Relational String Abstract Domains

Relational String Abstract Domains

Vincenzo Arceri[1], Martina Olliaro[2], Agostino Cortesi[2], and Pietro Ferrara[2]

[1] University of Parma, Italy
vincenzo.arceri@unipr.it
[2] Ca' Foscari University of Venice, Italy
{martina.olliaro,cortesi,pietro.ferrara}@unive.it

Abstract. In modern programming languages, more and more functionalities, such as reflection and data interchange, rely on string values. String analysis statically computes the set of string values that are possibly assigned to a variable, and it involves a certain degree of approximation. During the last decade, several abstract domains approximating string values have been introduced and applied to statically analyze programs. However, most of them are not precise enough to track relational information between string variables whose value is statically unknown (e.g., user input), causing the loss of relevant knowledge about their possible values. This paper introduces a generic approach to formalize relational string abstract domains based on ordering relationships. We instantiate it to several domains built upon different well-known string orders (e.g., substring). We implemented the domain based on the substring ordering into a prototype static analyzer for Go, and we experimentally evaluated its precision and performance on some real-world case studies.

Keywords: Relational abstract domains · Static analysis · String analysis · Abstract interpretation.

1 Introduction

String values play a fundamental role in most programming languages. Dynamically inspecting and modifying objects, transforming text into executable code at run-time, and handling data interchange formats (e.g., XML, JSON) are only a few examples of scenarios where strings are heavily used.

The static analysis community has spent a great effort in proposing new abstractions to better approximate and analyze string values. Unfortunately, almost all the existing string abstract domains are in a position to track information of single variables used in a program (e.g., if a string contains some characters, or if it starts with a given sequence), without inspecting their relationship with other values (e.g., if a string is a substring of another one, despite their actual values are unknown). Detecting relational information between variables is critical in vulnerability analysis, e.g., malware detection, or to verify if the string values manipulated by a program comply with specified consistency constraints.

```
func secName(name, pr1, pr2 string) {
  if hasPrefix(name, pr1) {
    return pr2 + name[4:]
  } else if hasPrefix(name, pr2) {
    return pr1 + name[4:]
  } else {
    return name
  }
}
```

Fig. 1: `secName` function.

For numerical values, advanced and sophisticated relational abstractions have been studied and improved over the years to track relations between variables. A representative example is the Polyhedra abstract domain [19], which has been continuously and heavily improved over the years, as reported by the more recent important works on its optimization, e.g., [9].

For string values not much attention has been given to a systematic design of relational domains. We illustrate the problem by considering the function `secName`[3] in Fig. 1. The function takes as input three arguments of type string, `name`, `pr1` and `pr2`. Then, if `name` has `pr1` as a prefix, the function returns `pr2` concatenated to the substring of `name` starting at index 4. Function `secName` behaves analogously when `name` starts with `pr2`, concatenating `pr1` to `name[4:]`. Otherwise, `name` is returned. The relational information we aim to capture here is the one relating `pr1` and `pr2` with `name` and the returned value. In particular, we want to infer that `name[4:]` is always contained in the returned value, and `pr1` (resp. `pr2`) is contained in the returned value if `name` starts with `pr2` (resp. `pr1`). Using non-relational abstract domains, there is no way to catch these relations. It is clear that using relational domains considerably improves the accuracy of any static analyzer, and the issue of providing a systematic construction of them deserves to be deeply investigated.

1.1 Paper Contribution

In this paper, we define a constructive method upon which relational strings abstract domains can be defined. We start from a string order of interest, and we introduce a suite of relational abstract domains fitting the proposed framework, based on length inequality, character inclusion, substring relations. Precisely, we first formalize how to track relations between single string variables; then, we extend the method to infer relations between string expressions and variables to improve the analysis's precision.

Abstract domains tracking relations among variables may lose information about the values (i.e., the content) of each variable and the only relational information may not be enough to precisely answer about programs of interest. Nevertheless, one standard way to cope with this problem (exploited also in the numerical world) is to combine the *relational* and *non-relational* abstractions

[3] `secName` is the result of a slight modification made to the function available at `https://www.codota.com/code/java/classes/java.lang.String`

by using Cartesian or reduced products [15]. One of these combinations is the Pentagons abstract domain [29], which combines intervals (non-relational information) with the strict upper bounds abstract domain (relational information) by means of the reduced product. Also in this paper, we rely on abstract domain combinations. In particular, we propose two combinations with our substring relational abstract domain, discussing the benefits of them: one with the constant propagation analysis and one with Tarsis [32], a non-relational finite-state automata-based string domain.

The design of relational string abstract domains is agnostic w.r.t. the analyzed programming language. Therefore, our formalization targets a core imperative language, while the examples and experimentation are based on real-word programming languages, namely Go (`https://golang.org/`), a multi-paradigm language heavily used for developing smart contracts for blockchains.

We implemented our framework[4] and instantiated it with the substring relation using a prototype static analyzer for Go. The experimental results show that the accuracy of our system outperforms state-of-the-art string analyses, as well as the scalability of our proposal.

1.2 Paper Structure

Sect. 2 discusses related work. Sect. 3 recalls some background definitions. Sect. 4 shows a core language for string-manipulating programs. Sect. 5 formalizes the construction of generic relational string abstract domains based on a given textual order (Sect. 5.1), and a suite of instantiations capturing different relational properties (Sect. 5.2-5.4). In particular, Sect. 5.4 will present the substring relational domain $\mathsf{Sub}^\star$, which tracks the set of expressions that are definitely substrings of each program variable. Sect. 6 presents the results of our experimental evaluation on $\mathsf{Sub}^\star$. Sect. 7 concludes.

2 Related Work

For numerical values, several relational abstract domains have been proposed, such as Polyhedra [19], Octagons [31], Pentagons [29], and Stripes [20]. Overall, this work line inspired our approach and, in particular, the string relational domains that we will define in Sect. 5. Indeed, consider the Octagons and the Pentagons abstract domains. Octagons track relations of the form $\pm x \pm y \leqslant k$, where k is a constant. Pentagons, a less precise domain than Octagons, combine the numerical properties tracked by the Interval domain (i.e., $x \in [n, m]$) and the symbolic ones captured by the Strict Upper Bound domain (i.e., $x < y$). Similar to the Strict Upper Bound domain, our framework instantiates domains that track information of the form $x \preceq y$, where $\preceq$ is a general partial order over string variables. Moreover, the framework extension we define to track relations between string expressions and variables, like $x + y \preceq z$, has been modelled similarly to Octagons. Other abstractions have been proposed to infer information

[4] Available at `https://github.com/UniVE-SSV/go-lisa`

about the relations between heap-allocated data structures a program manipulates [37]. In [23], an abstract domain that approximates "must" and "may" equalities among pointer expressions has been defined. A relational abstract domain for shape analysis has been presented in [24], built on the top of a set of logical connectives, that represents relations among memory states.

On the string approximation side, a significant effort has been applied to improve the accuracy of the abstraction. However, contrary to the numerical world, most of the existing string abstractions only focus on the approximation of a single variable. Such non-relational abstract domains were already introduced a decade ago [14, 13], such as Character Inclusion, Prefix, and Suffix. Precisely, they track the characters possibly and certainly contained in a string, its prefix, and suffix, respectively. The finite-state automata abstract domain [5, 7] is a sophisticated domain that abstracts a string set as the minimum automaton recognizing it. Even if it can keep information on programs that rely heavily on string manipulation (such as the ones using `eval` [7]) it suffers from scalability problems. M-String [11] is a (non-relational) parametric abstract domain for strings in C. In particular, it uses an abstract domain for the content of a string and an abstract domain for expressions, inferring when a string index position corresponds to an expression of the considered abstract domain. Other general-purpose string abstractions [30, 2, 39] or string abstract domains targeting a specific language [11, 25, 27, 26, 34, 4] have been proposed. The abstract domains we will introduce instead are general-purpose and can be adapted for analyzing programs written in different programming languages. Note that our framework can be easily instantiated with other basic string abstract domains leading to even more precise analyses. Precisely, we start by defining a framework from which domains capturing relations between string variables can be instantiated, and we proceed by extending it for tracking relations between string variables and expressions, enhancing the precision of the analysis. As future work, it could be interesting to study the similarities between our proposal and the subterm domain proposed in [22], a weakly relational abstract domain that infers syntactic equivalences among sub-expressions. For instance, our enhanced framework instantiated with the substring order could be seen as the reduced product [16] between the basic substring domain we propose and the subterm domain.

Besides the string analysis context, which has the advantage of not relying on SMT solvers, string abstractions are heavily used, among others, for string constraint solving. In particular, several works have been proposed on studying decidable fragments of string constraint formulas [1], and researching effective procedures to string constraints verification [36, 38, 1, 3, 39]. For example, a recent work [3] approximates strings as a *dashed string*, namely a sequence of concatenated blocks that specify the number of times the characters they contain must/may appear.

3 Background

String Notation. Given an alphabet of symbols Σ, a string is a sequence of zero or more symbols and it is denoted by σ. The Kleene-closure of Σ, denoted by Σ^*, is the set of any string of finite length over the alphabet Σ. The empty string is denoted by ϵ. Given $\sigma, \sigma' \in \Sigma^*$, we denote by $|\sigma|$ the length of σ, by $\sigma \cdot \sigma'$ the concatenation of σ with σ'. Given $\sigma \in \Sigma^*$ and $i \in [0, |\sigma - 1|]$, we denote by σ_i the symbol at the i-th position of σ. Given $\sigma \in \Sigma^*$ and $i, j \in [0, |\sigma|]$, with $i \leq j < |\sigma|$, we denote by $\sigma_i \dots \sigma_j$ the substring from i to j of σ, and by $\sigma' \curvearrowright \sigma$ if σ' is a substring of σ, i.e., $\exists i, j \in \mathbb{N}.\ 0 \leq i \leq j \leq |\sigma - 1|, \sigma_i \dots \sigma_j = \sigma'$. Note that $\curvearrowright \subseteq \Sigma^* \times \Sigma^*$ is a partial order. Given $\sigma, \sigma' \in \Sigma^*$ such that $\sigma' \curvearrowright \sigma$ we denote by $\mathsf{idx}(\sigma, \sigma')$ the position of the first occurrence of σ' in σ.

Order Theory. A pre-order is a reflexive and transitive binary relation, and if it is also antisymmetric it is called a partial order. A set L with a partial ordering relation $\sqsubseteq \subseteq L \times L$ is a poset and it is denoted by $\langle L, \sqsubseteq \rangle$. A poset $\langle L, \sqsubseteq, \sqcup, \sqcap \rangle$, where $\sqcup$ and $\sqcap$ are respectively the least upper bound (lub) and greatest lower bound (glb) operators of L, is a lattice if $\forall x, y \in L$ we have that $x \sqcup y$ and $x \sqcap y$ belong to L. We say that a lattice is also complete when for each $X \subseteq L$ we have that $\bigsqcup X, \bigsqcap X \in L$. Any finite lattice is a complete lattice. A complete lattice L, with ordering $\sqsubseteq$, lub $\sqcup$, glb $\sqcap$, greatest element (top) $\top$, and least element (bottom) $\bot$ is denoted by $\langle L, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$.

Abstract Interpretation. Abstract interpretation [15, 17] is a theory to soundly approximate program semantics, focusing on some run-time property of interest. The concrete and the abstract semantics are defined over two complete lattices, respectively called the concrete domain C and abstract domain A. Let C and A be complete lattices, a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ forms a *Galois Connection* (GC) between C and A if for every $x \in C$ and for every $y \in A$ we have $\alpha(x) \sqsubseteq_A y \Leftrightarrow x \sqsubseteq_C \gamma(y)$. We denote a Galois Connection by (C, α, γ, A). According to Prop. 7 of [18], a GC between two complete lattices A and C can be induced also if the abstraction function is a complete join preserving map, i.e., $\alpha(\bigcup X) = \bigsqcup \{\alpha(x) \mid x \in X\}$, with $X \subseteq C$. Given (C, α, γ, A), a concrete function $f : C \to C$ is, in general, not computable. Hence, an abstract function $f^\sharp : A \to A$ must correctly approximate the concrete function f. If so, we say that $f^\sharp$ is sound. Formally, given (C, α, γ, A) and a concrete function $f : C \to C$, an abstract function $f^\sharp : A \to A$ is sound w.r.t. f if $\forall c \in C. \alpha(f(c)) \sqsubseteq_A f^\sharp(\alpha(c))$.

4 The Imp Language

In this section, we briefly introduce a very generic imperative language providing the basic operators on strings, as a reference programming language for the rest of the paper. We consider the core running language Imp, whose syntax is given in Fig. 2. Imp is an imperative language handling arithmetic, Boolean, and string

$\mathtt{a} \in \text{AE} ::= x \mid n \mid \mathtt{a + a} \mid \mathtt{a - a} \mid \mathtt{a * a} \mid \mathtt{a / a} \mid \mathtt{length(s)} \mid \mathtt{indexOf(s,s)}$

$\mathtt{b} \in \text{BE} ::= x \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{b \&\& b} \mid \mathtt{b} \mid\mid \mathtt{b} \mid \mathtt{!\ b} \mid \mathtt{e} < \mathtt{e} \mid \mathtt{e == e}$
$\quad \mid \mathtt{contains}(\mathtt{s}_1,\mathtt{s}_2)$

$\mathtt{s} \in \text{SE} ::= x \mid \mathtt{"}\sigma\mathtt{"} \mid \mathtt{substr(s,a,a)} \mid \mathtt{s}_1 \mathtt{\ +\ } \mathtt{s}_2$

$\mathtt{e} \in \text{E} ::= \mathtt{a} \mid \mathtt{b} \mid \mathtt{s}$

$\mathtt{st} \in \text{STMT} ::= {}^{\ell_1}\mathtt{st}\ {}^{\ell_2}\mathtt{st}^{\ell_3} \mid {}^{\ell_1}\mathtt{skip;}^{\ell_2} \mid {}^{\ell_1}x \mathtt{\ =\ e;}^{\ell_2}$
$\quad \mid {}^{\ell_1}\mathtt{if\ (b)\ \{}\ {}^{\ell_2}\mathtt{st}^{\ell_3}\ \mathtt{\}\ else\ \{}\ {}^{\ell_4}\mathtt{st}^{\ell_5}\ \mathtt{\}}^{\ell_6}$
$\quad \mid {}^{\ell_1}\mathtt{while\ (b)\ \{}\ {}^{\ell_2}\mathtt{st}^{\ell_3}\ \mathtt{\}}^{\ell_4}$

$\mathtt{P} \in \text{IMP} ::= {}^{\ell_1}\mathtt{st}\ {}^{\ell_2}$

where $x \in X$ (finite set of variables), $n \in \mathbb{Z}$ and $\sigma \in \Sigma^*$

Fig. 2: IMP syntax.

expressions. Its basic values are integers, booleans, and strings, ranging over $\mathbb{Z}$, $\{\mathtt{true}, \mathtt{false}\}$ and Σ^*, respectively. We consider four string operations, `length`, `indexOf`, `contains`, and `substr` that respectively compute (i) the length of a given string, (ii) the index of the first occurrence of a string in another one, (iii) if a string is contained in another one, and (iv) the substring of a given string between two specified indexes. Let P be an IMP program. Each IMP statement is annotated with a label $\ell \in \mathit{Lab}_{\mathsf{P}}$ (not belonging to the syntax), where $\mathit{Lab}_{\mathsf{P}}$ denotes the set of the P labels, i.e., its program points.

As usual in static analysis, a program can be analyzed by looking at its control-flow graph (CFG for short), i.e., a directed graph that embeds the control structure of a program, where nodes are the program points, and edges express the flow paths from the entry to the exit block. Following [35], given a program $\mathsf{P} \in \text{IMP}$, we define the corresponding CFG $\mathtt{G}_{\mathsf{P}} \triangleq \langle \mathit{Nodes}_{\mathsf{P}}, \mathit{Edges}_{\mathsf{P}}, \mathit{In}_{\mathsf{P}}, \mathit{Out}_{\mathsf{P}} \rangle$ as the CFG whose nodes are the program points, i.e., $\mathit{Nodes}_{\mathsf{P}} \triangleq \mathit{Lab}_{\mathsf{P}}$, In_{P} is the entry program point, and $\mathit{Out}_{\mathsf{P}}$ is the last program point. The algorithm computing the CFG of a program P is standard and can be found in [35, 6]. An example of CFG is depicted in Fig. 3. A CFG embeds the control structure of the program. Hence, to define the behavior of a CFG, it is enough to formalize the semantics of the edge labels, namely $\text{IMP}^{\text{CFG}} ::= \mathtt{skip} \mid x = \mathtt{e} \mid \mathtt{b}$, expressing the effect that each edge has from its entry node to its exit node. Let $\text{VAL} \triangleq \mathbb{Z} \cup \Sigma^* \cup \{\mathtt{true}, \mathtt{false}\}$ be the set of the possible values associated with a variable. Let $\mathtt{m} \in \mathbb{M} \triangleq X \to \text{VAL}$ be the set of (finite) memories, where $\mathtt{m}_{\varnothing} = \varnothing$ is the empty memory. The semantics of expressions is captured by the function $(\!|\ \mathtt{e}\ |\!) : \mathbb{M} \to \text{VAL}$. Since the semantics of integer and Boolean expressions are standard (and not of interest to this paper),

```
ℓ1 x = "a"; ℓ2 y = "c";
ℓ3 while (B) {
   ℓ4 x = x + "a"; ℓ5
}ℓ6
```

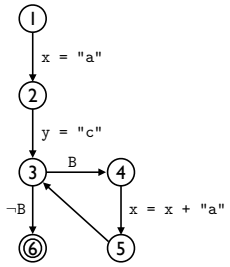

Fig. 3: Example of CFG generation.

in the following, we only give the concrete semantics of string expressions.

$$\begin{aligned}
&\llparenthesis x \rrparenthesis \mathbb{m} = \mathbb{m}(x) \qquad \llparenthesis \sigma \rrparenthesis \mathbb{m} = \sigma \qquad \llparenthesis \mathsf{s}_1 + \mathsf{s}_2 \rrparenthesis \mathbb{m} = \llparenthesis \mathsf{s}_1 \rrparenthesis \mathbb{m} \cdot \llparenthesis \mathsf{s}_2 \rrparenthesis \mathbb{m} \\
&\llparenthesis \texttt{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2) \rrparenthesis \mathbb{m} = \sigma_i \dots \sigma_j \\
&\qquad \text{where } \sigma = \llparenthesis \mathsf{s} \rrparenthesis \mathbb{m}, i = \llparenthesis \mathsf{a}_1 \rrparenthesis \mathbb{m}, j = \llparenthesis \mathsf{a}_2 \rrparenthesis \mathbb{m}, 0 \leq i \leq j < |\sigma| \\
&\llparenthesis \texttt{length}(\mathsf{s}) \rrparenthesis \mathbb{m} = |\llparenthesis \mathsf{s} \rrparenthesis \mathbb{m}| \\
&\llparenthesis \texttt{contains}(\mathsf{s}_1, \mathsf{s}_2) \rrparenthesis \mathbb{m} = \llparenthesis \mathsf{s}_2 \rrparenthesis \mathbb{m} \curvearrowright \llparenthesis \mathsf{s}_1 \rrparenthesis \mathbb{m} \\
&\llparenthesis \texttt{indexOf}(\mathsf{s}_1, \mathsf{s}_2) \rrparenthesis \mathbb{m} = \begin{cases} \mathsf{idx}(\llparenthesis \mathsf{s}_1 \rrparenthesis \mathbb{m}, \llparenthesis \mathsf{s}_2 \rrparenthesis \mathbb{m}) & \text{if } \llparenthesis \mathsf{s}_2 \rrparenthesis \mathbb{m} \curvearrowright \llparenthesis \mathsf{s}_1 \rrparenthesis \mathbb{m} \\ -1 & \text{otherwise} \end{cases}
\end{aligned}$$

Note that when the indexes of `substr` are out-of-bounds its semantics is undefined and the execution stops as usual with standard concrete semantics in case of runtime errors. We are finally in the position to formalize the edges label semantics. Abusing the notation, we define the function $\llparenthesis \mathsf{st} \rrparenthesis : \mathbb{M} \to \mathbb{M}$ to capture the semantics of the elements of $\textsc{Imp}^{\textsf{CFG}}$.

$$\begin{aligned}
&\llparenthesis \texttt{skip} \rrparenthesis \mathbb{m} = \mathbb{m} \qquad \llparenthesis x = \mathsf{e} \rrparenthesis \mathbb{m} = \mathbb{m}[x \leftarrow \llparenthesis \mathsf{e} \rrparenthesis \mathbb{m}] \\
&\llparenthesis \mathsf{b} \rrparenthesis \mathbb{m} = \begin{cases} \mathbb{m} & \text{if } \llparenthesis \mathsf{b} \rrparenthesis \mathbb{m} = \texttt{true} \\ \mathbb{m}_\varnothing & \text{if } \llparenthesis \mathsf{b} \rrparenthesis \mathbb{m} = \texttt{false} \end{cases}
\end{aligned}$$

As far as Boolean expressions are concerned, the semantics propagates the input memory if the Boolean expression holds, the empty memory otherwise.

Finally, a store is a collection of memories for each program point, defined as $\mathbb{s} \in \mathbb{S} \triangleq Lab_{\mathsf{P}} \to \mathbb{M}$ and it associates a memory to each program point.

Static analysis computes invariants for each program point. Thus, we first define a collecting semantics which relates each program point (i.e., each node of a CFG) to the set of the possible memories holding at that program point. This boils down to lifting the concrete semantics $\llparenthesis \mathsf{st} \rrparenthesis : \mathbb{M} \to \mathbb{M}$ (working on single memories), to the collecting semantics $[\![\mathsf{st}]\!] : \wp(\mathbb{M}) \to \wp(\mathbb{M})$ working on sets of memories. Thus, a collecting store mapping each program point to a set of memories is $\overline{\mathbb{s}} \in \overline{\mathbb{S}} \triangleq Lab_{\mathsf{P}} \to \wp(\mathbb{M})$.

Finally, we can apply standard fix-point analysis algorithms [35] which returns a store $\overline{\mathbb{s}}$ such that, for each $\ell \in Lab_{\mathsf{P}}$, $\overline{\mathbb{s}}(\ell)$ is the fix-point collecting

semantics (i.e., a set of memories) holding at ℓ. However, the set of the possible values for each variable, and for each node of a CFG, are not computable because of Rice's Theorem. Hence, we need abstractions to make static analysis decidable.

5 A Suite of String Relational Abstract Domains

This section provides a suite of relational string abstract domains based on several well-known orders over strings. We start by proposing a general framework to build string relational abstract domains parametrized on a given string order. Within this framework, we present three different string relational abstract domains: length inequality, character inclusion, and substring domains, with the corresponding abstract semantics of Imp.

5.1 General Relational Framework

We aim at capturing relations between string variables of the form $y \preceq x$ w.r.t. a given (partial or pre-order) relation $\preceq$ over strings, such as "the variable y is a substring of the variable x". As introduced in Sect. 2, in the numerical world such a relation is captured by the (strict) upper bound abstract domain [31, 29], which expresses relations of the form $y \leq x$. In this section, we generalize the upper bound abstract domain to string variables, making it parametric w.r.t. a given string order.

Our starting point is a (pre or partial) order $\preceq_{\Sigma^*} \subseteq \Sigma^* \times \Sigma^*$ between strings. Then, given a Imp program P we aim to analyze, we abuse notation denoting by $X_{str} \subseteq X$ the set of string variables used by the program P. Note that the set of string variables used by an Imp program is always finite. At this point, we build a new order $\preceq \subseteq X_{str} \times X_{str}$ between a pair of string variables, built upon $\preceq_{\Sigma^*}$. Finally, we design a relational string abstract domain based on $\preceq$.

Definition 1 (General string relational abstract domain). *Let $\preceq \subseteq X_{str} \times X_{str}$ be an order over string variables. The general string relational abstract domain $\mathcal{A}$ is defined as $\mathcal{A} \triangleq \wp(\{y \preceq x \mid x, y \in X_{str}\}) \cup \{\bot_{\mathcal{A}}\}$, where the top element, denoted by $\top_{\mathcal{A}}$, corresponds to the empty set $\varnothing$ and the bottom element is represented by the special element $\bot_{\mathcal{A}}$. The least upper bound, greatest lower bound, and the partial order of $\mathcal{A}$ are defined as follows*[5]*:*

$$A_1 \sqcup_{\mathcal{A}} A_2 \triangleq \begin{cases} A_1 & \text{if } A_2 = \bot_{\mathcal{A}} \\ A_2 & \text{if } A_1 = \bot_{\mathcal{A}} \\ \mathsf{Clos}(\{y \preceq x \mid y \preceq x \in A_1 \wedge y \preceq x \in A_2\}) & \text{otherwise} \end{cases}$$

$$A_1 \sqcap_{\mathcal{A}} A_2 \triangleq \begin{cases} \bot_A & \text{if } A_1 = \bot_{\mathcal{A}} \vee A_2 = \bot_{\mathcal{A}} \\ \{y \preceq x \mid y \preceq x \in A_1 \vee y \preceq x \in A_2\} & \text{otherwise} \end{cases}$$

$$A_1 \sqsubseteq_{\mathcal{A}} A_2 \iff A_1 = \bot_A \vee (A_1 \neq \bot_A \wedge A_2 \neq \bot_A \wedge A_1 \supseteq A_2)$$

[5] In general, while $\preceq$ (order on string variables) can be a pre or partial order, $\sqsubseteq_{\mathcal{A}}$ (order on the abstract domain $\mathcal{A}$) is always a partial order.

where $\mathsf{Clos} : \mathcal{A} \to \mathcal{A}$ *performs the transitive closure of an abstract element* $A \in \mathcal{A}$*, i.e.,* $\forall x, y, z \in X_{str}$ *if* $x \preceq y, y \preceq z \in A$*, then the function* Clos *returns a new abstract element containing all the relations of* A *adding the relation* $x \preceq z$*. In the least upper bound, when one of the elements is bottom, the other is returned, while in the greatest lower bound, when one of the elements is bottom, then bottom is returned. Finally, the partial order captures the fact that the bottom element* $\bot_{\mathcal{A}}$ *is the least element of* $\mathcal{A}$*.*

The abstract domain $\mathcal{A}$ is intended to collect $\preceq$-*must* relations, i.e., informally speaking, if a relation $y \preceq x$ is captured in the abstract world, it means that it surely holds in the concrete world.

Note that elements of $\mathcal{A}$ are sets of relations $y \preceq x$ between string variables. Moreover, the general abstract domain $\mathcal{A}$ is finite, given that the set of string variables used by the program we aim to analyze is finite and, in turn, also the number of possible relations. Thus, it is straightforward to prove that the domain $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}, \sqcup_{\mathcal{A}}, \sqcap_{\mathcal{A}}, \bot_{\mathcal{A}}, \top_{\mathcal{A}})$ is a complete lattice and that its least upper bound $\sqcup_{\mathcal{A}}$ and greatest lower bound $\sqcap_{\mathcal{A}}$ are defined as the intersection and union between abstract elements, respectively. Abstraction and concretization functions $\alpha^{\mathcal{A}} : \wp(\mathbb{M}) \to \mathcal{A}$ and $\gamma^{\mathcal{A}} : \mathcal{A} \to \wp(\mathbb{M})$ are defined as follows:

$$\alpha^{\mathcal{A}}(\mathsf{M}) \triangleq \begin{cases} \bot_{\mathcal{A}} & \text{if } \mathsf{M} = \varnothing \\ \{y \preceq x \mid \forall \mathtt{m} \in \mathsf{M}.\ \mathtt{m}(y) \preceq_{\Sigma^*} \mathtt{m}(x),\ x, y \in X_{str}\} & \text{otherwise} \end{cases} \tag{1}$$

$$\gamma^{\mathcal{A}}(A) \triangleq \begin{cases} \varnothing & \text{if } A = \bot_{\mathcal{A}} \\ \wp(\mathbb{M}) & \text{if } A = \top_{\mathcal{A}} \\ \bigcap_{y \preceq x \in A} \{\mathtt{m} \mid \mathtt{m}(x), \mathtt{m}(y) \in \Sigma^*, \mathtt{m}(y) \preceq_{\Sigma^*} \mathtt{m}(x)\} & \text{otherwise} \end{cases} \tag{2}$$

where we recall that $\preceq_{\Sigma^*}$ denotes an order over Σ^*. The abstraction function takes as input a set of memories M and returns the least set of relations that holds in any memory $\mathtt{m} \in \mathsf{M}$. Instead, the concretization function takes as input an element A of the general string relational abstract domain $\mathcal{A}$ and returns the empty set if $A = \bot_{\mathcal{A}}$, the set of any possible concrete memory if $A = \top_{\mathcal{A}}$, and the least set of concrete memories where all the relations contained in A holds, otherwise. $(\wp(\mathbb{M}), \alpha^{\mathcal{A}}, \gamma^{\mathcal{A}}, \mathcal{A})$ is a Galois Connection, since $\wp(\mathbb{M})$ and $\bot_{\mathcal{A}}$ are complete lattices and $\alpha^{\mathcal{A}}$ is a join-morphism.

Running Example: Length Relational Abstract Domain. For instance, one may be interested in capturing the relations concerning the length of a string variable w.r.t. another, when they interact during the program execution. Formally, we are interested in identifying the relation $\preceq_{\mathsf{len}} \subseteq X_{str} \times X_{str}$ between string variables such that, given $x, y \in X_{str}$, $y \preceq_{\mathsf{len}} x$ iff the length of y is smaller than or equal to the length of x. Note that $\preceq_{\mathsf{len}}$ is a partial order, but the string order upon which is based is a pre-order. Indeed, two strings may have the same length, but may not represent the same sequence of characters (the anti-symmetric property does not hold). For this reason, when we have that $x \preceq_{\mathsf{len}} y$ and $y \preceq_{\mathsf{len}} x$,

we can assert that x and y have the same length but we cannot assert that the strings tracked by the variables are equal.

We instantiate the general abstract domain of Def. 1 over the pre-order $\preceq_{\mathsf{len}}$. In particular, we replace the general string order $\preceq$ with $\preceq_{\mathsf{len}}$, obtaining the relational string length abstract domain $\mathsf{Len} \triangleq \wp(\{y \preceq_{\mathsf{len}} x \mid x, y \in X_{str}\}) \cup \{\bot_{\mathsf{len}}\}$, where the top element, denoted by $\top_{\mathsf{len}}$, is the empty set $\varnothing$, and $\bot_{\mathsf{len}}$ is a special element denoting the bottom element. The least upper bound and greatest lower bound operators $\sqcup_{\mathsf{len}}$ and $\sqcap_{\mathsf{len}}$ and the partial order $\sqsubseteq_{\mathsf{len}}$ (over Len) can be obtained by replacing any occurrence of $\preceq$ with $\preceq_{\mathsf{len}}$ in their general definition in Def. 1.

Lemma 1. $(\mathsf{Len}, \sqsubseteq_{\mathsf{len}}, \sqcup_{\mathsf{len}}, \sqcap_{\mathsf{len}}, \bot_{\mathsf{len}}, \top_{\mathsf{len}})$ *is a complete lattice.*

We define the abstraction $\alpha^{\mathsf{len}} : \wp(\mathbb{M}) \to \mathsf{Len}$ and the concretization $\gamma^{\mathsf{len}} : \mathsf{Len} \to \wp(\mathbb{M})$ functions of the relational string length abstract domain instantiating Eq. 1 and 2 replacing $\preceq_{\Sigma^*}$ with $\preceq_{\mathsf{len}}$.

$$\alpha^{\mathsf{len}}(\mathsf{M}) \triangleq \begin{cases} \bot_{\mathsf{len}} & \text{if } \mathsf{M} = \varnothing \\ \{y \preceq_{\mathsf{len}} x \mid \forall \mathtt{m} \in \mathsf{M}.\ |\mathtt{m}(y)| \leq |\mathtt{m}(x)|,\ x, y \in X_{str}\} & \text{otherwise} \end{cases}$$

$$\gamma^{\mathsf{len}}(\mathcal{L}) \triangleq \begin{cases} \varnothing & \text{if } \mathcal{L} = \bot_{\mathsf{len}} \\ \wp(\mathbb{M}) & \text{if } \mathcal{L} = \top_{\mathsf{len}} \\ \bigcap_{y \preceq_{\mathsf{len}} x \in \mathcal{L}} \{\mathtt{m} \mid \mathtt{m}(x), \mathtt{m}(y) \in \Sigma^*,\ |\mathtt{m}(y)| \leq |\mathtt{m}(x)|\} & \text{otherwise} \end{cases}$$

Theorem 1. $(\wp(\mathbb{M}), \alpha^{\mathsf{len}}, \gamma^{\mathsf{len}}, \mathsf{Len})$ *is a Galois Connection.*

Proof. The Galois Connection's existence comes from the fact that both $\wp(\mathbb{M})$ and Len are complete lattices, and α^{len} is a join-morphism (Prop. 7 of [18]).

At this point, we define a general and parametric abstract semantics of Imp. In particular, given an abstract domain $\mathcal{A}$, built upon the order $\preceq$ as shown in Def. 1, we define the function $[\![\ \mathsf{st}\ ]\!]^{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$, capturing the $\preceq$-relations between string variables generated by the statement st. We start by defining the parametric abstract semantics of the assignment $x = \mathsf{s}$. Here, the crucial point is the definition of the auxiliary function $\mathsf{extr} : \textsc{se} \to \wp(X_{str})$ that, given a string expression s, extracts all the variables *syntactically* appearing in s that are related w.r.t. $\preceq$ with s, i.e., it approximates the set of variables that are $\preceq$-related with s.

Extraction Function of Len. Given $x = \mathsf{s}$, we can see the string expression s as an ordered list of concatenated expressions $\mathsf{s}_0, \mathsf{s}_1, \ldots, \mathsf{s}_n$, and the string variables that surely have length less than or equal of x are the ones at the top-level of a concatenation appearing in s. For instance, consider the assignment $\mathtt{x} = \mathtt{y} + \mathtt{z} + \mathtt{w}$. The relations we aim to capture from it are $y \preceq_{\mathsf{len}} x$, $z \preceq_{\mathsf{len}} x$ and $w \preceq_{\mathsf{len}} x$, that is y, z, w have length less than or equal to the length of x. These variables are collected by the function $\mathsf{extr} : \textsc{se} \to \wp(X_{str})$, which extracts the variables that syntactically appears at the top-level of a string expression.

$$\mathsf{extr}(\mathsf{s}) = \begin{cases} \{y\} & \text{if } \mathsf{s} = y \in X_{str} \\ \mathsf{extr}(\mathsf{s}_1) \cup \mathsf{extr}(\mathsf{s}_2) & \text{if } \mathsf{s} = \mathsf{s}_1 + \mathsf{s}_2 \\ \varnothing & \text{otherwise} \end{cases}$$

Once defined the extraction function extr, we *semantically* interpret the syntactic components it extracts giving the general abstract semantics of the assignment $[\![\ x = \mathsf{s}\ ]\!]^{\mathcal{A}} A$, which is defined by the steps shown below. For the sake of simplicity, we suppose that the input abstract memory A is not $\bot_{\mathcal{A}}$: in this case, $\bot_{\mathcal{A}}$ is simply propagated, skipping the above phases.

- [**remove**]: $A_r = \begin{cases} A \setminus \{w \preceq z \mid w = x, z \in X_{str}\} & \text{if } x \in \mathsf{extr}(\mathsf{s}) \\ A \setminus \{w \preceq z \mid w = x \vee z = x\} & \text{otherwise} \end{cases}$
- [**add**]: $A_a = A_r \cup \{y \preceq x \mid y \in \mathsf{extr}(\mathsf{s})\}$
- [**closure**]: $[\![\ x = \mathsf{s}\ ]\!]^{\mathcal{A}} A = \mathsf{Clos}(A_a)$

The first phase is [**remove**]: given the input memory $A \in \mathcal{A}$, it removes the relations that surely do not hold anymore after the assignment execution. In particular, we always remove the relations of the form $x \preceq z$, for some $z \in X_{str}$, since x is going to be overwritten. Still, we also remove any relations of the form $w \preceq x$, for some $w \in X_{str}$, iff x does not appear at the top-level of the expression s. For instance, consider the fragment `x = w; x = x + y;` and the relational abstract domain Len. From the first assignment, we collect the relation $w \preceq_{\mathsf{len}} x$. This information also holds after the second assignment's execution, since x appears at the top-level of the assignment expression and inherits any previously gathered $\preceq_{\mathsf{len}}$-relation. Hence, in this case, we do not remove the previously gathered relations about the variable x. In the other cases, also the previous length relations of the form $w \preceq_{\mathsf{len}} x$ are removed.

Then, [**add**] adds the $\preceq$-relations $y \preceq x$, for each variable y collected in $\mathsf{extr}(\mathsf{s})$, and [**closure**] performs the transitive closure on the abstract memory obtained from [**add**], i.e., A_a, by means of the function Clos, to derive the implicit $\preceq$-relations not yet present in A_a.

As far as conditional expressions are concerned, the only Imp Boolean expressions that generate $\preceq$-relations for the string domains presented in this paper are $\mathtt{contains}(\mathsf{s}_1, \mathsf{s}_2)$, $\mathsf{s}_1 == \mathsf{s}_2$, conjunctive and disjunctive expressions. Note that, given the expression $\mathtt{contains}(\mathsf{s}_1, \mathsf{s}_2)$, we infer $\preceq$-relations only when s_1 is a variable, otherwise no other information is gathered. As in the assignment abstract semantics, we suppose that the input abstract memory A is not equal to $\bot_{\mathcal{A}}$, since in this case the bottom element is simply propagated.

$$[\![\ \mathtt{contains}(x, \mathsf{s})\ ]\!]^{\mathcal{A}} A = \mathsf{Clos}(A \cup \{y \preceq x \mid y \in \mathsf{extr}(\mathsf{s})\})$$

Similarly, we can infer $\preceq$-relations in the abstract semantics of $\mathsf{s}_1 == \mathsf{s}_2$ only when either s_1 or s_2 is a string variable.

$$[\![\ x == \mathsf{s}\ ]\!]^{\mathcal{A}} A = [\![\ \mathsf{s} == x\ ]\!]^{\mathcal{A}} A = \begin{cases} \mathsf{Clos}(A \cup \{y \preceq x, x \preceq y\}) & \text{if } \mathsf{s} = y \in X_{str} \\ \mathsf{Clos}(A \cup \{y \preceq x \mid y \in \mathsf{extr}(\mathsf{s})\}) & \text{otherwise} \end{cases}$$

As far as the semantics of the conjunctive and disjunctive expressions are concerned, we rely on the least upper bound and greatest lower bound operators given in Def. 1.

$$[\![\mathsf{e}_1 \ \&\& \ \mathsf{e}_2]\!]^{\mathcal{A}} A = A \cup ([\![\mathsf{e}_1]\!]^{\mathcal{A}} A \sqcup_{\mathcal{A}} [\![\mathsf{e}_2]\!]^{\mathcal{A}} A)$$
$$[\![\mathsf{e}_1 \ || \ \mathsf{e}_2]\!]^{\mathcal{A}} A = A \cup ([\![\mathsf{e}_1]\!]^{\mathcal{A}} A \sqcap_{\mathcal{A}} [\![\mathsf{e}_2]\!]^{\mathcal{A}} A)$$

Unlike the assignment, Boolean expressions' abstract semantics do not remove previous substring relations since they do not alter the (concrete) memory. For the other Boolean expressions, the abstract semantics is the identity, namely $[\![\mathsf{b}]\!]^{\mathcal{A}} A = A$.

Abstract Semantics of Len. The abstract semantics for Len is captured by the function $[\![\mathsf{st}]\!]^{\mathsf{len}} : \mathsf{Len} \to \mathsf{Len}$, that given an input abstract memory returns an abstract memory containing the new string length relations introduced by st, and it is defined by replacing any occurrence of $\preceq$ with $\preceq_{\mathsf{len}}$, in the general abstract semantics definition reported above.

Theorem 2. *The abstract semantics of* Len *is sound. Indeed, it holds that:*

$$\forall \mathsf{M} \in \wp(\mathbb{M}).\ \alpha^{\mathsf{len}}([\![x = \mathsf{s}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{len}} ([\![x = \mathsf{s}]\!]^{\mathsf{len}} \alpha^{\mathsf{len}}(\mathsf{M}))$$
$$\forall \mathsf{M} \in \wp(\mathbb{M}).\ \alpha^{\mathsf{len}}([\![\mathsf{b}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{len}} ([\![\mathsf{b}]\!]^{\mathsf{len}} \alpha^{\mathsf{len}}(\mathsf{M}))$$

Note that this general abstract semantics holds for the abstract domains instantiated and presented in this paper and other abstract domains, derived from other string orders, may define the abstract semantics also for other program constructs that are not considered here. For example, consider the `indexOf` operation. Its abstract semantics over Len does not generate new relations, while this may happen if other relational abstract domains are considered. Also, the extr function may differ from the one presented before if other string relations are considered: for instance, the extr function for the abstract domain based on the prefix relation is slightly different from the one used in Len: given extr(s), it would extract just the string expressions that are prefixes of the s and not any substring.

5.2 Character Inclusion Relational Abstract Domain

Within the formal framework presented above, we are able to generate several relational string abstract domains. In the following, we present the character inclusion relational abstract domain Char, tracking the characters included between a pair of string variables. Given $x, y \in X_{str}$, we are interested in capturing "if all the characters which appear in y occur in x". Formally, we introduce the binary relation $\preceq_{\mathsf{char}} \subseteq X_{str} \times X_{str}$ such that $y \preceq_{\mathsf{char}} x$ iff the set of characters of y is contained or equal to the set of characters of x. Similar to $\preceq_{\mathsf{len}}$ in the Len abstract domain, also $\preceq_{\mathsf{char}}$ is a partial order, based on the character inclusion pre-order between string values. Hence, if we have that $x \preceq_{\mathsf{char}} y$ and $y \preceq_{\mathsf{char}} x$

we can assert that x and y have the same characters but it is not guaranteed that they track the same string value.

We define the relational character inclusion string abstract domain $\mathsf{Char} \triangleq \wp(\{y \preceq_{\mathsf{char}} x \mid x, y \in X_{str}\}) \cup \{\bot_{\mathsf{char}}\}$. The top element is the empty set $\varnothing$, and the bottom element is represented by the special element $\bot_{\mathsf{char}}$.

5.3 Substring Relational Abstract Domain

The abstract domains Len and Char presented in the previous sections track relations about the lengths and the characters of a pair of string variables. The main limitation of these domains is that they are both based on strings pre-orders: hence, as we have already argued before, when we have the relations $x \preceq y$ and $y \preceq x$, we cannot assert that the values tracked by the variables x and y are equal. Moreover, Len loses any information about the *content* of a variable, and Char loses any information about the shape of a variable. We propose then a strictly more precise partial order-based relational string abstract domain, still fitting in the formal framework presented in Sect. 5.1 and solving the problems of Len and Char mentioned before.

Given $x, y \in X_{str}$, let the binary relation $\preceq_{\mathsf{sub}}$: $X_{str} \times X_{str}$ be such that $x \preceq_{\mathsf{sub}} y$ iff x is a substring of y. The relation $\preceq_{\mathsf{sub}}$ is a partial order, being reflexive, transitive and anti-symmetric, as well as the substring relation on which $\preceq_{\mathsf{sub}}$ is based on. Unlike the Len and Char cases, if we have$x \preceq_{\mathsf{sub}} y$ and $y \preceq_{\mathsf{sub}} x$, we can surely assert that the strings tracked by x and y are equal.

At this point, we define the relational string abstract domain $\mathsf{Sub} \triangleq \wp(\{y \preceq_{\mathsf{sub}} x \mid x, y \in X_{str}\}) \cup \{\bot_{\mathsf{sub}}\}$, where the top element is the empty set $\varnothing$, and $\bot_{\mathsf{sub}}$ is a special element representing the bottom element.

5.4 Extension to String Expressions

The abstract domain proposed in Sect. 5.3 can track when a single string variable is a substring of another one. In this section, we show how to improve Sub to catch even more substring relations. In order to highlight the limits of Sub (which Len and Char also suffer from), consider the following fragment: `x = y + y + w; z = y + w;`. If we analyze it with the substring abstract domain, the final abstract memory is $\{y \preceq_{\mathsf{sub}} x, w \preceq_{\mathsf{sub}} x, y \preceq_{\mathsf{sub}} z, w \preceq_{\mathsf{sub}} z\}$. Still, other substring relations may be inferred, such as $z \preceq_{\mathsf{sub}} x$ or $y + w \preceq_{\mathsf{sub}} x$. In the following, we slightly change the substring abstract domain to catch also such relations.

Given an Imp program P we aim to analyze, we recall that X_{str} denotes the finite set of string variables used by P. Similarly, we abuse notation denoting by SE the set of string expressions appearing in P. As X_{str}, also the set of string expressions appearing in P is finite. At this point, we introduce the binary relation $\preceq_{\mathsf{sub}^\star} \subseteq \text{SE} \times X_{str}$ that relates string expressions with string variables. For instance, $y + y \preceq_{\mathsf{sub}^\star} x$ means that the concatenation of y with y is a substring of x. Upon $\preceq_{\mathsf{sub}^\star}$, we build the new set of abstract memories able to relate string expressions to variables. In particular, we define the abstract domain

$$\mathsf{Sub}^\star \triangleq \wp(\{\mathsf{s} \preceq_{\mathsf{sub}^\star} x \mid \mathsf{s} \in \text{SE}, x \in X_{str}\}) \cup \{\bot_{\mathsf{sub}^\star}\}$$

$$\text{Lub: } \mathcal{S}_1^\star \sqcup_{\mathsf{sub}^\star} \mathcal{S}_2^\star \triangleq \begin{cases} \mathcal{S}_2^\star & \text{if } \mathcal{S}_1^\star = \bot_{\mathsf{sub}^\star} \\ \mathcal{S}_1^\star & \text{if } \mathcal{S}_2^\star = \bot_{\mathsf{sub}^\star} \\ \mathsf{Clos}(\{\mathsf{s} \preceq_{\mathsf{sub}^\star} x \mid \mathsf{s} \preceq_{\mathsf{sub}^\star} x \in \mathcal{S}_1^\star \wedge \mathsf{s} \preceq_{\mathsf{sub}^\star} x \in \mathcal{S}_2^\star\}) & \text{otherwise} \end{cases}$$

$$\text{Glb: } \mathcal{S}_1^\star \sqcap_{\mathsf{sub}^\star} \mathcal{S}_2^\star \triangleq \begin{cases} \bot_{\mathsf{sub}^\star} & \text{if } \mathcal{S}_1^\star = \bot_{\mathsf{sub}^\star} \\ & \vee \mathcal{S}_2^\star = \bot_{\mathsf{sub}^\star} \\ \{\mathsf{s} \preceq_{\mathsf{sub}^\star} x \mid \mathsf{s} \preceq_{\mathsf{sub}^\star} x \in \mathcal{S}_1^\star \vee \mathsf{s} \preceq_{\mathsf{sub}^\star} x \in \mathcal{S}_2^\star\} & \text{otherwise} \end{cases}$$

$$\text{Partial order: } \mathcal{S}_1^\star \sqsubseteq_{\mathsf{sub}^\star} \mathcal{S}_2^\star \iff \mathcal{S}_1^\star = \bot_{\mathsf{sub}^\star} \vee (\mathcal{S}_1^\star \neq \bot_{\mathsf{sub}^\star} \wedge \mathcal{S}_2^\star \neq \bot_{\mathsf{sub}^\star} \wedge \mathcal{S}_1^\star \supseteq \mathcal{S}_2^\star)$$

Fig. 4: Lattice operations over $\mathsf{Sub}^\star$.

where the top element is the empty set $\varnothing$, and $\bot_{\mathsf{sub}^\star}$ is a special element presenting the bottom element. We denote by $\mathcal{S}^\star$ an element of $\mathsf{Sub}^\star$. Note that $\mathsf{Sub}^\star$ is still a finite domain, since, given a program $\mathsf{P} \in \textsc{Imp}$, both the string variables and string expressions used by P are finite. Similarly to the previous cases, $(\mathsf{Sub}^\star, \sqsubseteq_{\mathsf{sub}^\star}, \sqcup_{\mathsf{sub}^\star}, \sqcap_{\mathsf{sub}^\star}, \bot_{\mathsf{sub}^\star}, \top_{\mathsf{sub}^\star})$ is a complete lattice, and the definition of its lattice operators and partial order is reported in Fig. 4. The abstraction $\alpha^{\mathsf{sub}^\star} : \wp(\mathbb{M}) \to \mathsf{Sub}^\star$ and concretization $\gamma^{\mathsf{sub}^\star} : \mathsf{Sub}^\star \to \wp(\mathbb{M})$ functions, forming again a Galois Connection, are defined as:

$$\alpha^{\mathsf{sub}^\star}(\mathsf{M}) \triangleq \begin{cases} \bot_{\mathsf{sub}^\star} & \text{if } \mathsf{M} = \varnothing \\ \{\mathsf{s} \preceq_{\mathsf{sub}^\star} x \mid \forall \mathtt{m} \in \mathsf{M}.\, [\![\, \mathsf{s} \,]\!]\mathtt{m} \curvearrowright \mathtt{m}(x), x \in X_{str}, \mathsf{s} \in \textsc{se}\} & \text{otherwise} \end{cases}$$

$$\gamma^{\mathsf{sub}^\star}(\mathcal{S}^\star) \triangleq \begin{cases} \varnothing & \text{if } \mathcal{S}^\star = \bot_{\mathsf{sub}^\star} \\ \wp(\mathbb{M}) & \text{if } \mathcal{S}^\star = \top_{\mathsf{sub}^\star} \\ \bigcap_{\mathsf{s} \preceq_{\mathsf{sub}^\star} x \in \mathcal{S}^\star} \{\mathtt{m} \mid [\![\, \mathsf{s} \,]\!]\mathtt{m}, \mathtt{m}(x) \in \Sigma^*, [\![\, \mathsf{s} \,]\!]\mathtt{m} \curvearrowright \mathtt{m}(x)\} & \text{otherwise} \end{cases}$$

We define the abstract semantics of $\mathsf{Sub}^\star$. Let $\mathsf{extr}^\star : \textsc{se} \to \wp(\textsc{se})$ extend the function extr introduced in Sect. 5.1, extracting from a string expression s all the sub-expressions that syntactically appear at the top-level of s. For instance, $\mathsf{extr}^\star(\texttt{y + w + "ab"}) = \{y, w, w + \text{“}ab\text{”}, y + w, y + w + \text{“}ab\text{”}, \text{“}a\text{”}, \text{“}b\text{”}, \text{“}ab\text{”}\}$. Note that, for some $\mathsf{s} \in \textsc{se}$, we have that $\mathsf{s} \in \mathsf{extr}^\star(\mathsf{s})$. The abstract semantics of the assignment $[\![\, x = \mathsf{s} \,]\!^{\mathsf{sub}^\star}\mathcal{S}^\star$ is defined by the following steps. As before, we suppose that $\mathcal{S}^\star$ is not the bottom element, since in this case the bottom element is simply propagated skipping the above phases.

- [**remove**]: $\mathcal{S}_r^\star = \begin{cases} \mathcal{S}^\star \setminus \{\mathsf{s}' \preceq_{\mathsf{sub}^\star} z \mid x \text{ appears in } \mathsf{s}', z \in X_{str}\} & \text{if } x \in \mathsf{extr}^\star(\mathsf{s}) \\ \mathcal{S}^\star \setminus \{\mathsf{s}' \preceq_{\mathsf{sub}^\star} z \mid z = x \vee x \text{ appears in } \mathsf{s}'\} & \text{otherwise} \end{cases}$
- [**add**]: $\mathcal{S}_a^\star = \mathcal{S}_r^\star \cup \{\mathsf{s}' \preceq_{\mathsf{sub}^\star} x \mid \mathsf{s}' \in \mathsf{extr}^\star(\mathsf{s})\}$
- [**inter-asg**]: $\mathcal{S}_i^\star = \mathcal{S}_a^\star \cup \{x \preceq_{\mathsf{sub}^\star} y \mid \forall \mathsf{s}' \preceq_{\mathsf{sub}^\star} x \in \mathcal{S}_a^\star\ \exists \mathsf{s}' \preceq_{\mathsf{sub}^\star} y \in \mathcal{S}_a^\star\}$
- [**closure**]: $[\![\, x = \mathsf{s} \,]\!^{\mathsf{sub}^\star}\mathcal{S}^\star = \mathsf{Clos}(\mathcal{S}_i^\star)$

The [**remove**], [**add**] and [**closure**] phases are similar to those of the definition of $[\![\, \cdot \,]\!]^{\mathcal{A}}$. The intermediate phase [**inter-asg**] instead differs from the

```
1        x = "ab";
2        y = "a";
3        z = "b";
4        w = y + z;
```

Fig. 5: IMP example.

previous definitions and works as follows: if from the previous steps, any substring of x is also a substring of a string variable y, as checked in the [**inter-asg**] phase, we can safely assert that x is a substring of y and we can add that relation to $\mathcal{S}_a^\star$. It is worth noting that we can safely add the substring relation $x \preceq_{\mathsf{sub}^\star} y$, for some $y \in X_{str}$, just because we are performing an assignment $x = \mathsf{s}$. Indeed, we are overwriting the variable x with the assignment and in the [**add**] phase we surely add the relation $\mathsf{s} \preceq_{\mathsf{sub}^\star} x$; hence, if we found that any gathered substring relation concerning x (included $\mathsf{s} \preceq_{\mathsf{sub}^\star} x$) is tracked also for y, we can safely say that $x \preceq_{\mathsf{sub}^\star} y$. The abstract semantics of Boolean expressions is straightforward.

Similarly, we can also extend the abstract domains Len and Char to make them able to track relations between expressions and string variables, obtaining $\mathsf{Len}^\star$ and $\mathsf{Char}^\star$.

Capturing Other Implicit Substring Relations. In the previous section, we have presented the substring domain $\mathsf{Sub}^\star$ tracking the string expressions that are definitely substrings of a variable. As discussed in Sect. 1.1, we may lose any information about the tracked string value, leading to the loss of some implicit substring relations. Let us show the problem on $\mathsf{Sub}^\star$ considering the IMP fragment reported in Fig. 5. If we analyze the IMP fragment with $\mathsf{Sub}^\star$, the substring relations concerning the variable w are: $y \preceq_{\mathsf{sub}^\star} w, z \preceq_{\mathsf{sub}^\star} w, y + z \preceq_{\mathsf{sub}^\star} w$, *"a"* $\preceq_{\mathsf{sub}^\star} w$, *"b"* $\preceq_{\mathsf{sub}^\star} w$. Note that, $\mathsf{Sub}^\star$ cannot track that the variables `y` and `z` are exactly the strings *"a"* and *"b"*, respectively, and in turn it is not able to infer that `x` is a substring of `w` and viceversa, that is the variables `x` and `w` have the same string value.

In order to cope with this problem and to be able to track also these implicit relations, as discussed in Sect. 1.1, we rely on the reduced product combination of $\mathsf{Sub}^\star$ with a non-relational domain. In particular, we rely on the string constant propagation analysis, which tracks for each variable its constant value.[6] We model the constant propagation as a map, denoted by $\mathcal{CS}$, associating each string variable with the corresponding constant string value and if a variable is not mapped by the analysis it means that it is not constant. For instance, if we consider the fragment reported in Fig. 5, the constant propagation analysis, at line 4, returns the following map: $\{x \mapsto$ *"ab"*$, y \mapsto$ *"b"*$, z \mapsto$ *"b"*$, w \mapsto$ *"ab"*$\}$. At this point, the idea is to exploit the constant propagation analysis adding a new phase, that we call [**propagate**], at the end of the assignment abstract semantics $[\![\ x = \mathsf{s}\ ]\!]^{\mathsf{sub}^\star}\mathcal{S}^\star$ presented before. Let us denote by $\mathcal{S}_c^\star$ the abstract memory returned by the [**closure**] phase presented in the previous section and by

[6] Full details about how the constant propagation analysis works are reported in [33].

$\mathcal{CS}$ the constant propagation analysis holding at the assignment program point.

[propagate] :

$$[\![x = \mathtt{s}]\!]^{\mathsf{sub}^\star} \mathcal{S}^\star = \mathcal{S}_c^\star \cup \{x \preceq_{\mathsf{sub}^\star} y, y \preceq_{\mathsf{sub}^\star} x \mid \exists y \in X_{str}. \mathcal{CS}(x) = \mathcal{CS}(y)\}$$

Before returning the assignment result, the **[propagate]** phase checks if there exists a variable $y \in X_{str}$ such that y has the same constant value of the assigned variable x. If so, the substring relations $x \preceq_{\mathsf{sub}^\star} y$ and $y \preceq_{\mathsf{sub}^\star} x$ are added to the result. In this way, if we analyze again the fragment reported in Fig. 5, we exploit the constant propagation analysis in order to infer, at line 4, that $x \preceq_{\mathsf{sub}^\star} w$ and $w \preceq_{\mathsf{sub}^\star} x$, and in turn, we can state that the two variables are equal.

6 Experimental Results

RSub is a prototype intraprocedural static analyzer for the Go language implementing the $\mathsf{Sub}^\star$ relational abstract domain, available at `https://github.com/UniVE-SSV/go-lisa`. Indeed, from a precision point of view, $\mathsf{Sub}^\star$ subsumes the others string relational abstract domains presented in this paper. RSub is built as an extension of LiSA [21] (`https://github.com/UniVE-SSV/lisa`), a library for the development and the implementation of abstract interpretation-based static analyzers. We tested RSub over several representative string case studies, taken from real-world software and hand-crafted. In the following, we use two of these fragments to show the limits and strengths of $\mathsf{Sub}^\star$.

The rest of the section is structured as follows: in Sect. 6.1 we compare our analysis with prefix Pr, suffix Su, char inclusion Ci and bricks Br abstract domains [14], and with Tarsis [32]. Tarsis is a non-relational finite state automata-based abstract domain that abstracts string values into regular expressions. In Sect. 6.2 we show how to improve the precision of Tarsis by combining it with $\mathsf{Sub}^\star$. Finally, we evaluate the performance of $\mathsf{Sub}^\star$ through an experimental comparison between Tarsis and its combination with $\mathsf{Sub}^\star$, measuring the overhead added by $\mathsf{Sub}^\star$.

6.1 Case Studies

We consider two code fragments manipulating strings (cf. Fig. 6), ncon and rep (slight modification of the programs in Chap. 5 of [10] and [32], respectively). ncon overrides the variable `x` either with `x + "c"` or `y + "c"`, depending on whether the equality between `x` and `y` is satisfied or not. rep iteratively appends a string read from the user input and stored in `v` concatenated with the string `"\n"` to variable `r`. The value of the Boolean guards of both programs are supposed to be statically unknown, as well as the value of `v` in rep.

Let us consider the program ncon. Tab. 1 illustrates the results of the analysis at the end of programs ncon where the second column reports the abstract value of `x` at the end of each analysis, and third and forth columns are ✓ if the corresponding analysis proves that the `assert` conditions at lines 7-8 of ncon

```
1  if x == y {
2    x = x + "c"
3  } else {
4    x = y + "c"
5  }
6
7  assert (Contains(x, "c"))
8  assert (Contains(x, y))
```

(a) Program NCON

```
1  v = readStr()
2  r = "Elem: \n" + v + "\n"
3
4  for ? {
5    v = readStr()
6    r = r + v + "\n"
7  }
8
9  assert (Contains(r, "em"));
10 assert (Contains(r, v));
```

(b) Program REP

Fig. 6: Program samples used for domain comparison.

Domain	x abstract value	Assert 7	Assert 8
PR	ϵ (unknown)	✗	✗
SU	c	✓	✗
CI	$\{c\}, \{\Sigma\}$	✓	✗
BR	$\{\star\}(0,+\infty)$ (unknown)	✗	✗
TARSIS	$\{\star\}$c	✓	✗
RSUB	c $\preceq_{\mathsf{sub}^\star}$ x, $y \preceq_{\mathsf{sub}^\star}$ x	✓	✓

Table 1: Analysis results for NCON (where the symbol $\star$ denotes *"any string"*).

hold, or ✗ otherwise. The analyses based on PR and BR do not precisely verify all the assertions since they abstract x with their corresponding top value. Instead, CI, SU, and TARSIS verify the assertion at line 7 but not the one at line 8, since they cannot track any relation between the variables x and y. Finally, RSUB verifies all the assertions since it tracks that both string "c" and the variable y are substrings of x.

Consider now REP, which involves a fix-point computation. The analysis results at the end of the program REP are reported in Tab. 2, where the second column reports the abstract value of r at the end of each analysis, and third and forth columns are ✓ if the corresponding analysis proves that the assert conditions at lines 9-10 of REP hold, or ✗ otherwise. We must verify two assertions for this program, those at lines 9-10, that certainly hold. Note that the value (unknown) in Tab. 2 means that the corresponding analysis has returned the top abstract value. PR can verify the assertion at line 9 but not the ones at line 10, since it loses any information on the rest of the string, except for the common prefix, and it does not track the fact that variable v is undoubtedly contained in r. SU, CI, and BR analyses lose any information about the value of r, abstracting it with their corresponding top value. So, these analyses are unable to verify the assertions at lines 9-10. TARSIS abstracts the value of r as the regular expression reported in Tab. 2, correctly verifying the assertion at line 9 but not the one at line 10, being unable to track the relationship between the variables r and v. Instead, RSUB behaves as TARSIS as far as assertion at line 9 is concerned, since the string Elem:␣ is definitely a substring of r. Moreover, RSUB verifies the assertion at line 10, since it tracks that the variable v, independently from its abstract value, is a substring of the variable r.

Domain	r abstract value	Assert 9	Assert 10
Pr	$\texttt{Elem:}\sqcup$	✓	✗
Su	ϵ (unknown)	✗	✗
Ci	$\{\Sigma\},\{\Sigma\}$ (unknown)	✗	✗
Br	$\{\star\}(0,+\infty)$ (unknown)	✗	✗
Tarsis	$\texttt{Elem:}\sqcup\star\backslash\texttt{n}(\star\backslash\texttt{n})^*$	✓	✗
RSub	$\texttt{Elem}:\sqcup \preceq_{\mathsf{sub}^\star} r, v \preceq_{\mathsf{sub}^\star} r, r+v+\backslash\texttt{n} \preceq_{\mathsf{sub}^\star} r$ $v+\backslash\texttt{n} \preceq_{\mathsf{sub}^\star} r, \backslash\texttt{n} \preceq_{\mathsf{sub}^\star} r$	✓	✓

Table 2: Analysis results for REP (where the symbol $\star$ denotes *"any string"*).

```
//https://golang.org/src/strings
   /strings.go
func Count(s, src string) int {
  if len(src) == 0 {
    return len(s) + 1
  }
  n := 0
  for true {
    i := strings.Index(s, src)
    if i == -1 {
      return n
    }
    n++
    s = s[i+len(src):]
  }
}
```

```
import  "strings"

func Write(text, pt string) {
  if Contains(text, pt) {
    c := Count(text, pt)
    SetResult("result", c) •
  }
}
```

Fig. 7: Golang program example.

6.2 Improving Precision of Non-relational Abstract Domains

We evaluated the abstract domain $\mathsf{Sub}^\star$ as a standalone abstraction, w.r.t. to some state-of-art string abstractions, showing that more relations can be captured. As discussed in Sect. 1.1, $\mathsf{Sub}^\star$ may lose information about the content of string variables and its reduced product combination with a non-relational string abstract domain can be investigated in order to cope with this problem. Note that, the benefits of the combination of $\mathsf{Sub}^\star$ with a non-relational string abstract domain can be already seen with the code fragment reported in Sect. 6.1: reduced product combination between Pr and $\mathsf{Sub}^\star$ correctly verifies all the assertions contained in NCON and REP.

In this section, we show and discuss how to improve the precision of Tarsis [32] by combining it with $\mathsf{Sub}^\star$. In particular, we show that the abstract semantics of Tarsis can be refined, in terms of precision, when combined with $\mathsf{Sub}^\star$. We denote by Tarsis$^+$ the Cartesian product between Tarsis and $\mathsf{Sub}^\star$, a new string abstract domain tracking both the regular expressions approximating the strings values of each program variable (the non-relational information tracked by Tarsis) *and* the set of substring relations (the relational information tracked by $\mathsf{Sub}^\star$) holding at each program point. As far as integers are concerned, we abstract them with the interval abstract domain [15]. Let us consider the `Write` function reported in Fig. 7 that uses two string operations, i.e., `Contains` and `Count`, whose source code is reported on the left of Fig. 7. In particular, the `Write` function computes the number of occurrences of `pt` in `text` after checking the containment of `pt` in `text`.

We aim to infer the integer abstract value of `c` at the hotspot labeled with •. Note that the function parameters' values are statically unknown; for this reason, Tarsis approximates the values of `c` as the interval $[0, +\infty]$, introducing noise to the resulting interval. Indeed, the spurious value 0 corresponds to have no occurrences of `pt` in `text`, even if the program checks the condition `Contains(text, pt)`. This happens because Tarsis, when reaching the hotspot •, cannot track that `pt` is surely contained in `text`, causing the consequent loss of precision. Then, we analyzed `Write` with Tarsis$^+$. When the program point • is reached, Tarsis$^+$ captures that `pt` is a substring of `text`, capturing the substring relation `pt` $\preceq_{\mathsf{sub}^\star}$ `text`, since to reach the hotspot, the Boolean guard `Contains(text, pt)` must be traversed. Hence, the Tarsis analysis for the function `Count` can be improved, refining the interval resulting from Tarsis semantics, i.e., $[0, +\infty]$, with $[1, +\infty]$, since at least one occurrence of `pt` can be found in `text`. Note that the interval resulting from the Tarsis$^+$ analysis is the best possible interval abstraction that we can obtain (in this sense, the analysis is *complete* for the above function [8]). Similarly, also the Tarsis abstract semantics of other string operations can be refined. For instance, let us consider two string variables x and y and suppose that $x \preceq_{\mathsf{sub}^\star} y$. Given `Index(x, y)`, Tarsis would return the interval $[-1, maxLen(x)+1]$,[7] having no information about `x` and `y`. Instead, having the information $x \preceq_{\mathsf{sub}^\star} y$, Tarsis$^+$ can refine the aforementioned interval in $[0, maxLen(x) + 1]$. Another example is the case of `Replace(x, y, z)`: having the information about the containment of `y` in `x`, tracked by $\mathsf{Sub}^\star$, would lead to a must-replacement, that returns the input automaton where any occurrence of `y` is replaced with `z`, rather than a may-replacement, that returns the lub between the input automaton and the input automaton where any occurrence of `y` is replaced with `z`. [32].

6.3 Scalability of $\mathsf{Sub}^\star$

We conclude the experimental evaluation by discussing the performance of $\mathsf{Sub}^\star$. As also discussed in [29], the upper bounds domain of the domains presented in this paper offers an efficient implementation since it can be represented as a multi-valued map. For instance, the substring relations set $\{y \preceq_{\mathsf{sub}^\star} x, z \preceq_{\mathsf{sub}^\star} x, w \preceq_{\mathsf{sub}^\star} x\}$ can be represented as the map $x \mapsto \{y, z, w\}$. To assert the scalability of $\mathsf{Sub}^\star$, we crawled from GitHub the Go repositories dealing with the `strings` package, namely the Go package implementing popular functions manipulating strings (`https://golang.org/pkg/strings/`). From these repositories, we have selected the top *best matched* repositories (according to GitHub API), we have filtered only the Go program files, and we have selected the repositories with at least 10 Go programs. Finally, we ran our Go static analyzer with the so obtained programs both using Tarsis and Tarsis$^+$, recalling that the latter corresponds to the combination between Tarsis and $\mathsf{Sub}^\star$. At this point, we computed the overhead added by $\mathsf{Sub}^\star$ in Tarsis$^+$ w.r.t. Tarsis.

[7] $maxLen(x)$ returns the maximum length of the string recognized by the automaton abstracting x if it is finite, $+\infty$ otherwise.

Repository	Go files	Analyzed	LOCs	Tarsis$_{t(s)}$	Tarsis$^{+}_{t(s)}$	Overhead
dnnrly/abbreviate	14	10	1837	25.77	26.93	1.93%
reiver/go-stringcase	25	17	541	46.16	48.33	4.48%
gookit/goutil	55	29	1256	110.68	113.34	2.34%
schigh/str	12	5	126	19.70	20.58	4.27%
ozgio/strutil	22	11	218	39.01	41.91	6.91%
andy-zhangtao/gogather	26	12	531	48.80	51.51	5.26%
woanware/lookuper	173	41	5436	420.16	427.13	1.63%
RamenSea/StringCheese	24	11	833	50.41	52.03	3.11%
bcampbell/fuzzytime	10	6	745	19.05	20.03	4.89%
Total	**360**	**142**	**11523**	**779.74**	**801.79**	**2.75%**

Table 3: Tarsis and Tarsis^{+} performance results. From left to right: the GitHub repository name, the number of Go programs contained, the number of Go programs that the static analyzer has analyzed, the total number of lines of code analyzed, Tarsis and Tarsis^{+} execution times in seconds, and the overhead.

Tab. 3 summarizes the performance results for Tarsis and Tarsis^{+} for each repository. The difference between the number of Go analyzed programs and the total number of Go programs is due to Go features that are not currently supported by our static analyzer (e.g., channels, high-order functions, Go routines) and not due to analysis weaknesses. As stated by Tab. 3, the addition of $\mathsf{Sub}^{\star}$ to Tarsis does not considerably affect its analysis execution time, adding an overhead no greater than the 7% for each repository. The overall results confirm this, since the total overhead is below 3%, and almost 7% in the worst case.

7 Conclusion

In this paper, we introduced a general framework to generate new relational abstract domains starting from orders on string values. In particular, we introduced a new relational substring domain, $\mathsf{Sub}^{\star}$, showing its impact on the accuracy of the analysis with respect to state-of-the-art string abstractions, even when used as a *standalone* abstract domain. We have shown how to improve the precision of Tarsis, a finite-state automata-based string abstract domain, by combining it with $\mathsf{Sub}^{\star}$. Finally, we have provided experimental evidence that the addition of $\mathsf{Sub}^{\star}$ to Tarsis does not considerably affect the Tarsis performances.

As future works, we aim to formally investigate the precision increment gained by Tarsis^{+} w.r.t. Tarsis, measuring the *distance* [28] between their results. Furthermore, we aim to investigate the completeness property of Tarsis^{+} by applying the techniques in [8]. Finally, we aim to combine the relational abstract domains proposed in this paper with sophisticated state-of-the-art abstractions, e.g., the M-String abstract domain [11].

References

1. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janku, P.: Chain-free string constraints. In: Proc. of ATVA'19. Lecture Notes in Computer Science, vol. 11781, pp. 277–293. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_16

2. Amadini, R., Gange, G., Gauthier, F., Jordan, A., Schachte, P., Søndergaard, H., Stuckey, P.J., Zhang, C.: Reference abstract domains and applications to string analysis. Fundam. Inform. **158**(4), 297–326 (2018). https://doi.org/10.3233/FI-2018-1650
3. Amadini, R., Gange, G., Stuckey, P.J.: Dashed strings for string constraint solving. Artificial Intelligence **289** (2020). https://doi.org/https://doi.org/10.1016/j.artint.2020.103368
4. Amadini, R., Jordan, A., Gange, G., Gauthier, F., Schachte, P., Søndergaard, H., Stuckey, P.J., Zhang, C.: Combining string abstract domains for javascript analysis: An evaluation. In: Proc. of TACAS'17. pp. 41–57 (2017). https://doi.org/10.1007/978-3-662-54577-5_3
5. Arceri, V., Mastroeni, I.: An automata-based abstract semantics for string manipulation languages. In: Proc. of VPT'19. EPTCS, vol. 299, pp. 19–33 (2019). https://doi.org/10.4204/EPTCS.299.5
6. Arceri, V., Mastroeni, I.: Analyzing dynamic code: A sound abstract interpreter for *Evil* eval. ACM Trans. Priv. Secur. **24**(2), 10:1–10:38 (2021). https://doi.org/10.1145/3426470, `https://doi.org/10.1145/3426470`
7. Arceri, V., Mastroeni, I., Xu, S.: Static analysis for ecmascript string manipulation programs. Appl. Sci. **10**, 3525 (2020). https://doi.org/10.3390/app10103525
8. Arceri, V., Olliaro, M., Cortesi, A., Mastroeni, I.: Completeness of string analysis for dynamic languages. Information and Computation p. 104791 (2021). https://doi.org/https://doi.org/10.1016/j.ic.2021.104791
9. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1-2), 3–21 (2008). https://doi.org/10.1016/j.scico.2007.08.001
10. Bultan, T., Yu, F., Alkhalaf, M., Aydin, A.: String Analysis for Software Verification and Security. Springer (2017). https://doi.org/10.1007/978-3-319-68670-7
11. Cortesi, A., Lauko, H., Olliaro, M., Rockai, P.: String abstraction for model checking of C programs. In: Proc. of SPIN'19. pp. 74–93 (2019). https://doi.org/10.1007/978-3-030-30923-7_5
12. Cortesi, A., Olliaro, M.: M-string segmentation: A refined abstract domain for string analysis in C programs. In: Proc. of TASE'18. pp. 1–8 (2018). https://doi.org/10.1109/TASE.2018.00009
13. Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: Proc. of ICFEM'11. Lecture Notes in Computer Science, vol. 6991, pp. 505–521. Springer (2011). https://doi.org/10.1007/978-3-642-24559-6_34
14. Costantini, G., Ferrara, P., Cortesi, A.: A suite of abstract domains for static analysis of string values. Softw., Pract. Exper. **45**(2), 245–287 (2015). https://doi.org/10.1002/spe.2218
15. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL'77. pp. 238–252 (1977). https://doi.org/10.1145/512950.512973
16. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979. pp. 269–282. ACM Press (1979). https://doi.org/10.1145/567752.567778
17. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of POPL'79. pp. 269–282 (1979). https://doi.org/10.1145/567752.567778

18. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Log. Program. **13**(2&3), 103–179 (1992). https://doi.org/10.1016/0743-1066(92)90030-7
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. of POPL'78. pp. 84–96 (1978). https://doi.org/10.1145/512760.512770
20. Ferrara, P., Logozzo, F., Fähndrich, M.: Safer unsafe code for .net. In: Proc. of OOPSLA'08. pp. 329–346. ACM (2008). https://doi.org/10.1145/1449764.1449791
21. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: experiencing lisa. In: Do, L.N.Q., Urban, C. (eds.) SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021. pp. 1–6. ACM (2021). https://doi.org/10.1145/3460946.3464316
22. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: An abstract domain of uninterpreted functions. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9583, pp. 85–103. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_4
23. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Proc. of PLDI'06. pp. 376–386 (2006). https://doi.org/10.1145/1133981.1134026
24. Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. In: Proc. of NFM'17. pp. 212–229 (2017). https://doi.org/10.1007/978-3-319-57288-8_15
25. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javascript. In: Proc. of SAS'09. pp. 238–255 (2009). https://doi.org/10.1007/978-3-642-03237-0_17
26. Kashyap, V., Dewey, K., Kuefner, E.A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., Hardekopf, B.: JSAI: a static analysis platform for javascript. In: Proc. of FSE-22. pp. 121–132 (2014). https://doi.org/10.1145/2635868.2635904
27. Lee, H., Won, S., Jin, J., Cho, J., Ryu, S.: SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In: Proc. of FOOL'12 (2012)
28. Logozzo, F.: Towards a quantitative estimation of abstract interpretations. In: Workshop on Quantitative Analysis of Software. Microsoft (June 2009), `https://www.microsoft.com/en-us/research/publication/towards-a-quantitative-estimation-of-abstract-interpretations/`
29. Logozzo, F., Fähndrich, M.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. Sci. Comput. Program. **75**(9), 796–807 (2010). https://doi.org/10.1016/j.scico.2009.04.004
30. Madsen, M., Andreasen, E.: String analysis for dynamic field access. In: Proc. of CC'14. pp. 197–217 (2014). https://doi.org/10.1007/978-3-642-54807-9_12
31. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
32. Negrini, L., Arceri, V., Ferrara, P., Cortesi, A.: Twinning automata and regular expressions for string static analysis. In: Proc. of VMCAI'21. Lecture Notes in Computer Science, vol. 12597, pp. 267–290. Springer (2021). https://doi.org/10.1007/978-3-030-67067-2_13
33. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6
34. Park, C., Im, H., Ryu, S.: Precise and scalable static analysis of jquery using a regular expression domain. In: Proc. of DLS'16. pp. 25–36 (2016). https://doi.org/10.1145/2989225.2989228

35. Seidl, H., Wilhelm, R., Hack, S.: Compiler Design - Analysis and Transformation. Springer (2012)
36. Wang, H., Chen, S., Yu, F., Jiang, J.R.: A symbolic model checking approach to the analysis of string and length constraints. In: Proc. of ASE'18. pp. 623–633. ACM (2018). https://doi.org/10.1145/3238147.3238189
37. Wilhelm, R., Sagiv, S., Reps, T.W.: Shape analysis. In: Proc. of CC'00. pp. 1–17 (2000). https://doi.org/10.1007/3-540-46423-9_1
38. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods Syst. Des. **44**(1), 44–70 (2014). https://doi.org/10.1007/s10703-013-0189-1
39. Yu, F., Bultan, T., Hardekopf, B.: String abstractions for string verification. In: Proc. of SPIN'11. pp. 20–37 (2011). https://doi.org/10.1007/978-3-642-22306-8_3