VMCAI '11. Pages 294-308. Springer-Verlag.
28. Kirkegaard C., Møller A. 2005. *Type Checking with XML Schema in Xact*. Technical Report RS-05-31. Presented at Programming Language Technologies for XML (PLAN-X).
29. Kirkegaard C., Møller A. 2006. *Static Analysis for Java Servlets and JSP*. In Proceedings of the 13th International Static Analysis Symposium, SAS '06. Volume 4134. Springer-Verlag.
30. Kirkegaard C., Møller A., Schwartzbach M.I. 2004. *Static Analysis of XML Transformations in Java*. IEEE Transactions on Software Engineering. Volume 30. Issue 3. Pages 181 - 192.
31. Logozzo F. and Fähndrich M. 2008. *Pentagons: A weakly relational domain for the efficient validation of array accesses*. In Proceedings of the 2008 ACM symposium on Applied computing, SAC '08. Pages 184-188. ACM.
32. Minamide Y. 2005. *Static approximation of dynamically generated web pages*. In Proceedings of the 14th international conference on World Wide Web, WWW '05. Pages 432-441. ACM.
33. Miné A. 2006. *The octagon abstract domain*. Higher-Order and Symbolic Computation. Volume 19, Issue 1. Pages 31-100. Kluwer Academic Publishers.
34. Møller A., Schwarz M. 2011. *HTML Validation of Context-Free Languages*. In Proceedings of the 14th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS '11. Springer-Verlag.
35. Møller A., Olesen M.Ø., Schwartzbach M.I. 2007. *Static Validation of XSL Transformations*. ACM Transactions on Programming Languages and Systems. Volume 29. Issue 4.
36. Møller A., Schwartzbach M.I. 2005. *The Design Space of Type Checkers for XML Transformation Languages*. In Proceedings of the 10th International Conference on Database Theory, ICDT '05. Volume 3363. Pages 17 - 36. Springer-Verlag.
37. Møller A., Schwartzbach M.I. 2011. *XML Graphs in Program Analysis*. Science of Computer Programming. Volume 76, Issue 6. Pages 492 - 515. Elsevier.
38. Tabuchi N., Sumii E., and Yonezawa A. 2002. *Regular expression types for strings in a text processing language*. In Electronic Notes in Theoretical Computer Science, 75. Pages 95-113.
39. Thiemann P. 2005. *Grammar-based analysis of string expressions*. In Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation, TLDI '05. Pages 59 - 70. ACM.
40. Van Hentenryck P., Cortesi A., and Le Charlier B. 1995. *Type analysis of prolog using type graphs*. In Journal of Logic Programming. Volume 22, Issue 3. Pages 179209. Elsevier.
41. Yu F., Bultan T., Cova M., and Ibarra O. 2008. *Symbolic string verification: An automata-based approach*. In Proceedings of the 15th international workshop on Model Checking Software, SPIN '08. Pages 306 - 324. Springer-Verlag.
42. Yu F., Bultan T., Ibarra O. 2010. *Relational String Verification Using Multi-track Automata*. In Proceedings of the 15th International Conference CIAA 2010. Volume 6482. Springer.

43. Yu F., Bultan T., Hardekopf B. 2011. *String Abstractions for String Verification*. In Proceedings of the 18th International SPIN Workshop. Pages 20 - 37. Springer.
44. Zanioli M., Ferrara P., and Cortesi A. 2012. *SAILS: static analysis of information leakage with Sample*. In Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12. Pages 1308-1313. ACM.