Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: xxx/xxxx

Tarsis: an effective automata-based abstract domain for string analysis

Luca Negrini[2] | **Vincenzo Arceri*[1]** | **Agostino Cortesi[2]** | **Pietro Ferrara[2]**

[1]Department of Mathematical, Physical and Computer Sciences, University of Parma, Italy
[2]Department of Environmental Sciences, Informatics and Statistics, Ca' Foscari University of Venice, Italy

Correspondence
*Vincenzo Arceri
Email: vincenzo.arceri@unipr.it

Present address
Parco Area delle Scienze, 53/A, 43124, Parma, Italy

In this paper we introduce TARSIS, a new abstract domain based on the abstract interpretation theory that approximates string values through finite state automata. The main novelty of TARSIS is that it works over an alphabet of strings instead of single characters. On the one hand, such an approach requires a more complex and refined definition of the lattice operators and of the abstract semantics of string operators. On the other hand, it is in position to obtain strictly more precise results than state-of-the-art approaches. We compare TARSIS both with simpler domains and with the standard automata model, targeting case studies containing standard yet challenging string manipulations. The performance gain w.r.t. the standard automata model is also assessed, measuring the speed-up gained by TARSIS. Experiments confirm that TARSIS can obtain precise results without incurring in excessive computational costs.

KEYWORDS
String analysis, Static analysis, Abstract interpretation.

1 | INTRODUCTION

Nowadays, string values play a key role in any modern programming language, since they are adopted for a variety of purposes and tasks. For instance, they allow to dynamically access object properties, to hide the program code by using string-to-code statements and reflection, or to manipulate data-interchange formats, such as JSON, just to name a few. In this context, the correctness of string manipulations is therefore crucial. Sound static analysis[1,2] has been widely applied to prove the correctness of programs (e.g., the absence of bugs). Recently, a relevant effort was spent towards the static approximation of string values in different contexts, such as SQL queries programmatically built by code[3], reflection[4,5], string-to-code statement analysis[6], and injection vulnerabilities[7,8].

Despite the great effort spent in reasoning about strings, static analysis often failed to manage programs that heavily manipulate strings, mainly due to the inaccuracy of the results and/or the prohibitive amount of resources (time, space) required to retrieve useful information on strings. On the one hand, finite height string abstractions[9] are computable in a reasonable time, but precision is suddenly lost when using advanced string manipulations. On the other hand, more sophisticated abstractions (e.g., the ones reported in[10,11]) compute precise results but they require a huge, and sometimes unrealistic, computational cost, making the analysis of real code intractable. A good representation of the latter abstractions is the finite state automata domain[10]. Over-approximating strings into finite state automata has shown to increase string analysis accuracy in many scenarios, but it does not scale up to real world programs dealing with statically unknown inputs and long text manipulations.

In this paper we introduce TARSIS, a new abstract domain for string values based on finite state automata (FSA). Standard FSA has been shown to provide precise abstractions of string values when all the components of such strings are known, but with high computational cost. Instead of considering finite automata built over the classical alphabet of single characters, TARSIS considers automata built over an alphabet of strings. The alphabet comprises a special value to represent statically unknown strings. This avoids the creation of self-loops with any possible character as input, which otherwise would significantly degrade performance. We define the TARSIS's abstract semantics on all string operations reported in[10], that we use as reference, either defined directly on the automaton or on its equivalent regular expression.


```go
1  func Count(s, substr string) int {
2    if len(substr) == 0 {
3      return len(s) + 1
4    }
5    n := 0
6    for true {
7      i := strings.Index(s, substr)
8      if i == -1 {
9        return n
10     }
11     n++
12     s = s[i+len(substr):]
13   }
14 }
```

FIGURE 1 The `strings.Count` function of the Go API

TARSIS has been implemented in GoLiSA[12], a static analyzer for Go based on LiSA[13,12,14]. By comparing TARSIS with other cutting-edge domains for string analysis, results show that (i) when applied to simple code that causes a precision loss in simpler domains, TARSIS correctly approximates string values within a comparable execution time, (ii) on code that makes the standard automata domain unusable due to the complexity of the analysis, TARSIS is in position to perform in a limited amount of time, making it a viable domain for complex and real codebases, and (iii) TARSIS is able to precisely abstract complex string operations that have not been addressed by state-of-the-art domains.

This paper is a revised and extended version of[15]. Specifically, we completed the previous version of the paper by covering all of the string operations considered in[10], with the addition of the string equality operator. For all supported operations (both newly added and already formalized in[15]), we also reported proofs of soundness and completeness (or incompleteness). Moreover, we repeated the original experimental evaluation using GoLiSA[12] instead of the prototypical analyzer used in the original paper. Finally, to clearly assess the performance gain of TARSIS w.r.t. the finite-state automata abstract domain introduced in[10], we extended our evaluation to deeply compare execution times of the two domains.

The rest of the paper is structured as follows. Sect. 2 introduces a motivating example. Sect. 3 defines the mathematical notation used throughout the paper. Sect. 4 formalizes TARSIS and its abstract semantics. Sect. 5 compares TARSIS with other domains. Sect. 6 discusses most related works, while Sect. 7 concludes. Appendix A reports soundness and completeness (or incompleteness) proofs for TARSIS abstract semantics.

2 MOTIVATING EXAMPLE

Consider the code of Fig. 1 that counts the occurrences of string `substr` into string `s`. This code is (a simplification of) the Go API function `strings.Count` (see https://cs.opensource.google/go/go/+/refs/tags/go1.20.1:src/strings/strings.go). Proving properties about the value of `n` at line 9, is particularly challenging, since it requires to correctly model a set of string operations (namely `len`, `Index`, and substring) and their interaction. State-of-the-art string analyses fail to precisely model most of such operations, since their abstraction of string values is not rigorous enough to deal with them. This loss of precision usually leads to failure in proving string-based properties (also on non-string values) in real-world software, such as the numerical bounds of the value returned by `Count` when applied to a string.

The goal of this paper is to provide an abstract interpretation-based static analysis, in order to deal with complex and nested string manipulations similar to the one reported in Fig. 1. As we will discuss in Sect. 5, TARSIS models (among the others) all string operations used in `Count`, and it is precise enough to infer, given the abstractions of `s` and `substr`, the precise range of values that `n` might have when the function returns.

3 PRELIMINARIES

Mathematical notation. Given a set S, S^* is the set of all finite sequences of elements of S. If $s = s_0 \dots s_n \in S^*$, s_i is the i-th element of s, $|s| = n + 1$ is its length, and $s[x/y]$ is the sequence obtained by replacing all occurrences of x in s with y. When s' is a subsequence of s, we write $s' \curvearrowright_{\mathsf{s}} s$. Given $s \in S^*$ and $i, j \in \mathbb{N} \,.\, 0 \leq i \leq j \leq |s|$, we denote the subsequence $s_i s_{i+1} \dots s_{j-1}$ by $s[i : j]$, with $s[i :]$ denoting the subsequence $s_i s_{i+1} \dots s_n$. We denote by s^n, with $n \geq 0$ the n-times repetition of the string s. Given

$\mathsf{a} \in \text{AE} ::= x \in \text{ID} \mid n \in \mathbb{Z} \mid \mathsf{a} + \mathsf{a} \mid \mathsf{a} - \mathsf{a} \mid \mathsf{a} \star \mathsf{a} \mid \mathsf{a} / \mathsf{a}$
$\quad \mid \texttt{length}(\mathsf{s}) \mid \texttt{indexOf}(\mathsf{s},\mathsf{s})$
$\mathsf{b} \in \text{BE} ::= x \in \text{ID} \mid \texttt{true} \mid \texttt{false} \mid \mathsf{b}\ \texttt{\&\&}\ \mathsf{b} \mid \mathsf{b}\ \texttt{||}\ \mathsf{b} \mid \texttt{!}\ \mathsf{b} \mid \mathsf{e} < \mathsf{e}$
$\quad \mid \mathsf{e} == \mathsf{e} \mid \texttt{contains}(\mathsf{s}_1,\mathsf{s}_2) \mid \texttt{startsWith}(\mathsf{s}_1,\mathsf{s}_2) \mid \texttt{endsWith}(\mathsf{s}_1,\mathsf{s}_2)$
$\mathsf{s} \in \text{SE} ::= x \in \text{ID} \mid \texttt{"}\sigma\texttt{"} \mid \texttt{substr}(\mathsf{s},\mathsf{a},\mathsf{a}) \mid \texttt{charAt}(\mathsf{s},\mathsf{a})$
$\quad \mid \texttt{repeat}(\mathsf{s},\mathsf{a}) \mid \texttt{concat}(\mathsf{s},\mathsf{s}) \mid \texttt{replace}(\mathsf{s},\mathsf{s},\mathsf{s})$
$\quad \mid \texttt{trim}(\mathsf{s}) \mid \texttt{trimLeft}(\mathsf{s}) \mid \texttt{trimRight}(\mathsf{s}) \qquad (\sigma \in \Sigma^*)$
$\mathsf{e} \in \text{E} ::= \mathsf{a} \mid \mathsf{b} \mid \mathsf{s}$
$\mathsf{st} \in \text{STMT} ::= \mathsf{st}\ ;\ \mathsf{st} \mid \texttt{skip} \mid x = \mathsf{e} \mid \texttt{if}\ (\mathsf{b})\ \{\ \mathsf{st}\ \}\ \text{else}\ \{\ \mathsf{st}\ \} \mid \text{while}\ (\mathsf{b})\ \{\ \mathsf{st}\ \}$
$\mathsf{P} \in \text{IMP} ::= \mathsf{st}\ ;$

FIGURE 2 IMP syntax

two sets S and T, $\wp(S)$ is the powerset of S, $S \smallsetminus T$ is the set difference, $S \subset T$ is the strict inclusion relation between S and T, $S \subseteq T$ is the inclusion relation between S and T, $S \times T$ is the Cartesian product between S and T, and $S \cdot T$ is the concatenation of S and T, i.e., $S \cdot T = \{\ s \cdot t \mid s \in S, t, \in T\ \}$. Given a set S and $n \in \mathbb{N}$, S^n is recursively defined as $S^0 \triangleq \{\epsilon\}$, and $S^{n>0} \triangleq S \cdot S^{n-1}$.

Ordered structures. A set L with a partial ordering relation $\leq \subseteq L \times L$ is a poset, denoted by $\langle L, \leq\rangle$. A poset $\langle L, \leq, \vee, \wedge\rangle$, where $\vee$ and $\wedge$ are respectively the least upper bound (lub) and greatest lower bound (glb) operators of L, is a lattice if $\forall x, y \in L\,.\,x \vee y$ and $x \wedge y$ belong to L. It is also complete if $\forall X \subseteq L$ we have that $\bigvee X, \bigwedge X \in L$. A complete lattice L, with ordering $\leq$, lub $\vee$, glb $\wedge$, top element $\top$, and bottom element $\bot$ is denoted by $\langle L, \leq, \vee, \wedge, \top, \bot\rangle$.

Abstract interpretation. Abstract interpretation [1,2] is a theoretical framework for sound reasoning about semantic properties of a program, establishing a correspondence between the concrete semantics of a program and an approximation of it, called abstract semantics. Let C and A be complete lattices, a pair of monotone functions $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ forms a *Galois Connection* (GC) between C and A if $\forall x \in C, \forall y \in A : \alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. We denote a GC as $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$. Given $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, a concrete function $f : C \rightarrow C$ is, in general, not computable. Hence, a function $f^\sharp : A \rightarrow A$ that must *correctly* approximate the function f is needed. If so, we say that the function $f^\sharp$ is *sound*. Given $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$ and a concrete function $f : C \rightarrow C$, an abstract function $f^\sharp : A \rightarrow A$ is sound w.r.t. f if $\forall c \in C\,.\,\alpha(f(c)) \leq_A f^\sharp(\alpha(c))$, or equivalently $\forall a \in A\,.\,f(\gamma(a)) \leq_C \gamma(f^\sharp(a))$. Completeness [16] can be obtained by enforcing the equality of the soundness conditions. Doing so, we obtain two notion of completeness. Given $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, a concrete function $f : C \rightarrow C$ and an abstract function $f^\sharp : A \rightarrow A$, $f^\sharp$ is *backward complete* w.r.t. f if $\forall c \in C.\ \alpha(f(c)) = f^\sharp(\alpha(c))$, and it is *forward complete* w.r.t. f if $\forall a \in A\,.\,f(\gamma(a)) = \gamma(f^\sharp(a))$.

Finite state automata and regular expression notation. We follow the notation reported in [10] for introducing finite state automata. A finite state automaton (FSA) is a tuple $\mathtt{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite alphabet of symbols, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is the set of final states. If $\delta : Q \times \Sigma \rightarrow Q$ is a function then $\mathtt{A}$ is called deterministic finite state automaton. The set of all the FSAs is FA. If $\mathscr{L} \subseteq \Sigma^*$ is recognized by a FSA, we say that $\mathscr{L}$ is a regular language. Given $\mathtt{A} \in$ FA, $\mathscr{L}(\mathtt{A})$ is the language accepted by $\mathtt{A}$. From the Myhill-Nerode theorem, for each regular language there uniquely exists a minimum FSA (w.r.t. the number of states) recognizing the language. Given a regular language $\mathscr{L}$, $\mathsf{Min}(\mathtt{A})$ is the minimum FSA $\mathtt{A}$ s.t. $\mathscr{L} = \mathscr{L}(\mathtt{A})$. Abusing notation, given a regular language $\mathscr{L}$, $\mathsf{Min}(\mathscr{L})$ is the minimal FSA recognizing $\mathscr{L}$. Given $\mathtt{A}$, we denote by $\mathsf{Kleene}(\mathtt{A})$ the automaton recognizing the Kleene closure of $\mathscr{L}(\mathtt{A})$.

We denote as $\mathsf{paths}(\mathtt{A}) \in \wp(\delta^*)$ the set of sequences of transitions corresponding to all the possible paths from the initial state q_0 to a final state $q_n \in F$. When $\mathtt{A}$ is cycle-free, the set $\mathsf{paths}(\mathtt{A})$ is finite and computable. Given $\pi \in \mathsf{paths}(\mathtt{A})$, $|\pi|$ is its length, meaning the sum of the lengths of the symbols that appear on the transitions composing the path. Furthermore, $|\mathsf{minPath}(\mathtt{A})| \in \mathbb{N}$ denotes the (unique) length of a minimum path. If $\mathtt{A}$ is a cycle-free automaton, $|\mathsf{maxPath}(\mathtt{A})| \in \mathbb{N}$ denotes the (unique) length of a maximum path. Given $\pi = t_0 \dots t_n \in \mathsf{paths}(\mathtt{A})$, σ_{π_i} is the symbol read by the transition t_i, $i \in [0, n]$, and $\sigma_\pi = \sigma_{\pi_0} \dots \sigma_{\pi_n}$ is the string recognized by such path. Predicate $\mathsf{cyclic}(\mathtt{A})$ holds if and only if the given automaton contains a cycle. Throughout the paper, it could be more convenient to refer to a FSA by its regular expression (regex for short), being equivalent. Given two regexes $\mathtt{r}_1$ and $\mathtt{r}_2$, $\mathtt{r}_1 \mid\mid \mathtt{r}_2$ is the disjunction between $\mathtt{r}_1$ and $\mathtt{r}_2$, $\mathtt{r}_1\mathtt{r}_2$ is the concatenation of $\mathtt{r}_1$ with $\mathtt{r}_2$, $(\mathtt{r}_1)^*$ is the Kleene-closure of $\mathtt{r}_1$.

The finite state automata abstract domain. Here, we report the necessary notions about the finite state automata abstract domain presented in [10], over-approximating string properties as the minimum deterministic finite state automaton recognizing

$$
\begin{aligned}
&[\![\ \texttt{substr}(\mathsf{s},\mathsf{a},\mathsf{a}')\ ]\!]\mathbb{m} = \sigma_i \dots \sigma_j \qquad \text{if } i \leq j < |\sigma|, i = [\![\ \mathsf{a}\ ]\!]\mathbb{m}, j = [\![\ \mathsf{a}'\ ]\!]\mathbb{m} \\
&[\![\ \texttt{charAt}(\mathsf{s},\mathsf{a})\ ]\!]\mathbb{m} = [\![\ \texttt{substr}(\mathsf{s},\mathsf{a},\mathsf{a}+1)\ ]\!]\mathbb{m} \\
&[\![\ \texttt{indexOf}(\mathsf{s},\mathsf{s}')\ ]\!]\mathbb{m} = \begin{cases} \min\{\ i \mid \sigma_i \dots \sigma_j = \sigma'\ \} & \text{if } \exists i,j \in \mathbb{N}\,.\, \sigma_i \dots \sigma_j = \sigma' \\ -1 & \text{otherwise} \end{cases} \\
&[\![\ \texttt{replace}(\mathsf{s},\mathsf{s}',\mathsf{s}'')\ ]\!]\mathbb{m} = \begin{cases} \sigma[\sigma'/\sigma''] & \text{if } \sigma' \curvearrowright_{\mathsf{s}} \sigma \\ \sigma & \text{otherwise} \end{cases} \\
&[\![\ \texttt{trimLeft}(\mathsf{s})\ ]\!]\mathbb{m} = \sigma' \text{ where } \sigma = \psi\sigma', \psi = \max\{\ \psi' \in \{\text{␣}\}^* \mid \sigma = \psi'\sigma''\ \} \\
&[\![\ \texttt{trimRight}(\mathsf{s})\ ]\!]\mathbb{m} = \sigma' \text{ where } \sigma = \sigma'\phi, \phi = \max\{\ \phi' \in \{\text{␣}\}^* \mid \sigma = \sigma''\phi'\ \} \\
&[\![\ \texttt{trim}(\mathsf{s})\ ]\!]\mathbb{m} = [\![\ \texttt{trimLeft}(\texttt{trimRight}(\mathsf{s}))\ ]\!]\mathbb{m} \\
&[\![\ \texttt{concat}(\mathsf{s},\mathsf{s}')\ ]\!]\mathbb{m} = \sigma \cdot \sigma' \qquad [\![\ \texttt{length}(\mathsf{s})\ ]\!]\mathbb{m} = |\sigma| \qquad [\![\ \mathsf{s} == \mathsf{s}'\ ]\!]\mathbb{m} = \sigma == \sigma' \\
&[\![\ \texttt{contains}(\mathsf{s},\mathsf{s}')\ ]\!]\mathbb{m} = \begin{cases} \texttt{true} & \text{if } \exists i,j \in \mathbb{N}\,.\, \sigma_i \dots \sigma_j = \sigma' \\ \texttt{false} & \text{otherwise} \end{cases} \\
&[\![\ \texttt{repeat}(\mathsf{s},\mathsf{a})\ ]\!]\mathbb{m} = \sigma^n \qquad \text{if } n = [\![\ \mathsf{a}\ ]\!]\mathbb{m}, n \geq 0 \\
&[\![\ \texttt{startsWith}(\mathsf{s},\mathsf{s}')\ ]\!]\mathbb{m} = \begin{cases} \texttt{true} & \text{if } \exists \phi \in \Sigma^*\,.\, \sigma = \sigma'\phi \\ \texttt{false} & \text{otherwise} \end{cases} \\
&[\![\ \texttt{endsWith}(\mathsf{s},\mathsf{s}')\ ]\!]\mathbb{m} = \begin{cases} \texttt{true} & \text{if } \exists \phi \in \Sigma^*\,.\, \sigma = \phi\sigma' \\ \texttt{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

FIGURE 3 Concrete semantics of IMP string expressions, where $\sigma = [\![\ \mathsf{s}\ ]\!]\mathbb{m}, \sigma' = [\![\ \mathsf{s}'\ ]\!]\mathbb{m}, \sigma'' = [\![\ \mathsf{s}''\ ]\!]\mathbb{m}$

them. Given an alphabet Σ, the finite state automata domain is defined as $\langle \text{FA}_{/\equiv}, \sqsubseteq_{\text{FA}}, \sqcup_{\text{FA}}, \sqcap_{\text{FA}}, \mathsf{Min}(\varnothing), \mathsf{Min}(\Sigma^*)\rangle$, where $\text{FA}_{/\equiv}$ is the quotient set of FA w.r.t. the equivalence relation induced by language equality, $\sqsubseteq_{\text{FA}}$ is the partial order induced by language inclusion, $\sqcup_{\text{FA}}$ and $\sqcap_{\text{FA}}$ are the lub and the glb, respectively. The minimum is $\mathsf{Min}(\varnothing)$, that is, the automaton recognizing the empty language, and the maximum is $\mathsf{Min}(\Sigma^*)$, that is, the automaton recognizing any possible string over Σ. We abuse notation by representing equivalence classes in $\text{FA}_{/\equiv}$ by one of its automaton (usually the minimum), i.e., when we write $\mathtt{A} \in \text{FA}_{/\equiv}$ we mean $[\mathtt{A}]_{\equiv}$. Since $\text{FA}_{/\equiv}$ does not satisfy the Ascending Chain Condition (ACC), i.e., it contains infinite ascending chains, it is equipped with the parametric widening ∇^n_{FA}. The latter is defined in terms of a state equivalence relation merging states that recognize the same language, up to a fixed length $n \in \mathbb{N}$, a parameter used for tuning the widening precision[17,18]. For instance, let us consider the automata $\mathtt{A}, \mathtt{A}' \in \text{FA}_{/\equiv}$ recognizing the languages $\mathscr{L} = \{\epsilon, a\}$ and $\mathscr{L}' = \{\epsilon, a, aa\}$, respectively. The result of the application of the widening ∇^n_{FA}, with $n = 1$, is $\mathtt{A}\ \nabla^n_{\text{FA}}\ \mathtt{A}' = \mathtt{A}''$ s.t. $\mathscr{L}(\mathtt{A}'') = \{\ a^n \mid n \in \mathbb{N}\ \}$.

Core language and semantics. We introduce a core language IMP, whose syntax is reported in Fig. 2. Such language, besides supporting arithmetic expressions (AE) and Boolean expressions (BE), also supports all of the string expressions (SE) discussed in[10], that we use as reference. Primitives values are $\text{VAL} = \mathbb{Z} \cup \Sigma^* \cup \{\texttt{true}, \texttt{false}\}$, namely integers, strings and booleans. Programs states $\mathbb{M} : \text{ID} \to \text{VAL}$ map identifiers to primitives values, ranged over the meta-variable $\mathbb{m}$. The concrete semantics of IMP statements is captured by the function $[\![\ \mathsf{st}\ ]\!] : \mathbb{M} \to \mathbb{M}$. The semantics is defined in a standard way and for this reason has been omitted. Such semantics relies on the one of expressions, that we capture, abusing notation, as $[\![\ \mathsf{e}\ ]\!] : \mathbb{M} \to \text{VAL}$. We define the part concerning strings in Fig. 3.

4 | THE TARSIS ABSTRACT DOMAIN

In this section, we recast the original finite state abstract domain working over an alphabet of characters Σ, reported in Sect. 3, to an augmented abstract domain based on finite state automata over an alphabet of strings.

4.1 | Abstract domain and widening

The key idea of TARSIS is to adopt the same FSA-based domain, changing the alphabet on which automata are defined to a set of strings, namely Σ^*. The main concern is that Σ^* is infinite and it would not permit us to adopt the FSA model, that requires the alphabet to be finite. Thus, in order to solve this problem, we make this abstract domain *parametric* to the program we aim to analyze and in particular to its strings. Given an IMP program P, we denote by Σ^*_{P} any substring of strings appearing in P (the set Σ^*_{P} can be easily computed collecting the constant strings in P by visiting its abstract syntax tree and then computing their substrings), *delimiting* the space of string properties we aim to check only on P.

At this point, we can instantiate the automata-based framework proposed in[10] with the new alphabet as

$$\langle \mathcal{T}\mathrm{FA}_{/\equiv}, \sqsubseteq_{\mathcal{T}}, \sqcup_{\mathcal{T}}, \sqcap_{\mathcal{T}}, \mathsf{Min}(\varnothing), \mathsf{Min}(\mathbb{A}^*_{\mathsf{P}})\rangle$$

The alphabet on which finite state automata are defined is $\mathbb{A}_{\mathsf{P}} \triangleq \Sigma^*_{\mathsf{P}} \cup \{\mathsf{T}\}$, where T is a special symbol that we intend as *"any possible string"*. Let $\mathcal{T}\mathrm{FA}$ be the set of any deterministic FSA over the alphabet $\mathbb{A}_{\mathsf{P}}$. Since we can have more automata recognizing a language, $\mathcal{T}\mathrm{FA}_{/\equiv}$ is the quotient set of $\mathcal{T}\mathrm{FA}$ w.r.t. the equivalence relation induced by language equality, that is, the elements of domain are equivalence classes. For simplicity, when we write $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$, we intend the equivalence class of $\mathtt{A}$. $\sqsubseteq_{\mathcal{T}}$ is the partial order induced by language inclusion, $\sqcup_{\mathcal{T}}$ and $\sqcap_{\mathcal{T}}$ are the lub and the glb over elements of $\mathcal{T}\mathrm{FA}_{/\equiv}$, computing the equivalence class of the union and the intersection of the two automata representing the corresponding classes, respectively. The bottom element is $\mathsf{Min}(\varnothing)$, corresponding to the automaton recognizing the empty language, and the maximum is $\mathsf{Min}(\mathbb{A}^*_{\mathsf{P}})$, namely the automaton recognizing any string over $\mathbb{A}_{\mathsf{P}}$.

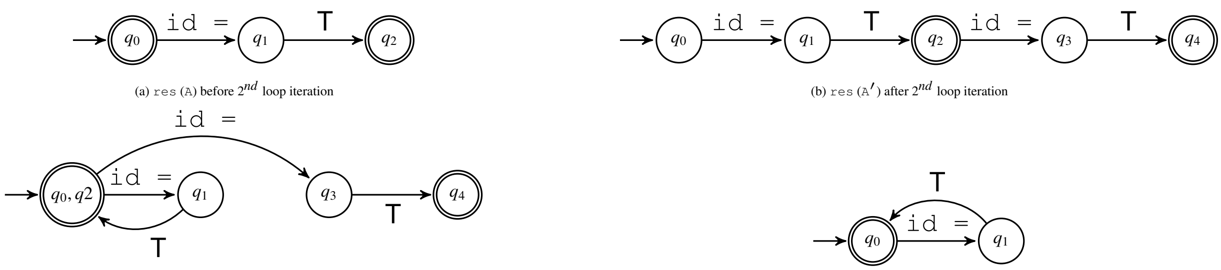
Similarly to the standard FSA domain $\mathrm{FA}_{/\equiv}$, also $\mathcal{T}\mathrm{FA}_{/\equiv}$ is not a complete lattice and, consequently, it does not form a Galois Connection with the string concrete domain $\wp(\Sigma^*)$. This comes from the non-existence, in general, of the best abstraction of a string set in $\mathcal{T}\mathrm{FA}_{/\equiv}$ (e.g., a context-free language has no best abstract element in $\mathcal{T}\mathrm{FA}_{/\equiv}$ approximating it). Nevertheless, this is not a concern since weaker forms of abstract interpretation are still possible[19] still guaranteeing soundness relations between concrete and abstract elements (e.g., polyhedra[20,21]). In particular, we can still ensure soundness comparing the concretizations of our abstract elements (cf. Sect. 8 of[19]). Hence, we define the concretization function $\gamma_{\mathcal{T}} : \mathcal{T}\mathrm{FA}_{/\equiv} \to \wp(\Sigma^*)$ as $\gamma_{\mathcal{T}}(\mathtt{A}) \triangleq \bigcup_{\sigma \in \mathscr{L}(\mathtt{A})} \mathsf{Flat}(\sigma)$, where Flat converts a string in $\mathbb{A}^*_{\mathsf{P}}$ into a set of strings in Σ^*. For instance, $\mathsf{Flat}(a\ \mathsf{TT}\ bb\ c) = \{\ a\sigma bbc \mid \sigma \in \Sigma^*\ \}$. Note that, the language of strings recognized by $\mathtt{A}$ corresponds to the concretization function reported above, namely $\mathscr{L}(\mathtt{A}) = \gamma_{\mathcal{T}}(\mathtt{A})$.

Widening. Similarly to the standard automata domain $\mathrm{FA}_{/\equiv}$, also $\mathcal{T}\mathrm{FA}_{/\equiv}$ does not satisfy ACC, meaning that fixpoint computations over $\mathcal{T}\mathrm{FA}_{/\equiv}$ may not converge in a finite time. Hence, we need to equip $\mathcal{T}\mathrm{FA}_{/\equiv}$ with a widening operator to ensure the convergence of the analysis. We define the widening $\nabla^n_{\mathcal{T}} : \mathcal{T}\mathrm{FA}_{/\equiv} \times \mathcal{T}\mathrm{FA}_{/\equiv} \to \mathcal{T}\mathrm{FA}_{/\equiv}$, parametric in $n \in \mathbb{N}$, taking two automata as input and returning an over-approximation of the least upper bounds between them, as required by widening definition. We rely on the standard automata widening reported in Sect. 3, that, informally speaking, can be seen as a *subset construction* algorithm[22] up to languages of strings of length n.

To explain the widening $\nabla^n_{\mathcal{T}}$, consider the following function manipulating strings; for the sake of readability, in the program examples presented in this paper the plus operation between strings corresponds to the string concatenation:

```
function f(v) {
    res = "";
    while (?)
        res = res + "id = " +  v;
    return res;
}
```

Function `f` takes as input parameter `v` and returns variable `res`. Let us suppose that `v` is a statically unknown string, corresponding to the automaton recognizing T (i.e., $\mathsf{Min}(\{\mathsf{T}\})$). The result of the function `f` is a string of the form `id =`T, repeated zero or more times. Since the `while` guard is unknown, the number of iterations is statically unknown, and in turn, also the number of concatenations performed inside the loop body. The goal here is to over-approximate the value returned by the function `f`, i.e., the value of `res` at the end of the function. Let $\mathtt{A}$, reported in Fig. 4a, be the automaton abstracting the value of `res` before starting the second iteration of the loop, and let $\mathtt{A}'$, reported in Fig. 4b be the automaton abstracting the value of `res` at the end of the second iteration. At this point, we want to apply the widening operator $\nabla^n_{\mathcal{T}}$, between $\mathtt{A}$ and $\mathtt{A}'$, working as follows. We first compute $\mathtt{A} \sqcup_{\mathcal{T}} \mathtt{A}'$ (corresponding to the automaton reported in Fig. 4b except that also q_0 is also a final state). On this automaton, we merge any state that recognizes the same $\mathbb{A}_{\mathsf{P}}$-strings of length n, with $n \in \mathbb{N}$. In our example, let n be 2. The



(a) res (A) before 2^{nd} loop iteration

(b) res (A$'$) after 2^{nd} loop iteration

(c) The result of $\mathtt{A} \nabla^2_{\mathcal{T}} \mathtt{A}'$

(d) Minimized version of $\mathtt{A} \nabla^2_{\mathcal{T}} \mathtt{A}'$

FIGURE 4 Example of widening application

resulting automaton is reported in Fig. 4c, where q_0 and q_2 are put together, and the other states are left as singletons. Fig. 4d depicts the minimized version of Fig. 4c.

The widening $\nabla^n_{\mathcal{T}}$ has been proved to meet the widening requirements (i.e., over-approximation of the least upper bounds and convergence on infinite ascending chains) in [18]. The parameter n, tuning the widening precision, is arbitrary and can be chosen by the user. As highlighted in [10], the higher n is, the more the corresponding widening operator is precise in over-approximating lubs of infinite ascending chains (i.e., in fixpoint computations). A classical improvement on widening-based fixpoint computations is to integrate a threshold [23], namely widening is applied to over-approximate lubs when a certain threshold (usually over some property of abstract values) is overcome. In fixpoint computations, we decide to apply the previously defined widening $\nabla^n_{\mathcal{T}}$ only when the number of the states of the lubbed automata overcomes the threshold $\tau \in \mathbb{N}$. This permits us to postpone the widening application, getting more precise abstractions when the automata sizes do not overcome the threshold. At the moment, the threshold τ is not automatically inferred, since it surely requires further investigations.

4.2 | String abstract semantics of IMP

In this section, we define the abstract semantics of the string operators defined in Sect. 3 over the new string domain $\mathcal{T}\text{FA}_{/\equiv}$. Soundness and completeness (or incompleteness) proofs of the TARSIS's abstract semantics are reported in Appendix A. While TARSIS also implements `startsWith` and `endsWith` operators, their abstract semantics is not discussed in this section since we adopt the same ones reported in [10].

Since IMP supports strings, integers and Booleans values, we need a way to merge the corresponding abstract domains. In particular, we abstract integers with the well-known interval abstract domain [1] defined as $\mathsf{Intv} \triangleq \{\ [a, b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\ \} \cup \{\bot_{\mathsf{Intv}}\}$ and Booleans with $\mathsf{Bool} \triangleq \wp(\{\mathtt{true}, \mathtt{false}\})$. As usual, we denote by $\sqcup_{\mathsf{Intv}}$ and $\sqcup_{\mathsf{Bool}}$ the lubs between intervals and Booleans, respectively. In particular, we merge such abstract domains in $\text{VAL}^\sharp$ by the smashed sum abstract domain [24] $\text{VAL}^\sharp \triangleq \mathcal{T}\text{FA}_{/\equiv} \oplus \mathsf{Intv} \oplus \mathsf{Bool}$ that *smashes* the bottom elements of the involved domains into a single one, and adds a new top above the ones from the domains.

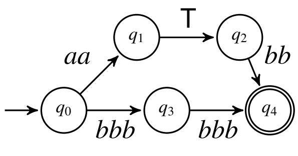
The program state is represented through abstract program memories $\mathbb{M}^\sharp : \text{ID} \rightarrow \text{VAL}^\sharp$ from identifiers to abstract values. The abstract semantics is captured by the function $⟦\ \mathsf{st}\ ⟧^\sharp : \mathbb{M}^\sharp \rightarrow \mathbb{M}^\sharp$, relying on the abstract semantics of expressions defined by, abusing notation, $⟦\ \mathsf{e}\ ⟧^\sharp : \mathbb{M}^\sharp \rightarrow \text{VAL}^\sharp$. We focus on the abstract semantics of string operations, while the semantics of the other expressions is standard and does not involve strings.

Concat

Given $\mathtt{A}, \mathtt{A}' \in \mathcal{T}\text{FA}_{/\equiv}$, the abstract semantics of `concat` returns a new automaton recognizing the language $\{\ \sigma \cdot \sigma' \mid \sigma \in \mathscr{L}(\mathtt{A}), \sigma' \in \mathscr{L}(\mathtt{A}')\ \}$, that is, the concatenation between the strings of $\mathscr{L}(\mathtt{A})$ with the strings of $\mathscr{L}(\mathtt{A}')$. This is achievable relying on the standard automata concatenation [22]. Let $\mathsf{s}, \mathsf{s}' \in \text{SE}$ and suppose that $⟦\ \mathsf{s}\ ⟧^\sharp \mathsf{m}^\sharp = \langle Q, \mathbb{A}_{\mathsf{P}}, \delta, q_0, F\rangle \in \mathcal{T}\text{FA}_{/\equiv}$, $⟦\ \mathsf{s}'\ ⟧^\sharp \mathsf{m}^\sharp = \langle Q', \mathbb{A}_{\mathsf{P}}, \delta', q_0', F'\rangle \in \mathcal{T}\text{FA}_{/\equiv}$. The abstract semantics of `concat` is:

$$⟦\ \mathtt{concat}(\mathsf{s}, \mathsf{s}')\ ⟧^\sharp \mathsf{m}^\sharp \triangleq \mathsf{Min}(\langle Q \cup Q', \mathbb{A}_{\mathsf{P}}, \delta \cup \delta' \cup \{\ (q_f, \epsilon, q_0') \mid q_f \in F\ \}, q_0, F'\rangle)$$

Following the standard automata concatenation, the semantics merges the automata introducing an ϵ-transition from each final state of $\mathtt{A}$ to the initial state of $\mathtt{A}'$. The result's initial state is the initial state of $\mathtt{A}$, while its final states are the ones of $\mathtt{A}'$.



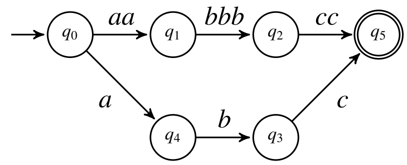


FIGURE 5 (a) $\mathtt{A}$ s.t. $\mathscr{L}(\mathtt{A}) = \{bbb\ bbb, aa\ \top\ bb\}$, (b) $\mathtt{A}'$ s.t. $\mathscr{L}(\mathtt{A}') = \{a\ b\ c, aa\ bbb\ cc\}$

Length

Given $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$, the abstract semantics of `length` returns an interval $[c_1, c_2]$ such that $\forall \sigma \in \mathscr{L}(\mathtt{A}) \,.\, c_1 \leq |\sigma| \leq c_2$. We recast the original idea of the abstract semantics of `length` over standard finite state automata. Let $\mathsf{s} \in \mathtt{SE}$, supposing that $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$. The `length` abstract semantics is:

$$[\![\ \mathtt{length(}\mathsf{s}\mathtt{)}\ ]\!]^\sharp \mathfrak{m}^\sharp \triangleq \begin{cases} [|\mathsf{minPath}(\mathtt{A})|, +\infty] & \text{if } \mathsf{cyclic}(\mathtt{A}) \vee \mathsf{readsTop}(\mathtt{A}) \\ [|\mathsf{minPath}(\mathtt{A})|, |\mathsf{maxPath}(\mathtt{A})|] & \text{otherwise} \end{cases}$$

where $\mathsf{readsTop}(\mathtt{A}) \Leftrightarrow \exists q, q' \in Q \,.\, (q, \top, q') \in \delta$. Note that, when evaluating the length of the minimum path, $\top$ is considered to have a length of 0. For instance, consider the automaton $\mathtt{A}$ reported in Fig. 5a. The minimum path of $\mathtt{A}$ is $(q_0, aa, q_1), (q_1, \top, q_2), (q_2, bb, q_4)$ and its length is 4. Since a transition labeled with $\top$ is in $\mathtt{A}$ (and its length cannot be statically determined), the abstract `length` of $\mathtt{A}$ is $[4, +\infty]$. Consider the automaton $\mathtt{A}'$ reported in Fig. 5b. In this case, $\mathtt{A}'$ has no cycles and has no transitions labeled with $\top$ and the length of every string recognized by $\mathtt{A}'$ can be determined. The length of the minimum path of $\mathtt{A}'$ is 3 (below path of $\mathtt{A}'$), the length of the maximum path of $\mathtt{A}'$ is 7 (above path of $\mathtt{A}'$) and consequently the abstract `length` of $\mathtt{A}'$ is $[3, 7]$.

Contains

Given $\mathtt{A}, \mathtt{A}' \in \mathcal{T}\mathrm{FA}_{/\equiv}$, the abstract semantics of `contains` should return `true` if every string of $\mathtt{A}'$ is contained into every string of $\mathtt{A}$, `false` if no string of $\mathtt{A}'$ is contained into any string of $\mathtt{A}$, and $\{\mathtt{true}, \mathtt{false}\}$ in the other cases. For instance, consider the automaton $\mathtt{A}$ depicted in Fig. 7a and suppose we check if it contains the automaton $\mathtt{A}'$ recognizing the language $\{aa, a\}$. The automaton $\mathtt{A}'$ is a *single-path automaton*[25], meaning that every string of $\mathtt{A}'$ is a prefix of its longest string. In this case, the containment of the longest string (on each automaton path) implies the containment of the others, such as in our example, namely it is enough to check that the longest string of $\mathtt{A}'$ is contained into $\mathtt{A}$. Note that, a single-path automaton cannot read the symbol $\top$. We rely on the predicate $\mathsf{singlePath}(\mathtt{A})$ when $\mathtt{A}$ is a non-cyclic single-path automaton and we denote by σ_{sp} its longest string. Let $\mathsf{s}, \mathsf{s}' \in \mathtt{SE}$, supposing that $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$, $[\![\ \mathsf{s}'\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A}' \in \mathcal{T}\mathrm{FA}_{/\equiv}$. The `contains` abstract semantics is:

$$[\![\ \mathtt{contains(}\mathsf{s}, \mathsf{s}'\mathtt{)}\ ]\!]^\sharp \mathfrak{m}^\sharp \triangleq \begin{cases} \{\mathtt{false}\} & \text{if } \mathtt{A}' \sqcap_{\mathcal{T}} \mathsf{FA}(\mathtt{A}) = \mathsf{Min}(\varnothing) \\ \{\mathtt{true}\} & \text{if } \mathsf{singlePath}(\mathtt{A}') \wedge \forall \pi \in \mathsf{paths}(\mathtt{A}^{ac}) \,.\, \sigma_{\mathsf{sp}} \curvearrowright_{\mathsf{s}} \sigma_\pi \\ \{\mathtt{true}, \mathtt{false}\} & \text{otherwise} \end{cases}$$

In the first case, we denote by $\mathsf{FA}(\mathtt{A})$ the *factor* automaton of $\mathtt{A}$, i.e., the automaton recognizing any substring of $\mathtt{A}$. In particular, if none of $\mathtt{A}$'s substrings is part of of $\mathtt{A}'$, the abstract semantics safely returns `false` (checking the emptiness of the greatest lower bound between $\mathsf{FA}(\mathtt{A})$ and $\mathtt{A}'$). Then, if $\mathtt{A}'$ is a single-path automaton, the abstract semantics returns `true` if every path of $\mathtt{A}^{ac}$ reads the longest string of $\mathtt{A}'$, with $\mathtt{A}^{ac}$ being a copy of $\mathtt{A}$ where all the cycles have been removed. Considering $\mathtt{A}^{ac}$ is necessary not only to make paths computable, but also to exclude *optional* strings recognized as part of loops. Here, we abuse notation denoting with $\sigma_{\mathsf{sp}} \curvearrowright_{\mathsf{s}} \sigma_\pi$ the fact that σ_{sp} is a substring of each string in $\mathsf{Flat}(\sigma_\pi)$. Otherwise, $\{\mathtt{true}, \mathtt{false}\}$ is returned.

String equality

Given $\mathtt{A}, \mathtt{A}' \in \mathcal{T}\mathrm{FA}_{/\equiv}$, the abstract semantics of string equality returns `true` when $\mathtt{A}$ and $\mathtt{A}'$ recognize a singleton string and they are equal, `false` if no string recognized by $\mathtt{A}$ is equal to any string recognized by $\mathtt{A}'$, $\{\mathtt{true}, \mathtt{false}\}$ in the other cases. Before defining the abstract semantics of string equality, we define equality between two strings over the alphabet $\Sigma^* \cup \{\top\}$. For example, the strings $a\top b$ and abb may be equal, while $a\top b$ and abd definitely are not. Alg. 1 defines the function $\mathsf{eq} : \{\Sigma \cup \{\top\}\}^* \times \{\Sigma \cup \{\top\}\}^* \to \mathsf{Bool}$, working on the expanded alphabet $\{\Sigma \cup \{\top\}\}^*$.

Algorithm 1 eq algorithm

let $\sigma, \sigma' \in \{\Sigma \cup \{\mathsf{T}\}\}^*$
1: **if** $\sigma = \sigma' \wedge \sigma = \epsilon$ **then**
2: **return** true
3: **else if** $(\sigma = \epsilon \wedge \sigma' = \mathsf{T}) \vee (\sigma' = \epsilon \wedge \sigma = \mathsf{T})$ **then**
4: **return** $\{\mathtt{true}, \mathtt{false}\}$
5: **else if** $(\sigma = \epsilon \wedge \sigma' \neq \mathsf{T}) \vee (\sigma' = \epsilon \wedge \sigma \neq \mathsf{T})$ **then**
6: **return** false
7: **else if** $\sigma = \mathsf{T} \vee \sigma' = \mathsf{T}$ **then**
8: **return** $\{\mathtt{true}, \mathtt{false}\}$
9: **else if** $\sigma_0 \neq \sigma'_0 \wedge \sigma_0 \neq \mathsf{T} \wedge \sigma'_0 \neq \mathsf{T}$ **then**
10: **return** false
11: **else if** $\sigma_0 = \sigma'_0 \wedge \sigma_0 \neq \mathsf{T}$ **then**
12: **return** $\mathsf{eq}(\sigma[1:], \sigma'[1:])$
13: **else if** $\sigma_0 = \mathsf{T} \vee \sigma'_0 = \mathsf{T}$ **then**
14: **return** $\mathtt{false} \sqcup \mathsf{eq}(\sigma, \sigma'[1:]) \sqcup \mathsf{eq}(\sigma[1:], \sigma'[1:]) \sqcup \mathsf{eq}(\sigma[1:], \sigma')$
15: **end if**

Intuitively, eq checks string equality by recursively inspecting smaller suffixes of the given strings (lines 11-15), returning definite answers only when T characters do not appear (lines 2, 6, and 10). Note that, when one of the given strings begins with T (lines 13-14), the algorithm can only prove inequality.

Let $\mathsf{s}, \mathsf{s}' \in \mathtt{SE}$ and suppose $[\![\ \mathsf{s}\ ]\!]^\sharp \mathtt{m}^\sharp = \mathtt{A}$ and $[\![\ \mathsf{s}'\ ]\!]^\sharp \mathtt{m}^\sharp = \mathtt{A}'$. The abstract semantics of string equality is defined as:

$$[\![\ \mathsf{s} == \mathsf{s}'\ ]\!]^\sharp \mathtt{m}^\sharp \triangleq \begin{cases} \{\mathtt{false}\} & \text{if } \mathtt{A} \sqcap_{\mathcal{T}} \mathtt{A}' = \mathsf{Min}(\varnothing) \\ \{\mathtt{true}, \mathtt{false}\} & \text{if } \mathsf{cyclic}(\mathtt{A}) \vee \mathsf{cyclic}(\mathtt{A}') \\ \bigsqcup_{\sigma \in \mathscr{L}(\mathtt{A}), \sigma' \in \mathscr{L}(\mathtt{A}')} \mathsf{eq}(\sigma, \sigma') & \text{otherwise} \end{cases}$$

In the first case, if the greatest lower bound between $\mathtt{A}$ and $\mathtt{A}'$ is $\mathsf{Min}(\varnothing)$, then the automata do not share any common string, hence the abstract semantics returns $\{\mathtt{false}\}$. In the second case, if either $\mathtt{A}$ or $\mathtt{A}'$ are cyclic, the abstract semantics of string equality returns $\{\mathtt{true}, \mathtt{false}\}$. Otherwise, we rely on eq to compare the $\mathbb{A}_{\mathsf{P}}$-strings recognized by $\mathtt{A}$ and $\mathtt{A}'$ and we lub the results. To avoid cluttering the notation, the conversion from strings over $\Sigma^* \cup \{\mathsf{T}\}$ to strings over $\{\Sigma \cup \{\mathsf{T}\}\}^*$ when calling the function eq is implicit.

IndexOf

Given $\mathtt{A}, \mathtt{A}' \in \mathcal{T}\mathrm{FA}_{/\equiv}$, the `indexOf` abstract semantics returns an interval of the first indexes of the strings of $\mathscr{L}(\mathtt{A}')$ inside strings of $\mathscr{L}(\mathtt{A})$, recalling that when at least one string of $\mathscr{L}(\mathtt{A}')$ is not a substring of any string of $\mathscr{L}(\mathtt{A}')$, the resulting interval must take into account -1 as well. Let $\mathsf{s}, \mathsf{s}' \in \mathtt{SE}$ and suppose $[\![\ \mathsf{s}\ ]\!]^\sharp \mathtt{m}^\sharp = \mathtt{A}$ and $[\![\ \mathsf{s}'\ ]\!]^\sharp \mathtt{m}^\sharp = \mathtt{A}'$. The abstract semantics of `indexOf` is defined as:

$$[\![\ \mathtt{indexOf}(\mathsf{s}, \mathsf{s}')\ ]\!]^\sharp \mathtt{m}^\sharp \triangleq \begin{cases} [-1, +\infty] & \text{if } \mathsf{cyclic}(\mathtt{A}) \vee \mathsf{cyclic}(\mathtt{A}') \vee \mathsf{readsTop}(\mathtt{A}') \\ [-1, -1] & \text{if } \forall \sigma' \in \mathscr{L}(\mathtt{A}')\ \nexists \sigma \in \mathscr{L}(\mathtt{A})\,.\, \sigma' \curvearrowright_{\mathsf{s}} \sigma \\ \displaystyle\bigsqcup_{\sigma \in \mathscr{L}(\mathtt{A}')}^{\mathsf{Intv}} \mathsf{IO}(\mathtt{A}, \sigma) & \text{otherwise} \end{cases}$$

If one of the automata has cycles or the automaton abstracting strings we aim to search for (i.e., $\mathtt{A}'$) has a T-transition, we return $[-1, +\infty]$. Moreover, if none of the strings recognized by $\mathtt{A}'$ is contained in a string recognized by $\mathtt{A}$ (note that this is a decidable check since $\mathtt{A}$ and $\mathtt{A}'$ are cycle-free, otherwise the interval $[-1, +\infty]$ would be returned in the first case), we can safely return the precise interval $[-1, -1]$. Otherwise, we rely on the auxiliary function $\mathsf{IO} : \mathcal{T}\mathrm{FA}_{/\equiv} \times \Sigma^* \rightarrow \mathsf{Intv}$, that, given an automaton $\mathtt{A}$ and a string $\sigma \in \Sigma^*$, returns an interval corresponding to the possible first positions of σ in strings recognized by $\mathtt{A}$. Since $\mathtt{A}'$ recognizes a finite language, we apply $\mathsf{IO}(\mathtt{A}, \sigma)$ to each $\sigma \in \mathscr{L}(\mathtt{A}')$ and to return the least upper bound of the resulting intervals. In particular, the function $\mathsf{IO}(\mathtt{A}, \sigma)$ returns an interval $[i, j] \in \mathsf{Intv}$ where, i and j are computed as follows.

$$i = \begin{cases} -1 & \text{if } \exists \pi \in \mathsf{paths}(\mathtt{A})\,.\, \sigma \not\curvearrowright_{\mathsf{s}} \sigma_\pi \\ \min\limits_{\pi \in \mathsf{paths}(\mathtt{A})} \left\{ i \;\middle|\; \sigma_f \in \mathsf{Flat}(\sigma_\pi) \wedge \sigma_{f_i} \ldots \sigma_{f_{i+n}} = \sigma \right\} & \text{otherwise} \end{cases}$$

$$
j = \begin{cases} -1 & \text{if } \forall \pi \in \mathsf{paths}(\mathtt{A})\,.\, \sigma \not\curvearrowright_{\mathsf{s}} \sigma_\pi \\ +\infty & \text{if } \exists \pi \in \mathsf{paths}(\mathtt{A})\,.\, \sigma \curvearrowright_{\mathsf{s}} \sigma_\pi \wedge \pi \text{ reads } \mathsf{T} \text{ before } \sigma \\ \max\limits_{\pi \in \mathsf{paths}(\mathtt{A})} \left\{ i \,\middle|\, \sigma_f \in \mathsf{Flat}(\sigma_\pi) \wedge \sigma_{f_i} \ldots \sigma_{f_{i+n}} = \sigma \wedge \sigma \not\curvearrowright_{\mathsf{s}} \sigma_{f_0} \ldots \sigma_{f_{i+n-1}} \right\} & \text{otherwise} \end{cases}
$$

As for the abstract semantics of `contains`, we abuse notation denoting with $\sigma \curvearrowright_{\mathsf{s}} \sigma_\pi$ the fact that σ is a substring of each string in $\mathsf{Flat}(\sigma_\pi)$. Given $\mathsf{IO}(\mathtt{A}, \sigma) = [i, j] \in \mathsf{Intv}$, i corresponds to the minimal position where the first occurrence of σ can be found in $\mathtt{A}$, while j to the maximal one. Let us first focus on the computation of the minimal position. If there exists a path π of $\mathtt{A}$ s.t. σ is not recognized by σ_π, then the minimal position where σ can be found in $\mathtt{A}$ does not exist and -1 is returned. Otherwise, the minimal position where σ begins across π is returned. Let us consider now the computation of the maximal position. If all paths of the automaton do not recognize σ, then -1 is returned. If there exists a path where σ is recognized but the character T appears earlier in the path, then $+\infty$ is returned. Otherwise, the maximal index of the first occurrences of σ across the paths of $\mathtt{A}$ is returned.

Repeat

Given $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ and $[i, j] \in \mathsf{Intv}$, the abstract semantics of `repeat` should return an automaton recognizing the language of every string recognized by $\mathtt{A}$ repeated k-times, with $i \leq k \leq j$. We first define the auxiliary function repeat, reported in Alg. 2, that inputs an automaton $\mathtt{A}$ and $n \in \mathbb{N}$ and returns an automaton recognizing the strings of $\mathtt{A}$ repeated n-times.

Algorithm 2 repeat algorithm

```
    let A ∈ TFA/≡, n ∈ ℕ
 1: if n = 0 then
 2:    return Min({ε})
 3: else if cyclic(A) then
 4:    A′ ← Min({ε})
 5:    for i ∈ [0, n − 1] do
 6:       A′ ← ⟦ concat(A′, A) ⟧♯
 7:    end for
 8:    return A′
 9: else
10:    A_r ← Min(∅)
11:    for π ∈ paths(A) do
12:       A′ ← Min({ε})
13:       for i ∈ [0, n − 1] do
14:          A′ ← ⟦ concat(A′, Min({σ_π})) ⟧♯
15:       end for
16:       A_r ← A_r ⊔_T A′
17:    end for
18:    return A_r
19: end if
```

Lines 1–2 handle the case when n is zero and return the automaton recognizing the empty string. Lines 3–8 handle the case when $\mathtt{A}'$ is a cyclic automaton and they build the automaton $\mathtt{A}'$ returned at line 8, corresponding to the n-concatenation of the automaton $\mathtt{A}$. Otherwise, for each path π of $\mathtt{A}$, lines 12–15 builds the automaton $\mathtt{A}'$ that corresponds to the n-repetition of σ_π, i.e., the string read by the path π. This operation is repeated for each path of the automaton (lines 11–17), and the obtained automata are lubbed together in $\mathtt{A}_r$ at line 16, that is finally returned.

Let $\mathsf{s} \in \mathrm{SE}$ and $\mathsf{a} \in \mathrm{AE}$, supposing that $⟦\, \mathsf{s} \,⟧^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ and $⟦\, \mathsf{a} \,⟧^\sharp \mathfrak{m}^\sharp = [i, j] \in \mathsf{Intv}$. W.l.o.g., let us suppose that $[i, j] \subseteq [0, +\infty]$; when negative values are met, the automaton recognizing the empty language is returned. The `repeat` abstract semantics is:

$$
⟦\, \mathtt{repeat}(\mathsf{s}, \mathsf{a}) \,⟧^\sharp \mathfrak{m}^\sharp = \begin{cases} \mathsf{Kleene}(\mathtt{A}) & \text{if } [i, j] = [0, +\infty] \\ \mathsf{repeat}(\mathtt{A}, i) & \text{if } i = j \wedge i \in \mathbb{N} \\ ⟦\, \mathtt{concat}(\mathsf{repeat}(\mathtt{A}, i), \mathsf{Kleene}(\mathtt{A})) \,⟧^\sharp & \text{if } j = +\infty \\ \bigsqcup_{k \in [i, j]} \mathsf{repeat}(\mathtt{A}, k) & \text{otherwise} \end{cases}
$$

(a) (b)

FIGURE 6 (a) A, (b) $[\![$ `repeat(A,`$[2,+\infty]$`)` $]\!]^\sharp$

When the input interval is $[0,+\infty]$, the first case is matched and the Kleene closure of $\mathtt{A}$ is returned. The second case is when the interval concretizes to a single value (e.g., $[2,2]$) and the abstract semantics returns $\mathsf{repeat}(\mathtt{A}, i)$. If the interval is $[i,+\infty]$, with $i \in \mathbb{N} \smallsetminus \{0\}$, since it is excluded in the first case, the abstract semantics returns the concatenation between the i-repetition of $\mathtt{A}$ with its Kleene closure. An example of this case is reported in Fig. 6. Otherwise, i.e., the interval $[i,j]$ is finite, the `repeat`'s abstract semantics returns the lub of the k-repetition of $\mathtt{A}$, for each $i \leq k \leq j$.

TrimLeft, TrimRight, and Trim

The concrete semantics of `trimLeft` removes leading whitespaces (i.e., at the begin) from a string. Similarly, its abstract semantics inputs an automaton $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ and removes leading whitespaces from the begin of each string recognized by $\mathtt{A}$. Let $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ and $\mathtt{r}$ be the regex corresponding to the language recognized by $\mathtt{A}$. The `trimLeft`'s abstract semantics is captured by the function `trimL` inductively defined on the structure of regexes as follows.

$$\mathtt{trimL}(\mathtt{r}) = \begin{cases} \mathtt{r} & \text{if } \mathtt{r} = \mathsf{T} \vee \mathtt{r} = \varnothing \\ [\![\, \mathtt{trimLeft}(\sigma) \,]\!] & \text{if } \mathtt{r} = \sigma \\ \mathtt{trimL}(\mathtt{r}_1) \parallel \mathtt{trimL}(\mathtt{r}_2) & \text{if } \mathtt{r} = \mathtt{r}_1 \parallel \mathtt{r}_2 \\ \mathtt{trimL}(\mathtt{r}_2) & \text{if } \mathtt{r} = \mathtt{r}_1\mathtt{r}_2 \wedge \mathtt{trimL}(\mathtt{r}_1) = \epsilon \\ \mathtt{trimL}(\mathtt{r}_1)(\mathtt{r}_2 \parallel \mathtt{trimL}(\mathtt{r}_2)) & \text{if } \mathtt{r} = \mathtt{r}_1\mathtt{r}_2 \wedge \mathsf{readWS}(\mathtt{r}_1) \\ \mathtt{trimL}(\mathtt{r}_1)\mathtt{r}_2 & \text{if } \mathtt{r} = \mathtt{r}_1\mathtt{r}_2 \wedge \neg\mathsf{readWS}(\mathtt{r}_1) \\ \epsilon & \text{if } \mathtt{r} = (\mathtt{r}_1)^* \wedge \mathtt{trimL}(\mathtt{r}_1) = \epsilon \\ \epsilon \parallel \mathtt{trimL}(\mathtt{r}_1)\ \mathtt{r} & \text{if } \mathtt{r} = (\mathtt{r}_1)^* \end{cases}$$

The predicate $\mathsf{readWS}(\mathtt{r})$ holds if the language of $\mathtt{r}$ recognizes a whitespace string, i.e., $\mathsf{readWS}(\mathtt{r}) \iff \exists ws \in \{␣\}^* .\, ws \in \mathscr{L}(\mathtt{r})$. If the regex is empty or T, `trimL` behaves as the identity function (first case). If the regex is an atom, we rely on the concrete semantics of `trimLeft` (second case). If the regex is a disjunction, the result is the disjunction of the application of `trimL` of the operands (third case). Then, three cases are needed for the regex concatenation $\mathtt{r}_1\mathtt{r}_2$. If $\mathtt{r}_1$ recognizes only whitespace strings (i.e., $\mathtt{trimL}(\mathtt{r}_1) = \epsilon$), then we need to trim left also $\mathtt{r}_2$ (fourth case). If $\mathtt{r}_1$ recognizes at least one whitespace string, it *might* be necessary to also trim $\mathtt{r}_2$, hence $\mathtt{trimL}(\mathtt{r}_1)$ is concatenated with the disjunction of $\mathtt{r}_2$ and $\mathtt{trimL}(\mathtt{r}_2)$ (fifth case). For instance, let us consider the regex $(␣ \parallel a)␣\, b$. The concrete semantics of `trimLeft` would return $\{a␣b, b\}$. Our abstract semantics, in this case, correctly returns $\mathtt{trimL}(␣ \parallel a)\ (␣b \parallel \mathtt{trimL}(␣b)) = a(␣b \parallel b) = a␣b \parallel b$. Lastly, if $\mathtt{r}_1$ does not recognize any whitespace string, $\mathtt{r}_1$ is left-trimmed and concatenated with $\mathtt{r}_2$ (sixth case). Finally, if $\mathtt{r} = (\mathtt{r}_1)^*$, two cases are identified. If $\mathtt{r}_1$ recognizes only whitespace strings, the empty string is returned (seventh case); otherwise the result may be the empty string (in the case of 0-repetition) or the whole regex $\mathtt{r}$ appended to trimmed-left $\mathtt{r}_1$ (eighth case). Similarly, we can define `trimR` that removes trailing whitespace from the input regex. The definition of `trimR` is left implicit since it is analogous to the one of `trimL`.

Let $\mathsf{s} \in \mathtt{SE}$, supposing that $[\![\, \mathsf{s} \,]\!]^\sharp\mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ and let $\mathtt{r}$ be the regex equivalent to $\mathtt{A}$. The abstract semantics of `trimLeft`, `trimRight`, and `trim` are:

$$[\![\, \mathtt{trimLeft(s)} \,]\!]^\sharp\mathfrak{m}^\sharp = \mathtt{trimL}(\mathtt{r}) \quad [\![\, \mathtt{trimRight(s)} \,]\!]^\sharp\mathfrak{m}^\sharp = \mathtt{trimR}(\mathtt{r}) \quad [\![\, \mathtt{trim(s)} \,]\!]^\sharp\mathfrak{m}^\sharp = \mathtt{trimL}(\mathtt{trimR}(\mathtt{r}))$$

Replace

To give the intuition about how the abstract semantics of `replace` works, consider three automata $\mathtt{A}, \mathtt{A}_s, \mathtt{A}_r \in \mathcal{T}\mathrm{FA}_{/\equiv}$. Let us refer to $\mathtt{A}_s$ as the *search automaton* and to $\mathtt{A}_r$ as the *replace automaton*. Roughly speaking, the abstract semantics of `replace` substitutes strings of $\mathtt{A}_s$ with strings of $\mathtt{A}_r$ inside strings of $\mathtt{A}$. We need to specify two types of possible replacements, by means

FIGURE 7 Example of may-replacement

of the following example. Consider $\mathbb{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ that is depicted in Fig. 7a and suppose that the search automaton $\mathbb{A}_s$ is the one recognizing the string *bbb* and the replace automaton $\mathbb{A}_r$ is a random automaton. In this case, the `replace` abstract semantics performs a *must-replace* over $\mathbb{A}$, namely substituting the sub-automaton composed by q_1 and q_2 with the replace automaton $\mathbb{A}_r$. Instead, let us suppose that the search automaton $\mathbb{A}_s$ is the one recognizing the language $\{bbb, cc\}$. Since it is unknown which string *must* be replaced (between *bbb* and *cc*), the `replace` abstract semantics needs to perform a *may-replace*: when a string recognized by the search automaton is met inside a path of $\mathbb{A}$ it is left unaltered in the automaton and, in the same position where the string is met, the abstract `replace` only extends $\mathbb{A}$ with the replace automaton. An example of may replacement is reported in Fig. 7, where $\mathbb{A}$ is the one reported in Fig. 7a, the search automaton $\mathbb{A}_s$ is the one recognizing the language $\{bbb, cc\}$ and the replace automaton $\mathbb{A}_r$ is the one recognizing the string *rr*.

Before introducing the abstract semantics of `replace`, we define how to replace a string into an automaton's path. In particular, we define algorithm RP in Alg. 3, that given a path π of an arbitrary automaton, a replace automaton $\mathbb{A}^r$, and $\sigma^s \in \Sigma^* \cup \{\mathsf{T}\}$ returns a new automaton built starting from the path, but where portions of the path that recognize σ^s have been replaced with $\mathbb{A}^r$.

Algorithm 3 RP algorithm

`let` $\pi = (q_0, \sigma_0, q_1), \ldots, (q_{n-1}, \sigma_{n-1}, q_n), \mathbb{A}^r = \langle Q^r, \mathbb{A}, \delta^r, q_0^r, F^r \rangle \in \mathcal{T}\mathrm{FA}_{/\equiv}, \sigma^s \in \Sigma^* \cup \{\mathsf{T}\}$
1: $Q^{result} \leftarrow \{\ q \mid (q, \sigma, q') \in \pi \vee (q', \sigma, q) \in \pi\ \}$
2: $\delta^{result} \leftarrow \pi$
3: **for** $(q_i, \sigma_0^s, q_{i+1}), \ldots, (q_{i+n-1}, \sigma_n^s, q_{i+n}) \in \pi$ **do**
4: $\quad \langle Q', \mathbb{A}, \delta', q_0', F' \rangle \leftarrow$ `clone`$(\mathbb{A}^r)$
5: $\quad \delta^{result} \leftarrow \delta^{result} \cup (q_i, \epsilon, q_0')$
6: $\quad \delta^{result} \leftarrow \delta^{result} \cup \{\ (q_f, \epsilon, q_{i+n}) \mid q_f \in F'\ \}$
7: $\quad Q^{result} \leftarrow Q^{result} \setminus \{q_{i+1}, \ldots, q_{i+n-1}\}$
8: $\quad \delta^{result} \leftarrow \delta^{result} \setminus \{(q_i, \sigma_0^s, q_{i+1}), \ldots, (q_{i+n-1}, \sigma_n^s, q_{i+n})\}$
9: **end for**
10: **return** $\langle Q^{result}, \mathbb{A}, \delta^{result}, q_0^o, F^o \rangle$

Alg. 3 searches the given string σ^s across path π, collecting the sequences of transitions that recognize the search string σ^s and extracting them from π (line 3). Whenever a matching sequence is found, $\mathbb{A}^r$ is cloned to $\mathbb{A}'$ to ensure that all additions target a different set of nodes (line 4). Then, an ϵ-transition is introduced going from the first state of the sequence to the initial state of $\mathbb{A}'$, and one such transition is also introduced for each final state of $\mathbb{A}'$, connecting that state with the ending state of the sequence (lines 5-6). The list of states composing the sequence of transitions is then removed from the result (line 7), together with the transitions connecting them (line 8), since those were needed only to recognize the string that has been replaced. Note that RP corresponds to a must-replace. At this point, we are ready to define the `replace` abstract semantics. In particular, if either $\mathbb{A}$ or $\mathbb{A}_s$ have cycles or if one of them has a T-transition, we return $\mathsf{Min}(\{\mathsf{T}\})$, namely the automaton recognizing T. Otherwise, the `replace` abstract semantics is:

$$\llbracket\ \texttt{replace}(\mathsf{s}, \mathsf{s}_s, \mathsf{s}_r)\ \rrbracket^\sharp \mathsf{m}^\sharp \triangleq \begin{cases} \mathbb{A} & \text{if } \forall \sigma_s \in \mathscr{L}(\mathbb{A}_s) \,.\, \nexists \sigma \in \mathscr{L}(\mathbb{A}) \,.\, \sigma_s \curvearrowright_{\mathsf{s}} \sigma \\ \bigsqcup_{\pi \in \mathsf{paths}(\mathbb{A})} \mathsf{RP}(\pi, \sigma_s, \mathbb{A}_r) & \text{if } \mathscr{L}(\mathbb{A}_s) = \{\sigma_s\} \\ \bigsqcup_{\substack{\sigma \in \mathscr{L}(\mathbb{A}_s) \\ \pi \in \mathsf{paths}(\mathbb{A})}} \mathsf{RP}(\pi, \sigma, \mathbb{A}_r \sqcup_{\mathcal{T}} \mathsf{Min}(\{\sigma\})) & \text{otherwise} \end{cases}$$

In the first case, if none of the strings recognized by the search automaton $\mathtt{A}_s$ is contained in strings recognized by $\mathtt{A}$, we can safely return the original automaton $\mathtt{A}$ without any replacement. In the special case where $\mathscr{L}(\mathtt{A}_s) = \{\sigma_s\}$, we return the automaton obtained by replacing σ_s across all paths of $\mathtt{A}$ using function $\mathsf{RP}(\pi, \sigma_s, \mathtt{A}_r)$. In the last case, for each string $\sigma \in \mathscr{L}(\mathtt{A}_s)$ and for each path $\pi \in \mathsf{paths}(\mathtt{A})$, we perform a may-replace of σ with $\mathtt{A}_r$: note that, this exactly corresponds to a call to RP where the replace automaton is $\mathtt{A}_r \sqcup_{\mathcal{T}} \mathsf{Min}(\{\sigma\})$. The so far obtained automata are finally lubbed together.

Algorithm 4 Sb algorithm

let r regex over $\mathbb{A}$, $i,j \in \mathbb{N}$
if $j = 0 \vee \mathtt{r} = \varnothing$ **then**
  return $\varnothing$
else if $\mathtt{r} = \sigma \in \Sigma^*$ **then**
  if $i > |\sigma|$ **then**
    return $\{(\epsilon, i - |\sigma|, j)\}$
  else if $i + j > |\sigma|$ **then**
    return $\{(\sigma_i \dots \sigma_{|\sigma|-1}, 0, j - |\sigma| + i)\}$
  else
    return $\{(\sigma_i \dots \sigma_{i+j}, 0, 0)\}$
  end if
else if $\mathtt{r} = \mathsf{T}$ **then**
  $\mathsf{result} \leftarrow \{(\epsilon, i-k, j) : 0 \leq k \leq i, k \in \mathbb{N}\}$
  $\mathsf{result} \leftarrow \mathsf{result} \cup \{\ (\bullet^k, 0, j-k) \mid 0 \leq k \leq j, k \in \mathbb{N}\ \}$
  return result
else if $\mathtt{r} = \mathtt{r}_1\mathtt{r}_2$ **then**
  $\mathsf{result} \leftarrow \emptyset$
  $\mathsf{subs}_1 \leftarrow \mathsf{Sb}(\mathtt{r}_1, i, j)$
  for $(\sigma_1, i_1, j_1) \in \mathsf{subs}_1$ **do**
    if $j_1 = 0$ **then**
      $\mathsf{result} \leftarrow \mathsf{result} \cup \{(\sigma_1, i_1, j_1)\}$
    else
      $\mathsf{result} \leftarrow \mathsf{result} \cup \{\ (\sigma_1 \cdot \sigma_2, i_2, j_2) \mid (\sigma_2, i_2, j_2) \in \mathsf{Sb}(\mathtt{r}_2, i_1, j_1)\ \}$
    end if
  end for
  return result
else if $\mathtt{r} = \mathtt{r}_1 || \mathtt{r}_2$ **then**
  return $\mathsf{Sb}(\mathtt{r}_1, i, j) \cup \mathsf{Sb}(\mathtt{r}_2, i, j)$
else if $\mathtt{r} = (\mathtt{r}_1)^*$ **then**
  $\mathsf{result} \leftarrow \{(\epsilon, i, j)\}$
  $\mathsf{partial} \leftarrow \varnothing$
  repeat
    $\mathsf{result} \leftarrow \mathsf{result} \cup \mathsf{partial}$;
    $\mathsf{partial} \leftarrow \varnothing$
    for $(\sigma_n, i_n, j_n) \in \mathsf{result}$ **do**
      for $(\mathsf{suff}, i_s, j_s) \in \mathsf{Sb}(\mathtt{r}_1, i_n, j_n)$ **do**
        if $(\sigma_n \cdot \mathsf{suff}, i_s, j_s) \notin \mathsf{result}$ **then**
          $\mathsf{partial} \leftarrow \mathsf{partial} \cup \{(\sigma_n \cdot \mathsf{suff}, i_s, j_s)\}$
        end if
      end for
    end for
  until $\mathsf{partial} \neq \varnothing$
  return result
end if

Substr and CharAt
Given $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ and two intervals $\mathsf{i}, \mathsf{j} \in \mathsf{Intv}$, the abstract semantics of `substr` returns a new automaton $\mathtt{A}'$ soundly approximating any substring from i to j of strings recognized by $\mathtt{A}$, for any $i \in \mathsf{i}, j \in \mathsf{j}$ s.t. $i \leq j$.

Given $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$, in the definition of the `substr` semantics, we rely on the corresponding regex $\mathtt{r}$ since the two representations are equivalent and regexes allow us to define a more intuitive formalization of the semantics of `substr`. Let us suppose that $[\![\ \mathsf{s}\ ]\!]^\sharp \mathtt{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ and let us denote by $\mathtt{r}$ the regex corresponding to the language recognized by $\mathtt{A}$. At the moment, let us consider exact intervals representing one integer value, namely $[\![\mathsf{a}_1]\!]^\sharp \mathfrak{m}^\sharp = [i,i]$ and $[\![\mathsf{a}_2]\!]^\sharp \mathfrak{m}^\sharp = [j,j]$, with $i,j \in \mathbb{Z}$. In this case, the abstract semantics is defined as:

$$[\![\texttt{substr}(\mathsf{s},\mathsf{a}_1,\mathsf{a}_2)]\!]^\sharp \mathfrak{m}^\sharp \triangleq \bigsqcup \mathsf{Min}(\{ \sigma \mid (\sigma,0,0) \in \mathsf{Sb}(\mathtt{r},i,j-i) \})$$

where Sb takes as input a regex $\mathtt{r}$, two indexes $i,j \in \mathbb{N}$, and computes the set of substrings from i to j of all the strings recognized by $\mathtt{r}$. In particular, Sb is defined by Alg. 4 and, given a regex $\mathtt{r}$ and $i,j \in \mathbb{N}$, it returns a set of triples of the form (σ,n_1,n_2), such that σ is the *partial substring* that Alg. 4 has computed up to now, $n_1 \in \mathbb{N}$ tracks how many characters have still to be skipped before the substring can be computed and $n_2 \in \mathbb{N}$ is the number of characters Alg. 4 still needs to look for to successfully compute a substring. Hence, given $\mathsf{Sb}(\mathtt{r},i,j)$, the result is a set of such triples; note that given an element of the resulting set (σ,n_1,n_2), $n_2 = 0$ means that no more characters are needed and σ corresponds to a proper substring of $\mathtt{r}$ from i to j. Thus, from the resulting set, we can filter out the partial substrings, and retrieve only proper substrings of $\mathtt{r}$ from i to j, by only considering the value of n_2. Alg. 4 is defined by case on the structure of the input regex $\mathtt{r}$:

1. $j = 0$ or $\mathtt{r} = \varnothing$ (lines 1-2): $\varnothing$ is returned since we either completed the substring or we have no more characters to add;
2. $\mathtt{r} = \sigma \in \Sigma^*$ (lines 3-10): if $i > |\sigma|$, the requested substring happens after this atom, and we return a singleton set $\{\epsilon, i-|\sigma|, j\}$, thus tracking the consumed characters before the start of the requested substring; if $i+j > |\sigma|$, the substring begins in σ but ends in subsequent regexes, and we return a singleton set containing the substring of σ from i to its end, with $n_1 = 0$ since we begun collecting characters, and $n_2 = j - |\sigma| + i$ since we collected $|\sigma| - i$ characters; otherwise, the substring is fully inside σ, and we return the substring of σ from i to $i+j$, setting both n_1 and n_2 to 0;
3. $\mathtt{r} = \top$ (lines 11-14): since $\mathtt{r}$ might have any length, we generate substrings that (a) gradually consume all the missing characters before the substring can begin (line 12) and (b) gradually consume all the characters that make up the substring, adding the unknown character $\bullet$ (line 13);
4. $\mathtt{r} = \mathtt{r}_1\mathtt{r}_2$ (lines 15-25): the desired substring can either be fully found in $\mathtt{r}_1$ or $\mathtt{r}_2$, or could overlap them; thus we compute all the partial substrings of $\mathtt{r}_1$, recursively calling Sb (line 17); for all $\{\sigma_1, i_1, j_1\}$ returned, substrings that are fully contained in $\mathtt{r}_1$ (i.e., when $j_1 = 0$) are added to the result (line 20) while the remaining ones are joined with ones computed by recursively calling Sb on $\mathtt{r}_2$ with $n_1 = j_1$ and $n_2 = j_2$;
5. $\mathtt{r} = \mathtt{r}_1 || \mathtt{r}_2$ (lines 26-27): we return the partial substring of $\mathtt{r}_1$ and the ones of $\mathtt{r}_2$, recursively calling Sb on both of them;
6. $\mathtt{r} = (\mathtt{r}_1)^*$ (lines 28-42): we construct the set of substrings through fixpoint iteration, starting by generating $\{\epsilon, i, j\}$ (corresponding to $\mathtt{r}_1$ repeated 0 times - line 29) and then, at each iteration, by joining all the partial results obtained until now with the ones generated by a further recursive call to Sb, keeping only the joined results that are new (lines 31-42).

Above, we have defined the abstract semantics of `substr` when intervals are constant. When $[\![\mathsf{a}_1]\!]^\sharp \mathfrak{m}^\sharp = [i,j]$ and $[\![\mathsf{a}_2]\!]^\sharp \mathfrak{m}^\sharp = [l,k]$, with $i,j,l,k \in \mathbb{Z}$, the abstract semantics of `substr` is

$$[\![\texttt{substr}(\mathsf{s},\mathsf{a}_1,\mathsf{a}_2)]\!]^\sharp \mathfrak{m}^\sharp \triangleq \bigsqcup_{a\in[i,j],b\in[l,k],a\leq b} \mathsf{Min}(\{ \sigma \mid (\sigma,0,0) \in \mathsf{Sb}(\mathtt{r},a,b-a) \})$$

We do not report the cases when input intervals are unbounded (e.g., $[1,+\infty]$). Nevertheless, these cases have been already considered in [10] and treated analogously in our implementation.

We exploit the abstract semantics of `substr` to instantiate the one of `charAt` as a special case of the former:

$$[\![\texttt{charAt}(\mathsf{s},\mathsf{a}_1)]\!]^\sharp \mathfrak{m}^\sharp \triangleq [\![\texttt{substr}(\mathsf{s},\mathsf{a}_1,\mathsf{a}_1+1)]\!]^\sharp \mathfrak{m}^\sharp$$

5 | EXPERIMENTAL RESULTS

TARSIS has been compared with five string abstract domains, namely the prefix (PR), suffix (SU), char inclusion (CI), bricks (BR) domains (all defined in [9]), and $\text{FA}_{/\equiv}$ (defined in [10]). All domains have been implemented in GoLiSA, that we will briefly introduce before presenting our experimental results. TARSIS and $\text{FA}_{/\equiv}$ share a common implementation for the automata structure that is

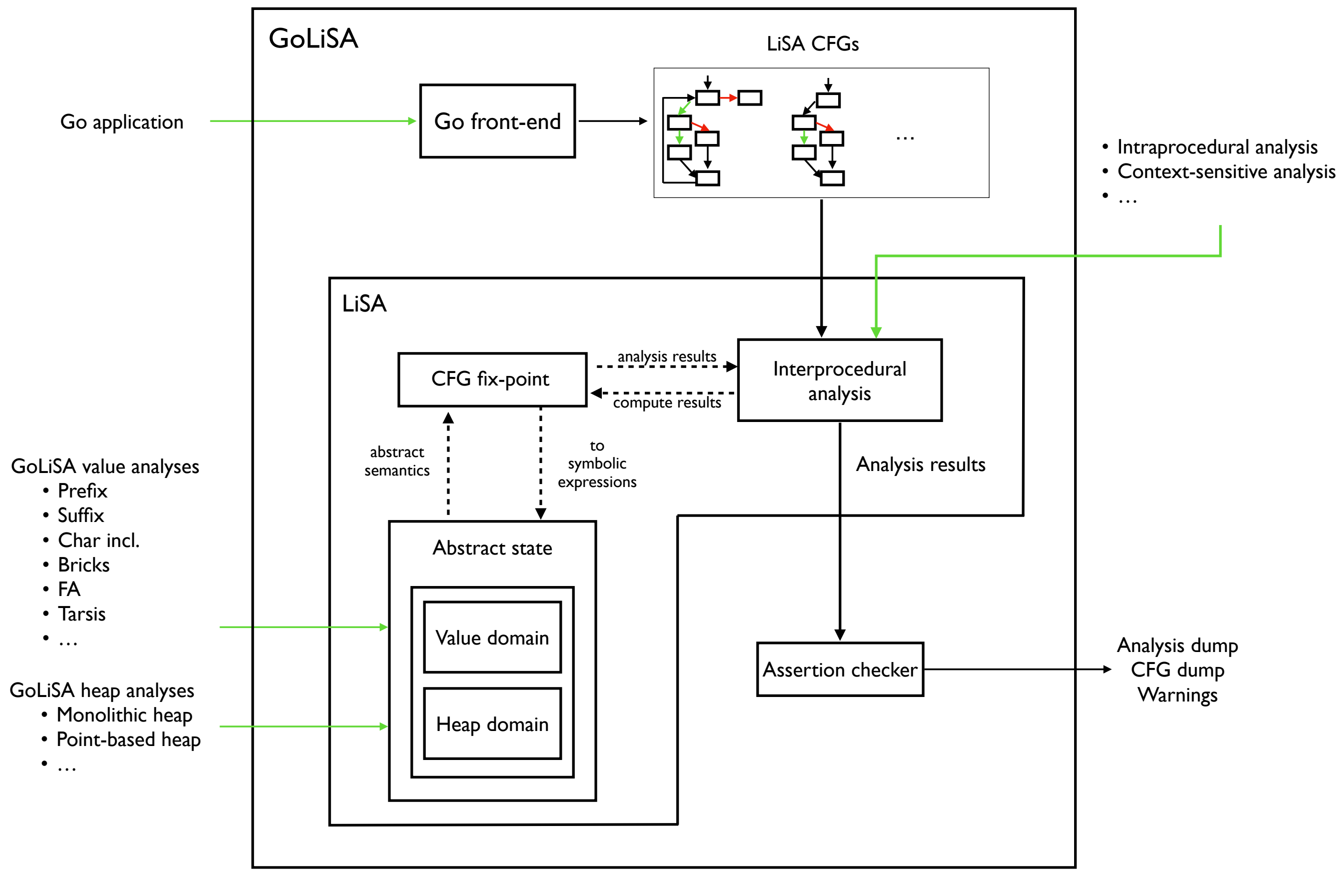


FIGURE 8 Schema of GoLiSA's architecture (taken from [12])

parametric to the alphabet they use. This ensures that performance differences can be accounted on the different size of the automata, eliminating biases that could be introduced with separate implementations having different degrees of optimization.

Comparisons have been performed by analyzing the code through the coalesced sum domain specified in Sect. 4.2 with trace partitioning [26] (note that all traces are merged when evaluating an assertion), plugging in the various string domains. All experiments have been performed on a HP EliteBook G6 machine, with an Intel Core i7-8565U @ 1.8GHz processor and 16 GB of RAM memory.

To achieve a fair comparison with the other string domains, the subjects of our evaluation are small hand-crafted code fragments that represent standard string manipulations that occur regularly in software. PR, SU, CI and BR have been built to model simple properties and to work with integers instead of intervals, and have been evaluated on small programs: Sect. 5.1 compares them to TARSIS and $\text{FA}_{/\equiv}$ without expanding the scope of such evaluations. Sect. 5.2 instead focuses on slightly more advanced and complex string manipulations that are not modeled by the aforementioned domains, but that $\text{FA}_{/\equiv}$ and TARSIS can indeed tackle, highlighting differences between them. Finally, Sect. 5.4 focuses on the performance difference between $\text{FA}_{/\equiv}$ and TARSIS, benchmarking their lattice operations and abstract transformers.

LiSA and GoLiSA

Experiments presented in this paper have been performed using GoLiSA (https://github.com/lisa-analyzer/go-lisa), a static analyzer for Go based on LiSA [13,14], whose high level infrastructure is visible in Fig. 8.

LiSA (**Li**brary for **S**tatic **A**nalysis) is a modular framework for developing static analyzers based on the abstract interpretation theory. LiSA analyzes CFGs whose statements do not have a predefined semantics: instead, users of the framework define custom statement instances implementing language-specific semantic functions, enabling the analysis of a wide range of programming languages and the development of multilanguage analyses. The analysis infrastructure is partitioned into three main areas: call evaluation, memory modeling and value analysis. Each area corresponds to a separate configurable analysis component, that

```go
1  func Subs(nondet bool) {
2    res := "substring test"
3    if (nondet) {
4      res = res + " passed"
5    } else {
6      res = res + " failed"
7    }
8    res = res[5:18]
9    assert (strings.Contains(res, "g"));
10   assert (strings.Contains(res, "p"));
11   assert (strings.Contains(res, "f"));
12   assert (strings.Contains(res, "d"));
13 }
```

(a) Program SUBS

```go
func Loop(value string, nondet bool) {
  res := "Repeat: "
  for nondet {
    res = res + value + "!"
  }
  assert (strings.Contains(res, "t"));
  assert (strings.Contains(res, "!"));
  assert (strings.Contains(res, "f"));
}
```

(b) Program LOOP

FIGURE 9 Program samples used for domain comparison

Domain	Program SUBS		Program LOOP	
PR	`ring test`	✗	`Repeat:`	✗
SU	ϵ	✗	ϵ	✗
CI	`[][abdefgilnprstu ]`	✗	`[:aepRt ][!:aepRt ⊤]`	✓
BR	`[{ring test fai,ring test pas}]`$(1,1)$	✓	$[\{\top\}]\,(0,+\infty)$	✗
$\mathrm{FA}_{/\equiv}$	`ring test (pas ∥ fai)`	✓	`Repeat:` $(\top)^*$	✓
TARSIS	`(ring test pas ∥ ring test fai)`	✓	`Repeat:` $(\top!)^*$	✓

TABLE 1 Values of `res` at the first assert of each program

operates agnostically w.r.t. how the others are implemented. The analysis begins in the *Interprocedural Analysis*, that executes a program-wide fixpoint by computing each individual CFG's fixpoint. Whenever a call is encountered, the computation of its result is delegated back to the *Interprocedural Analysis*. Instead, non-calling statements are decomposed into a sequence of atomic operations, called *symbolic expressions*, each with a precise semantics that the abstract domains can interpret. Memory-dealing expressions are handled by the *Heap Domain*, tracking their effect and rewriting them as abstract identifiers representing possible memory locations. Finally, the *Value Domain* tracks properties about variables (either program variables or abstract identifiers) and computes invariants for each program point.

Code parsing logic and the definition of language-specific statements are provided by *Frontends* such as GoLiSA, that can also provide implementations for LiSA's components. These constitute effective static analyzers for individual languages, that can be combined to obtain multilanguage analyses. In particular, GoLiSA provides a Go-specific *Heap Domain* to accurately track memory operations, while it exploits a context-based *Interprocedural Analysis* provided out-of-the-box by LiSA. All string abstractions considered in the evaluation are implemented as *Value Domain*s. Finally, after an analysis is completed, GoLiSA executes the *Assertion Checker*, that is, a program visitor that can access the results of string analyses to raise *definite* alarms (DA for short) when a failing assert (i.e., whose condition is definitely false) is met, or *possible* alarms (PA for short) when the assertion *might* fail (i.e., the assertion's condition evaluates to $\top_{\mathsf{Bool}}$). Note that Go does not have built-in `assert` instructions, and we simulate them by invoking a function that `panics` when the given condition is `false`.

5.1 | Precision of the various domains on test cases

We start by considering programs SUBS (Fig. 9a) and LOOP (Fig. 9b). SUBS calls `substr` on the concatenation between two strings, where the first is constant and the second one is chosen in a non-deterministic way (i.e., `nondet` condition is statically unknown, lines 3-7). LOOP builds a string by repeatedly appending a suffix, which contains an user input (i.e., an unknown string), to a constant value. Tab. 1 reports the value approximation for `res` for each abstract domain and analyzed program when the first assertion of each program is met, as well as if the abstract domain precisely dealt with the program assertions. For the sake of readability, TARSIS and $\mathrm{FA}_{/\equiv}$ approximations are expressed as regexes.

When analyzing SUBS, both PR and SU lose precision since the string to append to `res` is statically unknown. This leads, at line 8, to a partial substring of the concrete one with PR, and to an empty string with SU. Instead, the `substring` semantics of CI moves every character of the receiver in the set of possibly contained ones, thus the abstract value at line 8 is composed by an

```
1  func ToString(names []string) {
2    res := "People : {"
3    i := 0;
4    for i < len(names) {
5      res = res + names[i];
6      if i != len(names) -1 {
7        res = res + ",";
8      }
9      i++;
10   }
11   res = res + "}";
12   assert (strings.Contains(res, "People"));
13   assert (strings.Contains(res, ","));
14   assert (strings.Contains(res, "not"));
15 }
```

(a) Program ToString

```
func CountMatches(nondet boolean) {
  str := ""
  if nondet {
    str = "this is the thing";
  } else {
    str = "the throat";
  }
  count := strings.Count(str, "th");
  assert (count>0);
  assert (count==0);
  assert (count==3);
}
```

(b) Program CountMatches

FIGURE 10 Programs used for assessing domain precision

empty set of included characters, and a set of possibly included characters containing the ones of both strings. Finally, Br, $\text{FA}_{/\equiv}$ and Tarsis are expressive enough to track any string produced by any concrete execution of Subs.

When evaluating the assertions of Subs, a PA should be raised on lines 10 and 11, since `"p"` or `"f"` might be in `res`, together with a DA alarm on line 12, since `"d"` is surely not contained in `res`. No alarm should be raised on line 9 instead, since `"g"` is part of the common prefix of both branches and thus will be included in the substring. Such behavior is achieved when using Br, $\text{FA}_{/\equiv}$, or Tarsis. Since the `substring` semantics of Ci moves all characters to the set of possibly contained ones, PAs are raised on all four assertions. Moreover, Su loses all information about `res`, PAs are raised on lines 9-12 when using such domain. Pr instead tracks the definite prefix of `res`, thus the PA at line 9 is avoided.

When analyzing Loop, we expect to obtain no alarm at line 6 (since character `"t"` is always contained in the resulting string value), and PAs at lines 7 and 8. Pr infers as prefix of `res` the string `"Repeat: "`, keeping such value for the analysis of the whole program. This allows the analyzer to prove the assertion at line 6, but it raises PAs when it checks the ones at lines 7 and 8. Again, Su loses all information about `res` since the lub operation occurring at line 3 cannot find a common suffix between `"Repeat: "` and `"!"`, hence PAs are raised on lines 6-8. Since the set of possible characters contains $\top$, Ci can correctly state that any character might appear in the string. For this reason, two PAs are reported on lines 7 and 8, while no alarm is raised on line 6 (again, this is possible since the string used in the `contains` call has length 1). The alternation of $\top$ and `"!"` prevents Br normalization algorithm from merging similar bricks. This will eventually lead to overcoming the length threshold k_L, hence resulting in the $\left[\{\top\}\right](0,+\infty)$ abstract value. In such a situation, Br returns $\top_{\text{Bool}}$ on all `contains` calls, resulting in PAs on lines 6-8. The parametric widening of $\text{FA}_{/\equiv}$ collapses the colon into $\top$. In Tarsis, since the automaton representing `res` grows by two states each iteration, the parametric widening defined in Sect. 4.1 can collapse the whole content of the loop into a 2-states loop recognizing $\top$`!`. The precise approximation of `res` of both domains enable the analyzer to detect that the assertion at line 6 always holds, while PAs are raised on lines 7 and 8.

In summary, Pr and Su failed to produce the expected results on both Subs and Loop, while Ci and Br produced exact results in one case (Loop and Subs, respectively), but not in the other. Hence, $\text{FA}_{/\equiv}$ and Tarsis were the two only domains that produced the desired behavior in these rather simple test cases.

5.2 | Evaluation on realistic code samples

In this section, we explore two real world code samples. Method ToString (Fig. 10a) transforms an array of names that come as string values into a single string. While it resembles the code of Loop in Fig. 9b (thus, results of all the analyses show the same strengths and weaknesses), now assertions check `contains` predicates with a multi-character string. Method CountMatches (Fig. 10b) makes use of `strings.Count` (reported in Sect. 2) to prove properties about its return value. Tab. 2 reports the results of both methods (stored in `res` and `count`, respectively) evaluated by each analysis at the first assertion, as well as if the abstract domain precisely dealt with the program assertions.

As expected, when analyzing ToString, each domain showed results similar to those of Loop. In particular, we expect to obtain no alarm at line 12 (since `"People"` is surely contained in the resulting string), and two PAs at line 13 and 14. Pr, Su, CI and BR raise PAs on all the three assert statements. FA$_{/\equiv}$ and TARSIS detect that the assertion at line 12 always holds. Thus, when using them, the analyzer raises PAs on lines 13 and 14 since (i) comma character is part of `res` if the loop is iterated at least once, and (ii) T might match `"not"`.

If COUNTMATCHES was to be executed, `count` would be either 2 or 3 when the first assertion is reached, depending on the choice of `str`. Thus, no alarm should be raised at line 9, while a DA should be raised on line 10, and a PA on line 11. Since PR, SU, CI and BR do not define most of the operations used in the code, the analyzer does not have information about the string on which `strings.Count` is executed, and thus abstract `count` with the interval $[0, +\infty]$. Thus, PAs are raised on lines 9-11. Instead, FA$_{/\equiv}$ and TARSIS are instead able to detect that `sub` is present in all the possible strings represented by `str`. Thus, thanks to trace partitioning, the trace where the loop is skipped and `count` remains 0 gets discarded. Then, when the first `indexOf` call happens, $[0, 0]$ is stored into `idx`, since all possible values of `str` start with `sub`. Since the call to `length` yields $[10, 17]$, all possible substrings from $[2, 2]$ (`idx` plus the length of `sub`) to $[10, 17]$ are computed (namely, `"e throat"`, `"is is th"`, `"is is the"`, `...`, `"is is the thing"`), and the resulting automaton is the one that recognizes all of them. Since the value of `sub` is still contained in every path of such automaton, the loop guard still holds and the second iteration is analyzed, repeating the same operations. When the loop guard is reached for the third time, the remaining substring of the shorter starting string (namely `"roat"`) recognized by the automaton representing `str` will no longer contain `sub`: a trace where `count` equals $[2, 2]$ will leave the loop. A further iteration is then analyzed, after which `sub` is no longer contained in any of the strings that `str` might hold. Thus, a second and final trace where `count` equals $[3, 3]$ will reach the assertions, and will be merged by interval lub, obtaining $[2, 3]$ as final value for `count`. This allows TARSIS and FA$_{/\equiv}$ to identify that the assertion at line 10 never holds, raising a DA, while the one at line 11 might not hold, raising a PA.

5.3 | Efficiency w.r.t. simpler string domains

The detailed analysis of two test cases, and two examples taken from real-world code underlined that TARSIS and FA$_{/\equiv}$ are the only ones able to obtain precise results on them. We now discuss the efficiency of the analyses. Tab. 3 reports the execution times for all the domains on the case studies analyzed in this section, by taking the time that GoLiSA reports as the one needed to compute a program-wide fixpoint (thus excluding the time for booting up the analysis, parsing the code and dumping the results). Note that the times reported here are higher than the ones of the original paper due to the usage of a complete static analyzer: memory abstractions and call resolution were not performed by the prototypical analyzer used in[15]. Overall, PR, SU, CI, and BR are the fastest domains with execution times usually in the order of milliseconds, with the exception of TOSTRING that proved more challenging for all domains. Thus, if on the one hand these domains failed to prove some of the properties of interest, they are quite efficient and they might be helpful to prove simple properties. TARSIS execution times are sometimes higher but still comparable with them. Instead, FA$_{/\equiv}$ blows up on three out of the four test cases (and in particular on TOSTRING). Hence, TARSIS is the only domain that executes the analysis in a limited time while being able to prove all the properties of interest on these four case studies.

The reason behind the performance gap between TARSIS and FA$_{/\equiv}$ can be accounted on the alphabets underlying the automata. In FA$_{/\equiv}$, automata are built over an alphabet of single characters. While this simplifies the semantic operations, it also causes state and transition blow up w.r.t. the size of the string that needs to be represented. This does not happen in TARSIS, since atomic strings (not built through concatenation or other string manipulations) are part of the alphabet and can be used as transition symbol. Having less states and transitions to operate upon drastically lowers the time and memory requirements of automata operations, making TARSIS faster than FA$_{/\equiv}$.

TARSIS's alphabet has another peculiarity w.r.t. FA$_{/\equiv}$'s: it has a special symbol for representing the unknown string. Having such a symbol requires some fine-tuning of the algorithms to have them behave differently when the symbol is encountered, but

Domain	**Program** TOSTRING		**Program** COUNTMATCHES	
PR	`People: {`	✗	$[0, +\infty]$	✗
SU	ϵ	✗	$[0, +\infty]$	✗
CI	`[{}:Peopl ]` `[{}:,Peopl T]`	✗	$[0, +\infty]$	✗
BR	`[{T}]` $(0, +\infty)$	✗	$[0, +\infty]$	✗
FA$_{/\equiv}$	`People: {(T)*T}`	✓	$[2, 3]$	✓
TARSIS	`People: {}` $\|$ `People: {(T,)*T}`	✓	$[2, 3]$	✓

TABLE 2 Values of `res` and `CountMatches` at the first assert of the respective program

Domain	SUBS	LOOP	TOSTRING	COUNTMATCHES
PR	5ms	28ms	1s 509ms	129ms
SU	5ms	32ms	1s 599ms	110ms
CI	4ms	45ms	1s 594ms	138ms
BR	8ms	78ms	3s 911ms	345ms
$\text{FA}_{/\equiv}$	11ms	20s 895ms	22m 37s 621ms	9s 175ms
TARSIS	7ms	443ms	12s 437ms	123ms

TABLE 3 Execution times of the domains on each program

without additional tolls on their performances. $\text{FA}_{/\equiv}$'s alphabet does not have such a symbol, thus representing the unknown string is achieved through a state having one self-loop for each character in the alphabet (including the empty string). This requires significantly more resources for automata algorithms, leading to higher execution times.

It is important to notice that performances of programs relying on automata are heavily dependent on their implementation. Both $\text{FA}_{/\equiv}$ and TARSIS come as non-optimized implementations whose performances can be greatly improved, thus further reducing the gap between them and the simpler string abstractions. The source code of $\text{FA}_{/\equiv}$ is available at https://github.com/SPY-Lab/fsa, while TARSIS's implementation is published at https://github.com/UniVE-SSV/tarsis. Instead, implementations used in the experiments are part of LiSA available at https://github.com/lisa-analyzer/lisa.

5.4 | Performance benchmark of TARSIS and $\text{FA}_{/\equiv}$

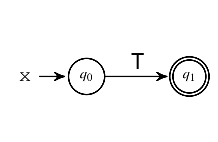
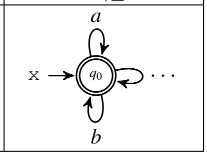
As automata-based abstractions have already been proved to be effective in string program analysis [10], we focus the final part of our evaluation on the difference in resource consumption between TARSIS and $\text{FA}_{/\equiv}$. In fact, our main goal is proving that the adoption of TARSIS can make automata-based abstractions viable for the analysis of non-trivial code.

To measure the performance of TARSIS and $\text{FA}_{/\equiv}$, one could track the resource consumption of full program analyses employing the two domains and reason about their difference. However, this could produce misleading results: the measured performance would be affected by the ones of other analysis components (e.g., memory abstractions, interprocedural analyses), and would hence be affected by their running time and memory consumption. The different semantics of TARSIS and $\text{FA}_{/\equiv}$ would thus only account for a small portion of each measurement, hindering the purpose of our experiments. Hence, to ensure that we can precisely measure only the performance of interest, i.e., the ones concerning TARSIS and $\text{FA}_{/\equiv}$, we directly compare each lattice operation and abstract transformer in isolation, benchmarking their execution times. This enables accurate speedup measurements, since such times are not affected by external factors like memory saturation caused by the remaining analysis components.

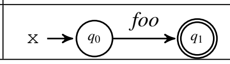
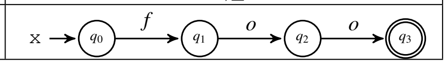
We compare lattice operators $\sqsubseteq$, $\sqcup$, $\sqcap$, and ∇, together with the semantics of string operators (`concat`, `contains`, `length`, `substr`, `trim`, `repeat`, and `indexOf`); since `trim`'s semantics relies on `trimLeft` and `trimRight`, the comparison of the latter operations have been omitted. Moreover, concerning the `trimLeft`, `trimRight`, and `trim` abstract semantics of $\text{FA}_{/\equiv}$, we adopted the same one reported in Sect. 4.2 for TARSIS, recasted to standard automata, since a bug was found in the original $\text{FA}_{/\equiv}$ abstract semantics when automata with cycles were involved, leading to unsound results. For instance, let us consider the automaton recognizing the language $\{ (\llcorner a \llcorner)^n \mid n \in \mathbb{N} \}$, the original `trim` abstract semantics of $\text{FA}_{/\equiv}$ returns the automaton recognizing the language $\{ a^n \mid n \in \mathbb{N} \}$, that is an unsound result (e.g., the string $a\llcorner\llcorner a$ is not recognized by the resulting automaton).

During an actual program analysis, the target operators would be invoked by a fixpoint engine on operands built through several string manipulations. Each such operand would thus be an automaton with an arbitrarily complex structure, depending on the combination of operators used. To ensure that our experiment measures the performance of each domain's operands on all possible input structures, we identify and define here 7 automata classes, each containing automata sharing a common structure, on which lattice operators and string operations can be applied to. Then, we compose a benchmark by using automata from all classes, thus ensuring that the measurements will take into account performance on different automata structures. In the following, we describe each class reporting an exemplification of a Go fragment that may generate an automaton belonging to each class (where `b` is a statically unknown Boolean value), together with the automaton computed by TARSIS and $\text{FA}_{/\equiv}$ for the string variable `x` at the end of the fragment.

1. Statically unknown strings. Such class contains the abstractions of statically unknown strings, such as an user input. Since both domains have a unique minimum automaton for such strings, this class is made of a single automaton.

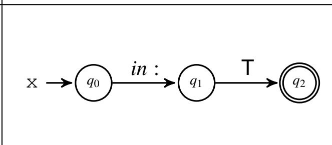
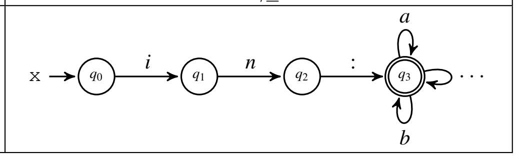
Go fragment	TARSIS	FA$_{/\equiv}$
`_, err := fmt.Scanf("%s", &x);`		

2. *Constant strings.* This class contains automata generated as abstractions of string literals appearing in the program.

Go fragment	TARSIS	FA$_{/\equiv}$
`x := "foo";`		

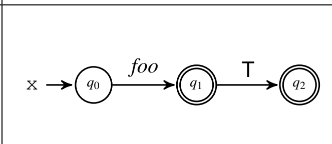
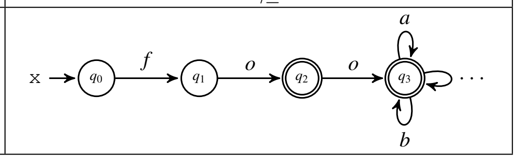
3. *Concatenated strings.* This class contains automata modeling the concatenation of simple strings, with each being either statically unknown or constant. Automata in this class are effective concatenations of ones from the previous classes.

Go fragment	TARSIS	FA$_{/\equiv}$

`z := "in:";`
`_, err := fmt.Scanf("%s", &y);`
`x := z + y;`
		

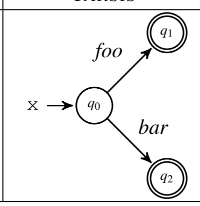
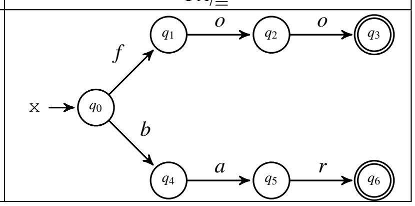
4. *Increasing strings.* The automata in this class model sets of strings built by optionally appending a finite number of strings at the end of an existing one, thus leading to single path automata. Each of such strings can be either statically unknown or constant (i.e., each automaton is part of one of the first two classes).

Go fragment	TARSIS	FA$_{/\equiv}$

`x := "foo";`
`_, err := fmt.Scanf("%s", &y);`
`if b {`
`  x := x + y;`
`}`
		

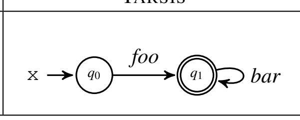
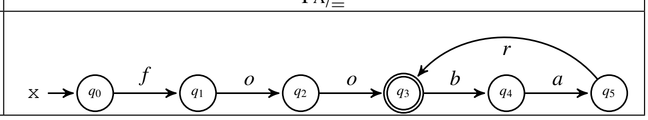
5. *Disjoint strings.* This class contains automata built as the union of up to four different automata coming from the third class, thus modeling their least upper bound. Automata in this class represent the result of merging branches where string variables were assigned to different values. Each string used in the lub can thus be either statically unknown, constant, or resulting from a concatenation.

Go fragment	TARSIS	FA$_{/\equiv}$

`if b {`
`  x := "foo";`
`} else {`
`  x := "bar";`
`}`
		

6. *Looping strings.* The automata in this class are built by inserting up to two loops in the automata of the third class, thus modeling strings that are built inside a loop.

Go fragment	TARSIS	FA$_{/\equiv}$

`x := "foo";`
`for b {`
`  x := x + "bar";`
`}`
		

7. *Random strings.* This class contains automata that are built using an arbitrary set of manipulations, and thus have no predefined structure. They represent worst-case scenarios for both domains, and thus automata of this class can be significantly complex. As this class does not follow any particular structure, examples automata with their generating code are omitted.

Operation	Domain	# Successes	# Timeouts	Total Time	Avg. Time	Min. Time	Max. Time	Time ratio
$\sqsubseteq$	TARSIS	100	0	2s 37ms	20ms	< 1ms	403ms	920.74x
	$\mathrm{FA}_{/\equiv}$	88	12	2m 27s 815ms	1s 679ms	< 1ms	18s 685ms	
$\sqcup$	TARSIS	100	0	29ms	< 1ms	< 1ms	10ms	114.98x
	$\mathrm{FA}_{/\equiv}$	100	0	3s 430ms	34ms	< 1ms	945ms 31ms	
$\sqcap$	TARSIS	100	0	4s 905ms	< 1ms	< 1ms	8ms	29.37x
	$\mathrm{FA}_{/\equiv}$	100	0	760ms	7ms	< 1ms	426ms	
∇	TARSIS	100	0	148ms	1ms	< 1ms	27ms	97.70x
	$\mathrm{FA}_{/\equiv}$	96	4	13s 485ms	140ms	< 1ms	3s 522ms	
`concat`	TARSIS	100	0	40ms	< 1ms	< 1ms	5ms	528.42x
	$\mathrm{FA}_{/\equiv}$	100	0	21s 537ms	215ms	< 1ms	14s 177ms	
`contains`	TARSIS	100	0	2s 187ms	21ms	< 1ms	668ms	7.16x
	$\mathrm{FA}_{/\equiv}$	99	1	10s 888ms	109ms	< 1ms	1s 922ms	
`length`	TARSIS	100	0	143ms	1ms	< 1ms	36ms	2206.42x
	$\mathrm{FA}_{/\equiv}$	74	26	1m 19s 427ms	1s 73ms	< 1ms	15s 454ms	
`indexOf`	TARSIS	100	0	6s 575ms	65ms	< 1ms	2s 222ms	1.72x
	$\mathrm{FA}_{/\equiv}$	100	0	11s 352ms	113ms	< 1ms	10s 693ms	
`substr`	TARSIS	100	0	22s 757ms	227ms	< 1ms	13s 807ms	465.37x
	$\mathrm{FA}_{/\equiv}$	53	47	3m 49s 127ms	4s 323ms	< 1ms	32s 998ms	
`replace`	TARSIS	99	1	326ms	3ms	< 1ms	47ms	7.98x
	$\mathrm{FA}_{/\equiv}$	99	1	2s 440ms	24ms	< 1ms	1s 470ms	
`trim`	TARSIS	99	1	20s 759ms	209ms	< 1ms	18s 270ms	486.42x
	$\mathrm{FA}_{/\equiv}$	57	43	1m 38s 269ms	1s 724ms	< 1ms	22s 760ms	
`repeat`	TARSIS	100	0	37ms	209ms	< 1ms	895ms	3.93x
	$\mathrm{FA}_{/\equiv}$	97	3	12s 488ms	128ms	< 1ms	2s 290ms	

TABLE 4 Lattice operators and abstract transformers benchmark results

Benchmark composition

We benchmark each lattice and string operators mentioned above by executing them 100 times (with each execution referred to as a *round*), each one using randomly generated automata from one of the above classes for all the required inputs (e.g., a round executing `concat` on automata of class 2 will generate 2 automata $\mathtt{A}_1$ and $\mathtt{A}_2$ from that class, and would then run $\mathtt{concat}(\mathtt{A}_1, \mathtt{A}_2)$). Specifically, 1 round is executed using the single automaton from class 1, 5 rounds with automata from class 2, 10 rounds are executed for classes 3 to 6, while the remaining 54 rounds use automata from class 7. Each round consists of (i) the generation of the TARSIS automata, (ii) their conversion to the equivalent $\mathrm{FA}_{/\equiv}$ ones, and (iii) the execution of each operation individually, measuring the required time. Each operation's execution has a 30 seconds timeout, as we deem it a reasonable time bound for an operation to complete when running as part of an analysis. Before executing the benchmark, a warm-up iteration is executed to ensure that setup operations of the JVM would not impact the actual measurement. Finally, integer indexes for the substring operation were randomly chosen between 0 and 20.

As the size of $\mathrm{FA}_{/\equiv}$ automata grows fast, we impose some limits on the structure of each generated TARSIS automaton, such that, when we generate the equivalent $\mathrm{FA}_{/\equiv}$ automaton, its size is limited. This prevents an excessive amount of timeouts that would invalidate our experiments. Specifically:

- all atomic strings (i.e., the ones used as symbols in TARSIS's transitions) have a random length of up to 10 characters;
- each atomic string has a 10% chance of being statically unknown;
- intermediate states of single path automata (class 4) have a 50% chance of being an additional accepting state;
- automata of class 7 can contain up to 5 states with up to 3 transitions per state; each state has a 25% chance of being an additional final state.

Benchmark results

Tab. 4 reports, for each operation and domain, the number of successful rounds and the one of timed-out ones, together with the total, average, minimum, and maximum execution times of the successful rounds. Finally, the speedup of TARSIS w.r.t. $\mathrm{FA}_{/\equiv}$ is determined by only comparing execution times of the rounds where both domains ran successfully, reporting the ratio $t_{\mathrm{FA}_{/\equiv}}/t_{\mathrm{TARSIS}}$.

The benchmark confirms that TARSIS is overall reliably faster than $\mathrm{FA}_{/\equiv}$. TARSIS operations time out significantly less than the ones of $\mathrm{FA}_{/\equiv}$ (2 timeouts instead of 137), reporting lower total times despite the higher number of automata tackled. Worst-case performances are also improved, with most operations requiring far less time with TARSIS even on complex automata, as highlighted by column *Max. Time* of Tab. 4. Moreover, when focusing only on rounds where both domains succeeded within the required time bound, the speedup (column *Time ratio* of Tab. 4) is still noticeable: even in the worst case of `indexOf`, TARSIS still performs almost twice as fast as $\mathrm{FA}_{/\equiv}$.

Let us comment the results concerning `length` and `trim`. For these operations, TARSIS and $\mathrm{FA}_{/\equiv}$ use the same algorithms to implement the corresponding abstract semantics. Still, TARSIS incurs in a single timeout when executing `trim` (against 43 timeouts of $\mathrm{FA}_{/\equiv}$) and in none when executing `length` (while $\mathrm{FA}_{/\equiv}$ incurs in 26 timeouts), also enabling significant speedup (more than 2000x for `length` and almost 500x for `trim`). This highlights that even when using the same abstract semantics for both domains, the alphabet chosen for TARSIS is still able to drastically reduce the required resources that a static analysis needs to analyze strings.

6 | RELATED WORK

The problem of statically analyzing strings has been already tackled in different contexts in the literature during the last two decades[11,10,27,28,29,30,9]. As already discussed, the TARSIS abstract domain build upon the finite state automata abstract domain defined in[10] in the context of dynamic languages, providing automata-based abstract semantics for common ECMAScript string operations. The same abstract domain has been integrated also for defining a sound-by-construction analysis for string-to-code statements[31]. As reported by the experimental results in Sect. 5 (Tab. 3 in particular), TARSIS is quite more efficient than $\mathrm{FA}_{/\equiv}$ while keeping (almost) the same level of precision.

Generally speaking, the practical comparison of different string static analyses is particularly challenging because of (i) the lack of standard benchmarks, (ii) the variety of the information that can be tracked over string values (e.g., included characters, their order, regular expressions, substring relations, ...), and (iii) the availability of the implementations of existing analyses (often formalized several years ago and not actively maintained). For these reasons, in the rest of this section we qualitatively discuss the differences between TARSIS and other related works, but we do not experimentally compare them.

Static analysis of string values: Our experimental results compared TARSIS with several simple abstract domains introduced in[32,9]. Those domains track relatively simple information about string values, such as characters included or not, prefixes and suffixes, and concatenation of constant string values. As reported in Tabs. 1, 2, and 3, these abstractions are quite more efficient but less precise than TARSIS. Overall, TARSIS exposed execution times comparable to the ones of[32,9], and precision similar to[10,31].

Static analysis was applied to string values in many different contexts, such as SQL queries programmatically built by code[3], reflection[4,5], and injection vulnerabilities[7,8]. Instead, the main goal of our work is to introduce a novel abstract domain (that is, approximation of string values) that outperforms state-of-the-art approaches in terms of precision or efficiency, and this can be applied to different contexts.

Automata-based approaches: Several approaches to statically analyze string values through automata (like TARSIS) have been proposed in the literature. For instance, the authors of[33] provided an automata abstraction merged with interval abstractions for analyzing JavaScript arrays and objects. Another interesting automata-based model is symbolic automata[34], which differs from the standard one having an alphabet of predicates (that can potentially be infinite) instead of single characters. Examples of applications of symbolic automata in the context of static analysis are regex processing, sanitizer analysis[35], and their usage as program models for mixing syntactic and semantic abstractions over the program[36]. Generally speaking, automata-based approaches are usually not efficient, since they need to perform algorithmically complex operations on the automata. Our approach is aimed at (partially) overcoming such limits by working on an alphabet of strings instead of single characters.

Regular expressions: A major stream of research focussed on the abstraction of string values through regular expressions. In particular, in[28], the authors proposed a static analysis of Java strings based on the abstraction of the control-flow graph as a context-free grammar. By relying on regular expressions, this approach is in position to track information not only on constants inside the string values like TARSIS, but also to approximate ranges of possible characters such as `0|(-?[1-9][0-9]*)` (that is the string representation of integer values). However, such approximation is strictly more complex and led to less efficient analyses.

Similarly, regular strings[37] is an abstraction of the finite state automata domain and approximates strings as a strict subset of regular expressions. The authors introduced an aggressive widening operator, that improves the efficiency but worsens the precision of the analysis.

String constraints verification: Another major research effort was spent on the context of string constraints verification by the investigation and development of various techniques and tools focused on the study of decidable fragments of string constraint formulas[38] or proposing new efficient decidable procedures or string constraints representations[39,40] also based on automata, such as[41,42], or involving type conversion string constraints[43]. For instance, Z3-str[44] extended the SMT-solver Z3[45] by treating strings as primitive types supporting the most common operations over strings (e.g., concatenation and substring). All those approaches allow to track very precise information over string values but usually require manually annotating some portions of the code (e.g., through loop invariants and pre- and post-conditions), and require solving NP-complete problems (e.g., SAT solving) causing slow-downs (or timeouts) of the analyses in some (hopefully corner) cases.

7 | CONCLUSION

In this paper we introduced TARSIS, an abstract domain for sound abstraction of string values. TARSIS is based on finite state automata paired with their equivalent regular expression: a representation that allows precise modeling of complex string values. Experiments show that TARSIS achieves great precision also on code that heavily manipulate string values, while the time needed for the analysis is comparable with the one of other simpler domains.

In order to enforce loop convergence, our analysis has been equipped with a widening with threshold operator over TARSIS automata. As it usually happens in abstract interpretation, in order to retrieve some information lost by the widening application, the analysis can be equipped also with a narrowing[23]. Hence, a narrowing operator for TARSIS will be studied, in order to get more precise results on loops analyses.

Finally, TARSIS provides a non-relational string abstraction: as such, information relating different variables is not modeled inside the domain. We thus intend on working on a relational extension of TARSIS that is also able to relate different variables, also extending existing solutions based on combining TARSIS with other domains[46].

Acknowledgements.
Work partially supported by Bando di Ateneo per la Ricerca 2022, funded by University of Parma, (MUR_DM737_2022_FIL_PROGETTI_B_ARCERI_COFIN, CUP: D91B21005370003), "Formal verification of GPLs blockchain smart contracts", SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU, iNEST-Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 49 1.5) NextGeneration EU - Project ID: ECS00000043, and SPIN-2021 "Static Analysis for Data Scientists" funded by Ca' Foscari University.

REFERENCES

1. Cousot P, Cousot R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Graham RM, Harrison MA, Sethi R., eds. *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*ACM 1977:238–252
2. Cousot P, Cousot R. Systematic Design of Program Analysis Frameworks. In: Aho AV, Zilles SN, Rosen BK., eds. *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*ACM Press 1979:269–282
3. Gould C, Su Z, Devanbu PT. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In: Finkelstein A, Estublier J, Rosenblum DS., eds. *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*IEEE Computer Society 2004:697–698
4. Landman D, Serebrenik A, Vinju JJ. Challenges for static analysis of Java reflection: literature review and empirical study. In: Uchitel S, Orso A, Robillard MP., eds. *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*IEEE / ACM 2017:507–518
5. Bodden E, Sewe A, Sinschek J, Oueslati H, Mezini M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Taylor RN, Gall HC, Medvidovic N., eds. *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*ACM 2011:241–250
6. Arceri V, Mastroeni I. A sound abstract interpreter for dynamic code. In: Hung C, Cerný T, Shin D, Bechini A., eds. *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*ACM 2020:1979–1988
7. Fu X, Lu X, Peltsverger B, Chen S, Qian K, Tao L. A Static Analysis Framework For Detecting SQL Injection Vulnerabilities. In: IEEE Computer Society 2007:87–96
8. Livshits VB, Lam MS. Finding Security Vulnerabilities in Java Applications with Static Analysis. In: McDaniel PD., ed. *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*USENIX Association 2005.
9. Costantini G, Ferrara P, Cortesi A. A suite of abstract domains for static analysis of string values. *Softw. Pract. Exp.*. 2015;45(2):245–287. doi: 10.1002/SPE.2218
10. Arceri V, Mastroeni I, Xu S. Static Analysis for ECMAScript String Manipulation Programs. *Appl. Sci.*. 2020;10:3525. doi: 10.3390/app10103525
11. Cortesi A, Olliaro M. M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs. In: Pang J, Zhang C, He J, Weng J., eds. *2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018*IEEE Computer Society 2018:1–8
12. Olivieri L, Negrini L, Arceri V, et al. Information Flow Analysis for Detecting Non-Determinism in Blockchain. In: Ali K, Salvaneschi G., eds. *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*. 263 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik 2023:23:1–23:25

13. Ferrara P, Negrini L, Arceri V, Cortesi A. Static analysis for dummies: experiencing LiSA. In: Do LNQ, 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*ACM 2021:1–6
14. Negrini L, Ferrara P, Arceri V, Cortesi A. *LiSA: A Generic Framework for Multilanguage Static Analysis*:19–42; Singapore: Springer Nature Singapore . 2023
15. Negrini L, Arceri V, Ferrara P, Cortesi A. Twinning Automata and Regular Expressions for String Static Analysis. In: Henglein F, Shoham S, Vizel Y., eds. *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*. 12597 of *Lecture Notes in Computer Science*. Springer 2021:267–290
16. Giacobazzi R, Ranzato F, Scozzari F. Making abstract interpretations complete. *J. ACM*. 2000;47(2):361–416. doi: 10.1145/333979.333989
17. Bartzis C, Bultan T. Widening Arithmetic Automata. In: Alur R, Peled DA., eds. *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. 3114 of *Lecture Notes in Computer Science*. Springer 2004:321–333
18. D'Silva V. Widening for Automata. MsC Thesis, Inst. Fur Inform. - UZH; 2006.
19. Cousot P, Cousot R. Abstract Interpretation Frameworks. *J. Log. Comput.*. 1992;2(4):511–547. doi: 10.1093/LOGCOM/2.4.511
20. Bagnara R, Hill PM, 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.*. 2008;72(1-2):3–21. doi: 10.1016/J.SCICO.2007.08.001
21. Becchi A, Zaffanella E. PPLite: Zero-overhead encoding of NNC polyhedra. *Inf. Comput.*. 2020;275:104620. doi: 10.1016/J.IC.2020.104620
22. Davis MD, Sigal R, Weyuker EJ. *Computability, Complexity, and Languages: Fund. of Theor. CS*. Academic Press Professional, Inc., 1994.
23. Cortesi A, Zanioli M. Widening and narrowing operators for abstract interpretation. *Comput. Lang. Syst. Struct.*. 2011;37(1):24–42. doi: 10.1016/J.CL.2010.09.001
24. Arceri V, Maffeis S. Abstract Domains for Type Juggling. *Electron. Notes Theor. Comput. Sci.*. 2017;331:41–55. doi: 10.1016/j.entcs.2017.02.003
25. Arceri V, Olliaro M, Cortesi A, Mastroeni I. Completeness of Abstract Domains for String Analysis of JavaScript Programs. In: Hierons RM, Mosbah M., eds. *Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings*. 11884 of *Lecture Notes in Computer Science*. Springer 2019:255–272
26. Rival X, Mauborgne L. The Trace Partitioning Abstract Domain. *ACM Trans. Program. Lang. Syst.*. 2007;29(5):26–es. doi: 10.1145/1275497.1275501
27. Park C, Im H, Ryu S. Precise and scalable static analysis of jQuery using a regular expression domain. In: Ierusalimschy R., ed. *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*ACM 2016:25–36
28. Christensen AS, Møller A, Schwartzbach MI. Precise Analysis of String Expressions. In: Cousot R., ed. *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. 2694 of *Lecture Notes in Computer Science*. Springer 2003:1–18
29. Madsen M, Andreasen E. String Analysis for Dynamic Field Access. In: Cohen A., ed. *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. 8409 of *Lecture Notes in Computer Science*. Springer 2014:197–217
30. Abdulla PA, Atig MF, Chen Y, et al. String Constraints for Verification. In: Biere A, Bloem R., eds. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 8559 of *Lecture Notes in Computer Science*. Springer 2014:150–166
31. Arceri V, Mastroeni I. Analyzing Dynamic Code: A Sound Abstract Interpreter for *Evil* Eval. *ACM Trans. Priv. Secur.*. 2021;24(2):10:1–10:38. doi: 10.1145/3426470
32. Costantini G, Ferrara P, Cortesi A. Static Analysis of String Values. In: Qin S, Qiu Z., eds. *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*. 6991 of *Lecture Notes in Computer Science*. Springer 2011:505–521
33. Almashfi N, Lu L. Precise String Domain for Analyzing JavaScript Arrays and Objects. In: May Huang SZ., ed. *3rd International Conference on Information and Computer Technologies, ICICT 2020, San Jose, CA, USA, March 9-12, 2020*IEEE 2020:17–23
34. D'Antoni L, Veanes M. Minimization of symbolic automata. In: Jagannathan S, Sewell P., eds. *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*ACM 2014:541–554
35. Veanes M. Applications of Symbolic Finite Automata. In: Konstantinidis S., ed. *Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings*. 7982 of *Lecture Notes in Computer Science*. Springer 2013:16–23
36. Preda MD, Giacobazzi R, Lakhotia A, Mastroeni I. Abstract Symbolic Automata: Mixed syntactic/semantic similarity analysis of executables. In: Rajamani SK, Walker D., eds. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*ACM 2015:329–341
37. Choi T, Lee O, Kim H, Doh K. A Practical String Analyzer by the Widening Approach. In: Kobayashi N., ed. *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*. 4279 of *Lecture Notes in Computer Science*. Springer 2006:374–388
38. Abdulla PA, Atig MF, Diep BP, Holík L, Janku P. Chain-Free String Constraints. In: Chen Y, Cheng C, Esparza J., eds. *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. 11781 of *Lecture Notes in Computer Science*. Springer 2019:277–293
39. Chen T, Hague M, Lin AW, Rümmer P, Wu Z. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*. 2019;3(POPL):49:1–49:30. doi: 10.1145/3290362
40. Amadini R, Gange G, Stuckey PJ. Dashed strings for string constraint solving. *Artif. Intell.*. 2020;289:103368. doi: 10.1016/J.ARTINT.2020.103368
41. Wang H, Chen S, Yu F, Jiang JR. A symbolic model checking approach to the analysis of string and length constraints. In: Huchard M, Kästner C, Fraser G., eds. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*ACM 2018:623–633
42. Yu F, Alkhalaf M, Bultan T, Ibarra OH. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods Syst. Des.*. 2014;44(1):44–70. doi: 10.1007/s10703-013-0189-1
43. Abdulla PA, Atig MF, Chen Y, et al. Efficient handling of string-number conversion. In: Donaldson AF, Torlak E., eds. *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*ACM 2020:943–957

44. Zheng Y, Zhang X, Ganesh V. Z3-str: a z3-based string solver for web application analysis. In: Meyer B, Baresi L, Mezini M., eds. *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*ACM 2013:114–124
45. Moura dLM, Bjørner NS. Z3: An Efficient SMT Solver. In: Ramakrishnan CR, Rehof J., eds. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 4963 of *Lecture Notes in Computer Science*. Springer 2008:337–340
46. Arceri V, Olliaro M, Cortesi A, Ferrara P. Relational String Abstract Domains. In: Finkbeiner B, Wies T., eds. *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*. 13182 of *Lecture Notes in Computer Science*. Springer 2022:20–42

□

APPENDIX

A SOUNDNESS AND COMPLETENESS PROOFS OF TARSIS'S SEMANTICS

We prove the soundness and completeness of TARSIS's abstract semantics by showing that their concretization is an over-approximation of the concrete one. As we formalized our transfer functions w.r.t. the smashed sum $\text{VAL}^\sharp \triangleq \mathcal{T}\text{FA}_{/\equiv} \oplus \mathsf{Intv} \oplus \mathsf{Bool}$, we compare concretizations of its elements with a concrete smashed sum $\overline{\text{VAL}} \triangleq \wp(\Sigma^*) \cup \wp(\mathbb{Z}) \cup \wp(\{\texttt{true}, \texttt{false}\})$, that is defined as a collecting semantics. We abuse notation denoting with $\mathbb{M} : \text{ID} \to \overline{\text{VAL}}$ the set of collecting memories, ranging over meta-variable $\mathbb{m}$, that associate each identifier to a collecting value. The concrete expression semantics of such domain is defined as $[\![\, \mathsf{e} \,]\!] : \mathbb{M} \to \overline{\text{VAL}}$, evaluating e and returning its possible values. Such semantics is defined as the additive lift of the one in Fig. 3. Function $\gamma_{\text{VAL}^\sharp} : \text{VAL}^\sharp \to \overline{\text{VAL}}$ is the smashed sum concretization and it is defined as:

$$\gamma_{\text{VAL}^\sharp}(a) \triangleq \begin{cases} \varnothing & \text{if } a = \bot \\ \gamma_{\mathsf{Intv}}(a) & \text{if } a \in \mathsf{Intv} \\ \gamma_{\mathsf{Bool}}(a) & \text{if } a \in \mathsf{Bool} \\ \gamma_{\mathcal{T}}(a) & \text{if } a \in \mathcal{T}\text{FA}_{/\equiv} \\ \overline{\text{VAL}} & \text{otherwise} \end{cases}$$

where $\gamma_{\mathsf{Intv}} : \mathsf{Intv} \to \wp(\mathbb{Z})$ and $\gamma_{\mathsf{Bool}} : \mathsf{Bool} \to \wp(\{\texttt{true}, \texttt{false}\})$ correspond to the concretization functions of intervals and Booleans, respectively. We can now define the abstract memories concretization $\gamma : \mathbb{M}^\sharp \to \mathbb{M}$ as $\gamma(\mathbb{m}^\sharp) \triangleq \{ \ (x, \gamma_{\text{VAL}^\sharp}(\mathbb{m}^\sharp(x))) \ \mid \ x \in dom(\mathbb{m}^\sharp) \ \}$. With this setup, we prove the abstract semantics to be sound by proving that $\forall \mathbb{m}^\sharp \in \mathbb{M}^\sharp . [\![\, \mathsf{e} \,]\!]\gamma(\mathbb{m}^\sharp) \subseteq \gamma_{\text{VAL}^\sharp}([\![\, \mathsf{e} \,]\!]^\sharp(\mathbb{m}^\sharp))$. We also prove completeness by enforcing the equality on such relation, and incompleteness by providing a counterexample.

In the following, we remove the subscript from γ to avoid cluttering the notation, since it is clear from the context which concretization function applies. Moreover, we mark proof steps as *automata lift* if they represent the transition from a condition over languages (that is, sets of strings) to its equivalent condition over automata. Finally, given the non-existence of a GC between the string concrete domain and TARSIS, from here on, when we refer to completeness, we intend forward completeness, defined in Sect. 3.

A.1 Concat

Theorem 1. *$[\![\, \texttt{concat}\,(\mathsf{s}, \mathsf{s}')\,]\!]^\sharp$ is a sound and complete abstraction of $[\![\, \texttt{concat}\,(\mathsf{s}, \mathsf{s}')\,]\!]$. Formally:*

$$\forall \mathbb{m}^\sharp \in \mathbb{M}^\sharp, \forall \mathsf{s}, \mathsf{s}' \in \text{SE} . [\![\, \texttt{concat}\,(\mathsf{s}, \mathsf{s}')\,]\!]\gamma(\mathbb{m}^\sharp) = \gamma([\![\, \texttt{concat}\,(\mathsf{s}, \mathsf{s}')\,]\!]^\sharp \mathbb{m}^\sharp)$$

Proof. Soundness and completeness follow from the fact that finite state automata and regular languages are closed under finite concatenation[22]. □

A.2 Length

Theorem 2. $[\![\ \mathit{length}\,(\mathsf{s})\ ]\!]^\sharp$ *is a sound but not complete abstraction of* $[\![\ \mathit{length}\,(\mathsf{s})\ ]\!]$. *Formally:*

$$\forall \mathfrak{m}^\sharp \in \mathbb{M}^\sharp, \forall \mathsf{s} \in \mathrm{SE}\,.\, [\![\ \mathit{length}\,(\mathsf{s})\ ]\!]\gamma(\mathfrak{m}^\sharp) \subsetneq \gamma([\![\ \mathit{length}\,(\mathsf{s})\ ]\!]^\sharp \mathfrak{m}^\sharp)$$

Proof. The collecting semantics of $\mathtt{length}$ is defined as the additive lift of the concrete one reported in Fig. 3, namely $[\![\ \mathtt{length}(\mathsf{s})\ ]\!]\mathfrak{m} = \{\ |\sigma| \mid \sigma \in \mathscr{L}\ \}$, where $[\![\ \mathsf{s}\ ]\!]\mathfrak{m} = \mathscr{L} \in \wp(\Sigma^*)$. Let us suppose that $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \mathscr{L} \in \wp(\Sigma^*)$, and let $I = \{\ |\sigma| \mid \sigma \in \mathscr{L}\ \}$. Following the semantics definition, if $\mathsf{cyclic}(\mathtt{A}) \vee \mathsf{readsTop}(\mathtt{A})$, we prove the soundness as:

$$\begin{aligned}
& [\![\ \mathtt{length}(\mathsf{s})\ ]\!]\gamma(\mathfrak{m}^\sharp) & \\
=\,& I & \wr \text{def. } \mathbb{E} \wr \\
\subseteq\,& \gamma([\min I, +\infty]) & \wr \text{def. } \min, \gamma \wr \\
=\,& \gamma([|\mathsf{minPath}(\mathtt{A})|, +\infty]) & \wr \text{def. } \mathsf{minPath} \wr \\
=\,& \gamma([\![\ \mathtt{length}(\mathsf{s})\ ]\!]^\sharp \mathfrak{m}^\sharp) & \wr \text{def. } \mathbb{E}^\#, 1^{st} \text{ case} \wr
\end{aligned}$$

Otherwise, since $\mathscr{L}$ is a finite language, soundness is proven as:

$$\begin{aligned}
& [\![\ \mathtt{length}(\mathsf{s})\ ]\!]\gamma(\mathfrak{m}^\sharp) & \\
=\,& I & \wr \text{def. } \mathbb{E} \wr \\
\subseteq\,& \gamma([\min I, \max I]) & \wr \text{def. } \max, \gamma \wr \\
=\,& \gamma([|\mathsf{minPath}(\mathtt{A})|, |\mathsf{maxPath}(\mathtt{A})|]) & \wr \text{def. } \mathsf{minPath}, \mathsf{maxPath} \wr \\
=\,& \gamma([\![\ \mathtt{length}(\mathsf{s})\ ]\!]^\sharp \mathfrak{m}^\sharp) & \wr \text{def. } \mathbb{E}^\#, 2^{nd} \text{ case} \wr
\end{aligned}$$

As a counterexample for completeness, let us consider $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \{a, aaa\}$.

$$[\![\ \mathtt{length}(\mathsf{s})\ ]\!]\gamma(\mathfrak{m}^\sharp) = \{1, 3\} \subsetneq \gamma([1, 3]) = \gamma([\![\ \mathtt{length}(\mathsf{s})\ ]\!]^\sharp \mathfrak{m}^\sharp)$$

□

A.3 Contains

Theorem 3. $[\![\ \mathit{contains}\,(\mathsf{s}, \mathsf{s}')\ ]\!]^\sharp$ *is a sound but not complete abstraction of* $[\![\ \mathit{contains}\,(\mathsf{s}, \mathsf{s}')\ ]\!]$. *Formally:*

$$\forall \mathfrak{m}^\sharp \in \mathbb{M}^\sharp, \forall \mathsf{s}, \mathsf{s}' \in \mathrm{SE}\,.\, [\![\ \mathit{contains}\,(\mathsf{s}, \mathsf{s}')\ ]\!]\gamma(\mathfrak{m}^\sharp) \subseteq \gamma([\![\ \mathit{contains}\,(\mathsf{s}, \mathsf{s}')\ ]\!]^\sharp \mathfrak{m}^\sharp)$$

Proof. The collecting semantics of $\mathtt{contains}$ is defined as the additive lift of the concrete one, that is $[\![\ \mathtt{contains}(\mathsf{s}, \mathsf{s}')\ ]\!]\mathfrak{m} = \{\ b \mid b = \mathtt{contains}(\sigma, \sigma'), \sigma \in \mathscr{L}, \sigma' \in \mathscr{L}'\ \}$, where $[\![\ \mathsf{s}\ ]\!]\mathfrak{m} = \mathscr{L} \in \wp(\Sigma^*)$, $[\![\ \mathsf{s}'\ ]\!]\mathfrak{m} = \mathscr{L}' \in \wp(\Sigma^*)$ and $\mathtt{contains} : \Sigma^* \times \Sigma^* \to \{\mathtt{true}, \mathtt{false}\}$ corresponds to the concrete semantics of Fig. 3. Let $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \mathscr{L} \in \wp(\Sigma^*)$ and $[\![\ \mathsf{s}'\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A}' \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}') = \mathscr{L}' \in \wp(\Sigma^*)$. We split the proof following possible values produced by the concrete semantics.
$\triangleright$ $[\![\ \mathtt{contains}(\mathsf{s}, \mathsf{s}')\ ]\!]\gamma(\mathfrak{m}^\sharp) = \{\mathtt{false}\}$. Thus, no substring of the strings in $\mathscr{L}$ is in $\mathscr{L}'$.

$$\begin{aligned}
& [\![\ \mathtt{contains}(\mathsf{s}, \mathsf{s}')\ ]\!]\gamma(\mathfrak{m}^\sharp) = \{\mathtt{false}\} \xLeftrightarrow{\text{def. } \mathbb{E}} \forall \sigma \in \mathscr{L}\ \forall \sigma' \in \mathscr{L}'\,.\, \sigma' \not\curlywedge_{\mathsf{s}} \sigma \xLeftrightarrow{\text{def. } \mathsf{FA}} \mathscr{L}(\mathsf{FA}(\mathtt{A})) \cap \mathscr{L}' = \varnothing \\
& \xLeftrightarrow{\text{automata lift}} \mathsf{FA}(\mathtt{A}) \sqcap_{\mathcal{T}} \mathtt{A}' = \mathsf{Min}(\varnothing) \xLeftrightarrow{\text{def. } \mathbb{E}^\#, 1^{st} \text{ case}} [\![\ \mathtt{contains}(\mathsf{s}, \mathsf{s}')\ ]\!]^\sharp \mathfrak{m}^\sharp = \{\mathtt{false}\}
\end{aligned}$$

$\triangleright$ $[\![\ \texttt{contains(s,s')}\ ]\!]\gamma(\mathtt{m}^\sharp) = \{\texttt{true}\}$. Thus, all strings in $\mathscr{L}$ contain all the strings of $\mathscr{L}'$. This invalidates the first case of our abstract semantics, as $\exists\sigma \in \mathscr{L}(\mathsf{FA}(\mathtt{A}))\,.\,\sigma \in \mathscr{L}'$. If $\mathsf{singlePath}(\mathtt{A}')$ holds, our semantics matches the concrete one:

$$[\![\ \texttt{contains(s,s')}\ ]\!]\gamma(\mathtt{m}^\sharp) = \{\texttt{true}\} \land \mathsf{singlePath}(\mathtt{A}') \xLeftrightarrow{\text{def. }\mathbb{E},\mathsf{singlePath}} \forall\sigma \in \mathscr{L}\,.\,\sigma_{\mathsf{sp}} \curvearrowright_{\mathsf{s}} \sigma \xLeftrightarrow{\mathscr{L}(\mathtt{A}^{ac})\subseteq\mathscr{L}} \forall\sigma \in \mathscr{L}(\mathtt{A}^{ac})\,.\,\sigma_{\mathsf{sp}} \curvearrowright_{\mathsf{s}} \sigma$$

$$\xLeftrightarrow{\text{automata lift}} \forall\pi \in \mathsf{paths}(\mathtt{A}^{ac})\,.\,\sigma_{\mathsf{sp}} \curvearrowright_{\mathsf{s}} \sigma_\pi \xLeftrightarrow{\text{def. }\mathbb{E}^\sharp,2^{nd}\text{ case}} [\![\ \texttt{contains(s,s')}\ ]\!]^\sharp\mathtt{m}^\sharp = \{\texttt{true}\}$$

Otherwise, if $\mathtt{A}'$ is not a single-path automaton, the semantics returns $\{\texttt{true},\texttt{false}\}$, and soundness is met.

$\triangleright$ $[\![\ \texttt{contains(s,s')}\ ]\!]\gamma(\mathtt{m}^\sharp) = \{\texttt{true},\texttt{false}\}$. In this case, soundness is met as none of the conditions appearing in the definition of $[\![\ \texttt{contains(s,s')}\ ]\!]^\sharp$ are satisfied, and the latter returns $\{\texttt{true},\texttt{false}\}$ as well (3^{rd} case).

As a counterexample for the completeness of `contains`, let us consider $[\![\ \mathtt{s}\ ]\!]^\sharp\mathtt{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \{ab, bba\}$ and $[\![\ \mathtt{s}'\ ]\!]^\sharp\mathtt{m}^\sharp = \mathtt{A}' \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}') = \{a, b\}$.

$$[\![\ \texttt{contains(s,s')}\ ]\!]\gamma(\mathtt{m}^\sharp) = \{\texttt{true}\} \subsetneq \{\texttt{true},\texttt{false}\} = \gamma([\![\ \texttt{contains(s,s')}\ ]\!]^\sharp\mathtt{m}^\sharp)$$

□

A.4 IndexOf

Theorem 4. *$[\![\ \texttt{indexOf(s,s')}\ ]\!]^\sharp$ is a sound but not complete abstraction of $[\![\ \texttt{indexOf(s,s')}\ ]\!]$. Formally:*

$$\forall\mathtt{m}^\sharp \in \mathbb{M}^\sharp, \forall\mathtt{s},\mathtt{s}' \in \mathrm{SE}\,.\,[\![\ \texttt{indexOf(s,s')}\ ]\!]\gamma(\mathtt{m}^\sharp) \subsetneq \gamma([\![\ \texttt{indexOf(s,s')}\ ]\!]^\sharp\mathtt{m}^\sharp)$$

Proof. The collecting semantics of `indexOf` is defined as the additive lift of the concrete one: $[\![\ \texttt{indexOf(s,s')}\ ]\!]\mathtt{m} = \{\ i \mid i = \texttt{indexOf}(\sigma,\sigma'), \sigma \in \mathscr{L}, \sigma' \in \mathscr{L}'\ \}$, where $[\![\ \mathtt{s}\ ]\!]\mathtt{m} = \mathscr{L} \in \wp(\Sigma^*)$, $[\![\ \mathtt{s}'\ ]\!]\mathtt{m} = \mathscr{L}' \in \wp(\Sigma^*)$ and $\texttt{indexOf} : \Sigma^* \times \Sigma^* \to \mathbb{N}$ corresponds to the concrete semantics of Fig. 3. Let $[\![\ \mathtt{s}\ ]\!]^\sharp\mathtt{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \mathscr{L} \in \wp(\Sigma^*)$ and $[\![\ \mathtt{s}'\ ]\!]^\sharp\mathtt{m}^\sharp = \mathtt{A}' \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}') = \mathscr{L}' \in \wp(\Sigma^*)$. By definition of the concrete semantics, $[\![\ \texttt{indexOf(s,s')}\ ]\!]\gamma(\mathtt{m}^\sharp) \subseteq \gamma([-1,+\infty])$. When $\mathtt{A}$ or $\mathtt{A}'$ are cyclic or $\mathtt{A}'$ has a T transition, the abstract semantics returns the interval $[-1,+\infty]$, guaranteeing soundness. We thus continue by assuming that $\mathtt{A}$ and $\mathtt{A}'$ are not cyclic and $\mathtt{A}'$ has no T transitions (i.e., $\mathscr{L}'$ is finite). We split the proof following the possible concrete values.

$\triangleright$ $[\![\ \texttt{indexOf(s,s')}\ ]\!]\gamma(\mathtt{m}^\sharp) = \{-1\}$. No string of $\mathscr{L}'$ is contained in any string of $\mathscr{L}$. Formally:

$$[\![\ \texttt{indexOf(s,s')}\ ]\!]\gamma(\mathtt{m}^\sharp) = \{-1\} \xLeftrightarrow{\text{necessary condition}} \forall\sigma' \in \mathscr{L}'\ \nexists\sigma \in \mathscr{L}\,.\,\sigma' \curvearrowright_{\mathsf{s}} \sigma$$

$$\xLeftrightarrow{\text{def. }\mathbb{E}^\sharp,2^{nd}\text{ case}} [\![\ \texttt{indexOf(s,s')}\ ]\!]^\sharp\mathtt{m}^\sharp = [-1,-1] =$$

As $\gamma_{\mathsf{Intv}}([-1,-1]) = \{-1\}$, soundness is met.

$\triangleright$ $[\![\ \texttt{indexOf(s,s')}\ ]\!]\gamma(\mathtt{m}^\sharp) = I \subseteq \{\ n \in \mathbb{Z} \mid n \geq -1\ \}\,.\,\exists i \in I\,.\,i \geq 0$. This implies $\exists\sigma \in \mathscr{L}(\mathtt{A}), \sigma' \in \mathscr{L}(\mathtt{A}')\,.\,\sigma' \curvearrowright_{\mathsf{s}} \sigma$, as the collecting semantics returns at least one value that is not -1. Here, the abstract semantics relies on function IO that computes an interval for each string $\sigma' \in \mathscr{L}(\mathtt{A}')$, lubbing the results together. Hence, it is enough to prove the correctness of IO. Given $\sigma' \in \mathscr{L}'$, let us define the set $I_{\sigma'} \subseteq I = \{\ i \mid i = \texttt{indexOf}(\sigma,\sigma'), \sigma \in \mathscr{L}\ \}$ of positions where σ' can be found in $\mathscr{L}$ and let $m, M \in I_{\sigma'}$ be the minimal and the maximal elements of $I_{\sigma'}$. Therefore, it is sufficient to prove that $\gamma_{\mathsf{Intv}}([m,M]) \subseteq \gamma_{\mathsf{Intv}}([i,j])$, where $[i,j] = \mathsf{IO}(\mathtt{A},\sigma')$. For soundness to hold, $i \leq m$ and $M \leq j$ must be true, according to γ_{Intv}. We first prove $i \leq m$, identifying two cases. If $m = -1$:

$$-1 \in I_{\sigma'} \xLeftrightarrow{\text{necessary condition}} \exists\sigma \in \mathscr{L}\,.\,\sigma' \not\curvearrowright_{\mathsf{s}} \sigma \xLeftrightarrow{\text{automata lift}} \exists\pi \in \mathsf{paths}(\mathtt{A})\,.\,\sigma' \not\curvearrowright_{\mathsf{s}} \sigma_\pi \xLeftrightarrow{\text{def. }i,1^{st}\text{ case}} i = -1$$

Instead, if $m > -1$:

$$m = \min I_{\sigma'}, m \neq -1 \xLeftrightarrow{\text{necessary cond.}} \exists\sigma \in \mathscr{L}\,.\,\sigma_m \ldots \sigma_{m+|\sigma'|} = \sigma' \land \forall\sigma \in \mathscr{L}\ \nexists n < m\,.\,\sigma_n \ldots \sigma_{n+|\sigma'|} = \sigma'$$

$$\xLeftrightarrow{\text{automata lift}} \exists\pi \in \mathsf{paths}(\mathtt{A})\,.\,\exists\sigma_f \in \mathsf{Flat}(\sigma_\pi)\,.\,\sigma_{f_m} \ldots \sigma_{f_{m+|\sigma'|}} = \sigma' \land \forall\pi \in \mathsf{paths}(\mathtt{A})\ \forall\sigma_f \in \mathsf{Flat}(\sigma_\pi)\,.\,\sigma_{f_k} \ldots \sigma_{f_{k+|\sigma'|}} = \sigma' \Rightarrow k \geq m$$

$\xLeftrightarrow{\text{def. } i, 2^{nd} \text{ case}} i = \min\ k = m$

We now prove that $M \leq j$, identifying three cases. If $M = -1$:

$$I_{\sigma'} = \{-1\} \xLeftrightarrow{\text{necessary condition}} \forall \sigma \in \mathscr{L} \,.\, \sigma' \not\curvearrowright_{\mathsf{s}} \sigma \xLeftrightarrow{\text{automata lift}} \forall \pi \in \mathsf{paths}(\mathtt{A}) \,.\, \sigma' \not\curvearrowright_{\mathsf{s}} \sigma_\pi \xLeftrightarrow{\text{def. } j, 1^{st} \text{ case}} j = 1$$

Instead, if $M > -1$ and $\forall \pi \in \mathsf{paths}(\mathtt{A}) \,.\, \pi$ reads $\sigma \implies \pi$ does not read T before σ:

$$M = \max I_{\sigma'} \xLeftrightarrow{\text{necessary cond.}} \exists \sigma \in \mathscr{L} \,.\, \sigma_M \ldots \sigma_{M+|\sigma'|} = \sigma' \wedge \forall \sigma \in \mathscr{L}\ \nexists n > M \,.\, \sigma_n \ldots \sigma_{n+|\sigma'|} = \sigma'$$

$$\xLeftrightarrow{\text{automata lift}} \exists \pi \in \mathsf{paths}(\mathtt{A}) \,.\, \exists \sigma_f \in \mathsf{Flat}(\sigma_\pi) \,.\, \sigma_{f_M} \ldots \sigma_{f_{M+|\sigma'|}} = \sigma' \wedge \forall \pi \in \mathsf{paths}(\mathtt{A})\ \forall \sigma_f \in \mathsf{Flat}(\sigma_\pi) \,.\, \sigma_{f_k} \ldots \sigma_{f_{k+|\sigma'|}} = \sigma' \Rightarrow k \leq M$$

$$\xLeftrightarrow{\text{def. } j, 3^{rd} \text{ case}} j = \max\ k = M$$

Finally, if $M > -1$ and $\exists \pi \in \mathsf{paths}(\mathtt{A}) \,.\, \pi$ reads T before σ, $j = +\infty$ by the 2^{nd} case of the definition of j, that is thus greater than M. As both inequalities are always satisfied, we can conclude that soundness is met in all cases. $\square$

As a counterexample for completeness, let us consider $[\![\ \mathsf{s}\ ]\!]^\sharp \mathbb{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \{a, bba\}$ and $[\![\ \mathsf{s}'\ ]\!]^\sharp \mathbb{m}^\sharp = \mathtt{A}' \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}') = \{a\}$.

$$[\![\ \mathtt{indexOf}(\mathsf{s}, \mathsf{s}')\ ]\!]\gamma(\mathbb{m}^\sharp) = \{0, 2\} \subsetneq \gamma([0, 2]) = \gamma([\![\ \mathtt{indexOf}(\mathsf{s}, \mathsf{s}')\ ]\!]^\sharp \mathbb{m}^\sharp)$$

A.5 Repeat

The abstract semantics of `repeat` relies of the auxiliary function repeat, described by Alg. 2. First, we prove the soundness of repeat.

Theorem 5. *Given* $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$, $i \in \mathbb{N}$, repeat *is sound. Namely:*

$$\mathsf{repeat}(\mathscr{L}(\mathtt{A}), i) \subseteq \gamma(\mathsf{repeat}(\mathtt{A}, i))$$

where we abuse notation defining the collecting semantics $\mathsf{repeat}(\mathscr{L}, i) \triangleq \{\ \sigma^i \mid \sigma \in \mathscr{L}\ \}$.

Proof. We split the proof in the following cases.
▷ $i = 0$. In this case, Alg. 2 returns the automaton recognizing the empty string ($\mathsf{Min}(\{\epsilon\})$), and repeat is sound:

$$\mathsf{repeat}(\mathscr{L}(\mathtt{A}), i) = \{\ \sigma^0 \mid \sigma \in \mathscr{L}(\mathtt{A})\ \} = \{\epsilon\} = \gamma(\mathsf{Min}(\{\epsilon\})) \qquad \wr\text{lines 1–2 of Alg. 2}\wr$$

▷ $\mathtt{A}$ is cyclic. In the following we rely on the fact that, for some $i \in \mathbb{N}$:

$$\{\ \sigma^i \mid \sigma \in \mathscr{L}\ \} \subseteq \mathscr{L}^i \tag{A1}$$

$$\begin{aligned}
&\mathsf{repeat}(\mathscr{L}(\mathtt{A}), i) = \\
&\quad = \{\ \sigma^i \mid \sigma \in \mathscr{L}(\mathtt{A})\ \} && \wr\text{def. } \mathsf{repeat}(\mathscr{L}, i)\wr \\
&\quad = \{\epsilon\} \cdot \{\ \sigma^i \mid \sigma \in \mathscr{L}(\mathtt{A})\ \} && \wr\text{neutrality of } \epsilon \text{ for concatenation}\wr \\
&\quad \subseteq \{\epsilon\} \cdot \mathscr{L}(\mathtt{A})^i && \wr\text{Eq. } A1\wr
\end{aligned}$$

The last formula is the language obtained by concatenating the empty string with the i-concatenation of $\mathscr{L}(\mathtt{A})$, corresponding the language of concatenating i-times the automaton $\mathtt{A}$ with itself and the automaton recognizing ϵ. These operations are the ones performed at lines 3–8 of Alg. 2.
▷ $i \neq 0$ and $\mathtt{A}$ is not cyclic. In this case, we have:

$$\mathsf{repeat}(\mathscr{L}(\mathtt{A}), i) =$$

$$
\begin{aligned}
&= \{\ \sigma^i \mid \sigma \in \mathscr{L}(\mathtt{A})\ \} && \wr\text{def. } \mathsf{repeat}(\mathscr{L}, i)\wr \\
&= \{\ \sigma^i \mid \exists \pi \in \mathsf{paths}(\mathtt{A}).\ \sigma = \sigma_\pi\ \} && \wr \sigma \in \mathscr{L}(\mathtt{A}) \wr \\
&= \bigcup_{\pi \in \mathsf{paths}(\mathtt{A})} \sigma_\pi^i && \wr \sigma = \sigma_\pi \wr \\
&= \gamma(\bigsqcup_{\pi \in \mathsf{paths}(\mathtt{A})} \mathsf{Min}(\{\sigma_\pi^i\})) && \wr \text{def.} \gamma, \sqcup \wr
\end{aligned}
$$

The input of function γ in the last formula corresponds to lines 10–18 of Alg. 2, where $\mathsf{Min}(\{\sigma_\pi^i\})$ is computed by lines 13–15. □

Theorem 6. *$[\![\ \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ ]\!]^\sharp$ is a sound but not complete abstraction of $[\![\ \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ ]\!]$. Formally:*

$$\forall \mathfrak{m}^\sharp \in \mathbb{M}^\sharp, \forall \mathsf{s} \in \mathrm{SE}, \mathsf{a} \in \mathrm{AE}\,.\ [\![\ \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ ]\!]\gamma(\mathfrak{m}^\sharp) \subsetneq \gamma([\![\ \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ ]\!]^\sharp \mathfrak{m}^\sharp)$$

Proof. The collecting semantics of `replace` is defined as the additive lift of the concrete one, that is $[\![\ \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ ]\!]\mathfrak{m} = \{\ \sigma^k \mid k \in [i,j], \sigma \in \mathscr{L}\ \}$, where $[\![\ \mathsf{s}\ ]\!]\mathfrak{m} = \mathscr{L} \in \wp(\Sigma^*)$ and $[\![\ \mathsf{a}\ ]\!]\mathfrak{m} = [i,j] \in \wp(\mathbb{Z})$. Let $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \mathscr{L} \in \wp(\Sigma^*)$, and $[\![\ \mathsf{a}\ ]\!]^\sharp \mathfrak{m}^\sharp = [i,j] \in \mathsf{Intv}$. By definition of the concrete semantics, we suppose that $[i,j] \subseteq [0, +\infty]$. We split the proof in the following cases.
$\triangleright$ $[i,j] = [0, +\infty]$.

$$
\begin{aligned}
&[\![\ \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ ]\!]\gamma(\mathfrak{m}^\sharp) = \\
&\qquad = \{\ \sigma^i \mid \sigma \in \mathscr{L}(\mathtt{A}), i \in \mathbb{N}\ \} && \wr\text{def. } \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ \wr \\
&\qquad \subseteq \bigcup_{i \in \mathbb{N}} \mathscr{L}(A)^i && \wr\text{Eq. A1}\wr \\
&\qquad = \gamma(\mathsf{Kleene}(\mathtt{A})) && \wr\text{def. } \mathsf{Kleene}\wr
\end{aligned}
$$

$\triangleright$ $i == j \wedge i \in \mathbb{N}$. Soundness follows from Thm. 5.
$\triangleright$ $j = +\infty$.

$$
\begin{aligned}
&[\![\ \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ ]\!]\gamma(\mathfrak{m}^\sharp) = \\
&\qquad = \{\ \sigma^k \mid \sigma \in \mathscr{L}(\mathtt{A}), k \geq i\ \} && \wr\text{def. } \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ \wr \\
&\qquad = \{\ \sigma^i \sigma^l \mid \sigma \in \mathscr{L}(\mathtt{A}), l \in \mathbb{N}\ \} && \wr\text{def. } \geq \text{, string concat.}\wr \\
&\qquad = \{\ \sigma^i \mid \sigma \in \mathscr{L}(\mathtt{A})\ \} \cdot \{\ \sigma^l \mid \sigma \in \mathscr{L}(\mathtt{A}), l \in \mathbb{N}\ \} && \wr\text{def. lang. concat.}\wr \\
&\qquad \subseteq \gamma([\![\ \mathtt{concat}\ (\mathsf{repeat}(\mathtt{A}, i), \mathsf{Kleene}(\mathtt{A}))\ ]\!]^\sharp) && \wr\text{def. } \mathsf{Kleene}, \mathsf{repeat}, \gamma\wr
\end{aligned}
$$

$\triangleright$ $i, j \in \mathbb{N}$. Soundness follows from Thm. 5 and the definition of $\sqcup_{\mathcal{T}}$.
As a counterexample for completeness, let $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \{\ a^n \mid n \in \mathbb{N}\ \} \cup \{b\}$ and $[\![\ \mathsf{a}\ ]\!]^\sharp \mathfrak{m}^\sharp = [2,2]$.

$$[\![\ \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ ]\!]\gamma(\mathfrak{m}^\sharp) = \{\ a^n \mid n \in \mathbb{N}\ \} \cup \{bb\} \subsetneq \{\ a^n b a^m b \mid n, m \in \mathbb{N}\ \} = \gamma([\![\ \mathtt{repeat}\ (\mathsf{s}, \mathsf{a})\ ]\!]^\sharp \mathfrak{m}^\sharp)$$

□

A.6 TrimLeft, TrimRight, and Trim

The abstract semantics of `trimLeft` relies of the auxiliary function `trimL`, working on regexes. In the following we prove the soundness of `trimL`. The soundness proof for `trimR` is analogous, while soundness of `trim` follows from the soundness of `trimL` and `trimR`.

Theorem 7. *Given* $\mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$*, let* $\mathtt{r}$ *be the regex equivalent to* $\mathtt{A}$*.* $\mathtt{trimL}$ *is sound, namely:*

$$\mathtt{trimL}(\gamma(\mathtt{r})) \subseteq \gamma(\mathtt{trimL}(\mathtt{r}))$$

where we abuse notation of $\mathtt{trimL}$ *defining the collecting semantics* $\mathtt{trimL}(\mathscr{L}) \triangleq \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma \in \mathscr{L}, \sigma = \psi\sigma', \\ \psi = \max\{\ \psi' \in \{␣\}^* \mid \sigma = \psi'\sigma''\ \} \end{array} \right\}$.

Proof. The proof is done by induction on the structure of the regex $\mathtt{r}$.

Base cases

▷ $\mathtt{r} = \varnothing$. Soundness holds since $\mathtt{trimL}(\mathscr{L}(\varnothing)) = \varnothing = \gamma(\mathtt{trimL}(\varnothing))$ (1^{st} case).

▷ $\mathtt{r} = \mathsf{T}$. Soundness holds since $\mathtt{trimL}(\mathscr{L}(\mathsf{T})) = \Sigma^* = \gamma(\mathtt{trimL}(\mathsf{T}))$ (1^{st} case).

▷ $\mathtt{r} = \sigma$. If the regex is an atom, the abstract semantics relies on its concrete semantics, hence soundness holds.

Inductive cases

▷ $\mathtt{r} = \mathtt{r}_1 \parallel \mathtt{r}_2$.

$$\begin{aligned}
&\mathtt{trimL}(\gamma(\mathtt{r}_1 \parallel \mathtt{r}_2)) = \\
&\quad = \mathtt{trimL}(\gamma(\mathtt{r}_1) \cup \gamma(\mathtt{r}_2))) && \wr \text{def. } \gamma \wr \\
&\quad = \mathtt{trimL}(\gamma(\mathtt{r}_1)) \cup \mathtt{trimL}(\gamma(\mathtt{r}_2)) && \wr \text{distrib. of } \mathtt{trimL} \wr \\
&\quad = \gamma(\mathtt{trimL}(\mathtt{r}_1)) \cup \gamma(\mathtt{trimL}(\mathtt{r}_2)) && \wr \text{ind. hp.} \wr \\
&\quad = \gamma(\mathtt{trimL}(\mathtt{r}_1) \cup \mathtt{trimL}(\mathtt{r}_2)) && \wr \text{def. } \cup, \gamma \wr \\
&\quad = \gamma(\mathtt{trimL}(\mathtt{r}_1 \parallel \mathtt{r}_2)) && \wr \text{distrib. of } \mathtt{trimL}, 3^{rd} \text{ case} \wr
\end{aligned}$$

▷ $\mathtt{r} = \mathtt{r}_1\mathtt{r}_2$. For regex concatenation, we split the proof in three sub-cases.

- $\gamma(\mathtt{r}_1) \subseteq \{␣\}^* \implies \mathtt{trimL}(\gamma(\mathtt{r}_1)) = \epsilon$

$$\begin{aligned}
&\mathtt{trimL}(\gamma(\mathtt{r}_1\mathtt{r}_2)) = \\
&\quad = \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma \in \gamma(\mathtt{r}_1\mathtt{r}_2), \sigma = \psi\sigma', \\ \psi = \max\{\ \psi' \in \{␣\}^* \mid \sigma = \psi'\sigma''\ \} \end{array} \right\} && \wr \text{def. } \mathtt{trimL} \wr \\
&\quad = \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma_1 \in \gamma(\mathtt{r}_1), \sigma_2 \in \gamma(\mathtt{r}_2), \sigma_1\sigma_2 = \psi\sigma', \\ \psi = \max\{\ \psi' \in \{␣\}^* \mid \sigma_1\sigma_2 = \psi'\sigma''\ \} \end{array} \right\} && \wr \text{def. } \gamma(\mathtt{r}_1\mathtt{r}_2) \wr \\
&\quad = \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma_2 \in \gamma(\mathtt{r}_2), \sigma_2 = \psi\sigma', \\ \psi = \max\{\ \psi' \in \{␣\}^* \mid \sigma_2 = \psi'\sigma''\ \} \end{array} \right\} && \wr \mathtt{trimL}(\gamma(\mathtt{r}_1)) = \epsilon \implies \sigma_1 \in \psi \wr \\
&\quad = \mathtt{trimL}(\gamma(\mathtt{r}_2)) && \wr \text{def. } \mathtt{trimL} \wr \\
&\quad \subseteq \gamma(\mathtt{trimL}(\mathtt{r}_2)) && \wr \text{ind. hp.} \wr \\
&\quad = \gamma(\mathtt{trimL}(\mathtt{r}_1\mathtt{r}_2)) && \wr 4^{th} \text{ case} \wr
\end{aligned}$$

- $\mathsf{readWS}(\mathtt{r}_1) \implies \gamma(\mathtt{r}_1) = \mathscr{L}^{ws} \cup \mathscr{L}^{\neg ws}$ s.t. $\mathscr{L}^{ws} \subseteq \{␣\}^*$ and $\mathscr{L}^{\neg ws} \cap \{␣\}^* = \varnothing$

$$\begin{aligned}
&\mathtt{trimL}(\gamma(\mathtt{r}_1\mathtt{r}_2)) = \\
&\quad = \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma \in \gamma(\mathtt{r}_1\mathtt{r}_2), \sigma = \psi\sigma', \\ \psi = \max\{\ \psi' \in \{␣\}^* \mid \sigma = \psi'\sigma''\ \} \end{array} \right\} && \wr \text{def. } \mathtt{trimL} \wr \\
&\quad = \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma_1 \in \gamma(\mathtt{r}_1), \sigma_2 \in \gamma(\mathtt{r}_2), \sigma_1\sigma_2 = \psi\sigma', \\ \psi = \max\{\ \psi' \in \{␣\}^* \mid \sigma_1\sigma_2 = \psi'\sigma''\ \} \end{array} \right\} && \wr \text{def. } \gamma(\mathtt{r}_1\mathtt{r}_2) \wr \\
&\quad = \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma_1 \in \mathscr{L}^{ws}, \sigma_2 \in \gamma(\mathtt{r}_2), \sigma_1\sigma_2 = \psi\sigma', \\ \psi = \max\{\ \psi' \in \{␣\}^* \mid \sigma_1\sigma_2 = \psi'\sigma''\ \} \end{array} \right\} \\
&\quad \cup \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma_1 \in \mathscr{L}^{\neg ws}, \sigma_2 \in \gamma(\mathtt{r}_2), \sigma_1\sigma_2 = \psi\sigma', \\ \psi = \max\{\ \psi' \in \{␣\}^* \mid \sigma_1\sigma_2 = \psi'\sigma''\ \} \end{array} \right\} && \wr \text{def. } \mathscr{L}^{ws}, \mathscr{L}^{\neg ws} \wr
\end{aligned}$$

$$= \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma_2 \in \gamma(\mathtt{r}_2), \sigma_2 = \psi\sigma', \\ \psi = \max\{ \psi' \in \{\text{␣}\}^* \mid \sigma_2 = \psi'\sigma'' \} \end{array} \right\}$$

$$\cup \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma_1 \in \mathscr{L}^{\neg ws}, \sigma_2 \in \gamma(\mathtt{r}_2), \sigma_1\sigma_2 = \psi\sigma', \\ \psi = \max\{ \psi' \in \{\text{␣}\}^* \mid \sigma_1\sigma_2 = \psi'\sigma'' \} \end{array} \right\} \quad \wr \sigma_1 \in \{\text{␣}\}^* \wr$$

$$= \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma_2 \in \gamma(\mathtt{r}_2), \sigma_2 = \psi\sigma', \\ \psi = \max\{ \psi' \in \{\text{␣}\}^* \mid \sigma_2 = \psi'\sigma'' \} \end{array} \right\}$$

$$\cup \left\{ \sigma' \,\middle|\, \begin{array}{l} \sigma_1 \in \mathscr{L}^{\neg ws}, \sigma_2 \in \gamma(\mathtt{r}_2), \sigma_1\sigma_2 = \psi\sigma', \\ \psi = \max\{ \psi' \in \{\text{␣}\}^* \mid \sigma_1\sigma_2 = \psi'\sigma'' \} \end{array} \right\} \quad \wr \sigma_1 \notin \{\text{␣}\}^*, \psi \in \sigma_1 \wr$$

$$= \mathtt{trimL}(\gamma(\mathtt{r}_2)) \cup \mathtt{trimL}(\gamma(\mathtt{r}_1))\gamma(\mathtt{r}_2) \quad \wr \text{def. } \mathtt{trimL} \wr$$

$$\subseteq \gamma(\mathtt{trimL}(\mathtt{r}_2)) \cup \gamma(\mathtt{trimL}(\mathtt{r}_1)\mathtt{r}_2) \quad \wr \text{ind. hp., def. } \gamma \wr$$

$$= \gamma(\mathtt{trimL}(\mathtt{r}_2) \parallel \mathtt{trimL}(\mathtt{r}_1)\mathtt{r}_2) \quad \wr \text{def.} \gamma(\mathtt{r}_1 \parallel \mathtt{r}_2) \wr$$

$$= \gamma(\mathtt{trimL}(\mathtt{r}_1\mathtt{r}_2)) \quad \wr 5^{th} \text{ case} \wr$$

- $\neg\mathsf{readWS}(\mathtt{r}_1)$. The proof is analogous to the previous case.

$\triangleright$ $\mathtt{r} = (\mathtt{r}_1)^*$. If $\mathtt{trimL}(\mathtt{r}_1) = \epsilon$, then $\mathtt{trimL}(\mathtt{r}) = \epsilon = \gamma(\mathtt{trimL}(\mathtt{r}))$, hence soundness trivially holds. Otherwise, the proof is analogous to the case $\mathtt{r} = \mathtt{r}_1\mathtt{r}_2$. □

As a counterexample for completeness, let $[\![\mathsf{s}]\!]^\sharp \mathtt{m}^\sharp = \mathtt{A} = \mathsf{Min}(\{\mathsf{T}ab\})$, thus $\gamma(\mathtt{A}) = \{ \sigma ab \mid \sigma \in \Sigma^* \}$.

$$[\![\mathtt{trimLeft(s)}]\!]\gamma(\mathtt{m}^\sharp) = \{ \sigma ab \mid \sigma \in \Sigma^* \setminus \{\text{␣}\}^* \} \subsetneq \gamma(\mathsf{Min}(\{\mathsf{T}ab\}) = \gamma([\![\mathtt{trimLeft(s)}]\!]^\sharp \mathtt{m}^\sharp)$$

Consequently, also the abstract semantics of `trimRight` and `trim` are not complete.

A.7 Replace

Theorem 8. $[\![\mathit{replace}(\mathsf{s}, \mathsf{s}_s, \mathsf{s}_r)]\!]^\sharp$ *is a sound but not complete abstraction of* $[\![\mathit{replace}(\mathsf{s}, \mathsf{s}_s, \mathsf{s}_r)]\!]$. *Formally:*

$$\forall \mathtt{m}^\sharp \in \mathbb{M}^\sharp, \forall \mathsf{s}, \mathsf{s}_s, \mathsf{s}_r \in \mathtt{SE}\,.\, [\![\mathit{replace}(\mathsf{s}, \mathsf{s}_s, \mathsf{s}_r)]\!]\gamma(\mathtt{m}^\sharp) \subsetneq \gamma([\![\mathit{replace}(\mathsf{s}, \mathsf{s}_s, \mathsf{s}_r)]\!]^\sharp \mathtt{m}^\sharp)$$

Proof. The collecting semantics of `replace` is defined as the additive lift of the concrete one, that is $[\![\mathtt{replace}(\mathsf{s}, \mathsf{s}_s, \mathsf{s}_r)]\!]\mathtt{m} = \{ \sigma' \mid \sigma' = \mathtt{replace}(\sigma, \sigma_s, \sigma_r), \sigma \in \mathscr{L}, \sigma_s \in \mathscr{L}_s, \sigma_r \in \mathscr{L}_r \}$, where $[\![\mathsf{s}]\!]\mathtt{m} = \mathscr{L} \in \wp(\Sigma^*)$, $[\![\mathsf{s}_s]\!]\mathtt{m} = \mathscr{L}_s \in \wp(\Sigma^*)$, $[\![\mathsf{s}_r]\!]\mathtt{m} = \mathscr{L}_r \in \wp(\Sigma^*)$ and $\mathtt{replace} : \Sigma^* \times \Sigma^* \times \Sigma^* \to \Sigma^*$ corresponds to the concrete semantics of Fig. 3. Let $[\![\mathsf{s}]\!]^\sharp \mathtt{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \mathscr{L} \in \wp(\Sigma^*)$, $[\![\mathsf{s}_s]\!]^\sharp \mathtt{m}^\sharp = \mathtt{A}_s \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}_s) = \mathscr{L}_s \in \wp(\Sigma^*)$, and $[\![\mathsf{s}_r]\!]^\sharp \mathtt{m}^\sharp = \mathtt{A}_r \in \mathcal{T}\mathrm{FA}_{/\equiv}$ $\gamma(\mathtt{A}_r) = \mathscr{L}_r \in \wp(\Sigma^*)$. When $\mathtt{A}$ or $\mathtt{A}_s$ have a cycle or have a T-transition, our semantics returns $\mathsf{Min}(\{\mathsf{T}\})$ and is thus trivially sound. Otherwise, when no replacement happens, (i.e., $[\![\mathtt{replace}(\mathscr{L}, \mathscr{L}_s, \mathscr{L}_r)]\!]\mathtt{m} = \mathscr{L}$):

$$[\![\mathtt{replace}(\mathscr{L}, \mathscr{L}_s, \mathscr{L}_r)]\!] = \mathscr{L} \xLeftrightarrow{\text{necessary condition}} \forall \sigma_s \in \mathscr{L}_s\, \nexists \sigma \in \mathscr{L}\,.\, \sigma_s \not\curvearrowright_{\mathsf{s}} \sigma \xLeftrightarrow{\text{def. } \mathbb{E}^\sharp, 1^{st} \text{ case}} [\![\mathtt{replace}(\mathtt{A}, \mathtt{A}_s, \mathtt{A}_r)]\!] = \mathtt{A}$$

Otherwise, when at least one replacement happens, the semantics returns the lub of several applications of RP ranging over all possible combinations of strings in $\mathscr{L}$ and $\mathscr{L}_s$, that can be thoroughly explored since $\mathscr{L}$ and $\mathscr{L}_s$ are finite sets. Once RP has been proven correct, soundness naturally follows according to the properties of lub. We thus prove that $\forall \sigma \in \mathscr{L}, \forall \sigma_s \in \mathscr{L}_s, \forall \pi \in \mathsf{paths}(\mathtt{A})\,.\, \sigma_\pi = \sigma$:

$$[\![\mathtt{replace}(\{\sigma\}, \{\sigma_s\}, \mathscr{L}_r)]\!] \subseteq \gamma(\mathsf{RP}(\pi, \sigma_s, \mathtt{A}_r))$$

Specifically, RP removes every occurrence of σ_s in π (lines 7–8 of Alg. 3, where states and transitions composing σ_s are removed from the resulting automaton), substituting them with a copy of the replace automaton (line 4) that is connected to the path with ϵ–transitions. This means that all $\sigma' \curvearrowright_{\mathsf{s}} \sigma_\pi\,.\, \sigma' = \sigma_s$ are substituted with *all* the strings recognized by $\mathtt{A}_r$. We can then characterize the language of the automaton returned by RP as $\{ \sigma_\pi[\sigma_s/\sigma_r] \mid \sigma_r \in \mathscr{L}(\mathtt{A}_r) \}$. Soundness is thus ensured:

$$[\![\mathtt{replace}(\{\sigma\}, \{\sigma_s\}, \mathscr{L}_r)]\!] = \{ \sigma[\sigma_s/\sigma_r] \mid \sigma_r \in \mathscr{L}_r \}$$

$$\subseteq \{ \sigma_\pi[\sigma_s/\sigma_r] \mid \sigma_r \in \mathscr{L}(\mathtt{A}_r) \} \quad \wr \text{automata lift} \wr$$

$$= \gamma(\mathsf{RP}(\pi, \sigma_s, \mathtt{A}_r)) \qquad \wr\text{def. } \mathsf{RP}\wr$$

Soundness is thus proven as the result on individual strings can be lifted to languages, and since the $\mathtt{A}_r$ passed to RP is an over-approximation of the concrete strings it represents (as the semantics performs a may-replacement whenever $|\mathscr{L}_s| > 1$). The abstract semantics of `replace` is not complete, due to it returns $\mathsf{Min}(\{\mathsf{T}\})$ when either the input automaton or the search automaton contain cycles or read T.

As a counterexample for completeness, let $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$, $[\![\ \mathsf{s}^s\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A}^s \in \mathcal{T}\mathrm{FA}_{/\equiv}$, $[\![\ \mathsf{s}^r\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A}^r \in \mathcal{T}\mathrm{FA}_{/\equiv}$ s.t. $\gamma(\mathtt{A}) = \{abc\}$, $\gamma(\mathtt{A}^s) = \{a, z\}$ and $\gamma(\mathtt{A}^r) = \{r\}$.

$$[\![\ \mathtt{replace}(\mathsf{s}, \mathsf{s}^s, \mathsf{s}^r)\ ]\!]\gamma(\mathfrak{m}^\sharp) = \{rbc\} \subsetneq \{abc, rbc\} = \gamma([\![\ \mathtt{replace}(\mathsf{s}, \mathsf{s}^s, \mathsf{s}^r)\ ]\!]^\sharp \mathfrak{m}^\sharp)$$

□

A.8 Substring and CharAt

Theorem 9. $[\![\ \mathit{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2)\ ]\!]^\sharp$ *is a sound and complete abstraction of* $[\![\ \mathit{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2)\ ]\!]$. *Formally:*

$$\forall \mathfrak{m}^\sharp \in \mathbb{M}^\sharp, \forall \mathsf{s} \in \mathrm{SE}, \forall \mathsf{a}_1, \mathsf{a}_2 \in \mathrm{AE}\,.\, [\![\ \mathit{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2)\ ]\!]\gamma(\mathfrak{m}^\sharp) = \gamma([\![\ \mathit{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2)\ ]\!]^\sharp \mathfrak{m}^\sharp)$$

Proof. The collecting semantics of `substr` is defined as the additive lift of the concrete one, that is $[\![\ \mathtt{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2)\ ]\!]\mathfrak{m} = \{\ \sigma \mid \sigma = \mathtt{substr}(\sigma, i, j), \sigma \in \mathscr{L}, i \in I, j \in J\ \}$, where $[\![\ \mathsf{s}\ ]\!]\mathfrak{m} = \mathscr{L} \in \wp(\Sigma^*)$, $I = [\![\ \mathsf{a}\ ]\!]\mathfrak{m}$, $J = [\![\ \mathsf{a}'\ ]\!]\mathfrak{m}$ and $\mathtt{substr} : \Sigma^* \times \mathbb{N} \times \mathbb{N} \to \Sigma^*$ corresponds to the concrete semantics of Fig. 3. Without loss of generality, we can prove the semantics to be sound when $[\![\ \mathsf{a}_1\ ]\!]^\sharp \mathfrak{m}^\sharp = [i, i]$ and $[\![\ \mathsf{a}_2\ ]\!]^\sharp \mathfrak{m}^\sharp = [j, j]$, with $i, j \in \mathbb{N}, 0 \leq i \leq j$, as the abstract semantics lifts such result to non-singleton intervals applying lub. Let $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A}$ s.t. $\gamma(\mathtt{A}) = \mathscr{L}$, and that $[\![\ \mathsf{a}_1\ ]\!]^\sharp \mathfrak{m}^\sharp = [i, i]$ and $[\![\ \mathsf{a}_2\ ]\!]^\sharp \mathfrak{m}^\sharp = [j, j]$, with $i, j \in \mathbb{N}$. Furthermore, let $\mathtt{r} \equiv \mathtt{A}$ be the regex equivalent to $\mathtt{A}$. We can thus prove completeness of the semantics by proving the following:

$$[\![\ \mathtt{substr}(\mathscr{L}, \{i\}, \{j\})\ ]\!]\gamma(\mathfrak{m}^\sharp) = \gamma(\mathsf{Min}(\{\ \sigma \mid (\sigma, 0, 0) \in \mathsf{Sb}(\mathtt{r}, i, j - i)\ \})).$$

Completeness is proven by structural induction over the structure of the regular expression, referencing the lines of Alg. 4 that are involved in the computation as §x, where x is the line number. Moreover, when $\mathsf{Sb}(\mathtt{r}, i, j)$ produces the set $S = \{(\sigma_1, i_1, j_1), \dots, (\sigma_n, i_n, j_n)\}$, we denote the automaton $\mathsf{Min}(\{\ \sigma \mid (\sigma, 0, 0) \in S\ \})$ as either, abusing notation, $\mathsf{Min}(\mathsf{Sb}(\mathtt{r}, i, j))$ or $\mathsf{Min}(\{(\sigma_1, i_1, j_1), \dots, (\sigma_n, i_n, j_n)\})$. With the latter notation, we abuse notation writing $\sigma_i \notin \mathsf{Sb}$ to denote that (σ_i, i_i, j_i) is not in the final result of Sb.

Base cases

▷ $\mathtt{r} = \varnothing$ ($\mathscr{L}(\mathtt{r}) = \varnothing$):

$$\begin{aligned} [\![\ \mathtt{substr}(\varnothing, \{i\}, \{j\})\ ]\!] &= \varnothing \\ &= \gamma(\mathsf{Min}(\varnothing)) && \wr\text{automata lift}\wr \\ &= \gamma(\mathsf{Min}(\mathsf{Sb}(\varnothing, i, j - i))) && \wr\S 2\wr \end{aligned}$$

▷ $\mathtt{r} = \sigma \in \Sigma^*$: here, we identify three cases. If $i \leq j < |\sigma|$:

$$\begin{aligned} [\![\ \mathtt{substr}(\{\sigma\}, \{i\}, \{j\})\ ]\!] &= \{\sigma_i \dots \sigma_j\} \\ &= \gamma(\mathsf{Min}(\{\sigma_i \dots \sigma_j\})) && \wr\text{automata lift}\wr \\ &= \gamma(\mathsf{Min}(\mathsf{Sb}(\{\sigma\}, i, j - i))) && \wr\S 6\wr \end{aligned}$$

Instead, when $i > |\sigma|$:

$$\begin{aligned} [\![\ \mathtt{substr}(\{\sigma\}, \{i\}, \{j\})\ ]\!] &= \varnothing \\ &= \gamma(\mathsf{Min}(\{(\epsilon, i - |\sigma|, j - i)\})) && \wr i - |\sigma| > 0 \implies \epsilon \notin \mathsf{Sb}\wr \\ &= \gamma(\mathsf{Min}(\mathsf{Sb}(\{\sigma\}, i, j - i))) && \wr\S 4\wr \end{aligned}$$

computing an empty partial substring (that is still concretized as the empty set of strings), but taking into account that σ has been read $(i - |\sigma|)$ and no character from σ has been taken $(j - i)$. Finally, if $i < |\sigma|$ and $j > |\sigma|$ (where $k = j - |\sigma| + i$):

$$
\begin{aligned}
\llbracket \texttt{substr}(\{\sigma\}, \{i\}, \{j\}) \rrbracket &= \varnothing \\
&= \gamma(\mathsf{Min}(\{(\sigma_i \ldots \sigma_{|\sigma|-1}, 0, k)\})) && \wr k > 0 \implies \sigma_i \ldots \sigma_{|\sigma|-1} \notin \mathsf{Sb} \int \\
&= \gamma(\mathsf{Min}(\mathsf{Sb}(\{\sigma\}, i, j - i))) && \wr \S 5 \int
\end{aligned}
$$

computing an partial substring (that is still concretized as the empty set of strings) that is a suffix of σ, and noting that $j - (i - |\sigma_i \ldots \sigma_{|\sigma|-1}|)$ characters still have to be read before completing the substring.
$\triangleright$ $\mathtt{r} = \mathsf{T}$:

$$
\begin{aligned}
\llbracket \texttt{substr}(\Sigma^*, \{i\}, \{j\}) \rrbracket &= \{\ \sigma \mid |\sigma| = j - i\ \} \\
&= \gamma(\mathsf{Min}(\{(\bullet^{j-i}, 0, 0)\})) && \wr \text{automata lift} \int \\
&\quad \cup\ \gamma(\mathsf{Min}(\{(\bullet^{l}, 0, j - l)\})), l < j - i && \wr j - l > 0 \implies \bullet^{l} \notin \mathsf{Sb} \int \\
&\quad \cup\ \gamma(\mathsf{Min}(\{(\epsilon, i - l, j)\})), 0 \leq l \leq i && \wr i - l > 0 \implies \epsilon \notin \mathsf{Sb} \int \\
&= \gamma(\mathsf{Min}(\mathsf{Sb}(\mathsf{T}, i, j - i))) && \wr \S 8, \S 9 \int
\end{aligned}
$$

where, for the sake of clarity, strings returned by Sb are split into three sets, the first $(\{(\bullet^{j-i}, 0, 0)\})$ simulating substrings generated when $i, j \leq |\sigma|$, the second one $((\bullet^{l}, 0, j - l))$ representing partial substrings when $i \geq |\sigma|$, and the third symbolizing partial substrings generated when $i < |\sigma| \wedge j \geq |\sigma|$. Note that only strings from the first set are part of the final concretization, while partial substrings from the second and third automata only serve in computations of successive substrings.

Inductive steps

$\triangleright$ $\mathtt{r} = \mathtt{r}_1 || \mathtt{r}_2$: let $\mathscr{L}, \mathscr{L}_1, \mathscr{L}_2 \in \wp(\Sigma^*)$ be the languages recognized by $\mathtt{r}$, $\mathtt{r}_1$ and $\mathtt{r}_2$, respectively. It is easy to see that $\llbracket \texttt{substr}(\mathscr{L}, \{i\}, \{j\}) \rrbracket = \llbracket \texttt{substr}(\mathscr{L}_1, \{i\}, \{j\}) \rrbracket \cup \llbracket \texttt{substr}(\mathscr{L}_2, \{i\}, \{j\}) \rrbracket$. We assume $\llbracket \texttt{substr}(\mathscr{L}_1, \{i\}, \{j\}) \rrbracket = \gamma(\mathsf{Min}(\mathsf{Sb}(\mathtt{r}_1, i, j - i)))$ and $\llbracket \texttt{substr}(\mathscr{L}_2, \{i\}, \{j\}) \rrbracket = \gamma(\mathsf{Min}(\mathsf{Sb}(\mathtt{r}_2, i, j - i)))$ to hold for inductive hypothesis. We then prove soundness with the following:

$$
\begin{aligned}
\llbracket \texttt{substr}(\mathscr{L}, \{i\}, \{j\}) \rrbracket &= \llbracket \texttt{substr}(\mathscr{L}_1, \{i\}, \{j\}) \rrbracket \cup \llbracket \texttt{substr}(\mathscr{L}_2, \{i\}, \{j\}) \rrbracket \\
&= \gamma(\mathsf{Min}(\mathsf{Sb}(\mathtt{r}_1, i, j - i))) \cup \gamma(\mathsf{Min}(\mathsf{Sb}(\mathtt{r}_2, i, j - i))) && \wr \text{ind. hp.} \int \\
&= \gamma(\mathsf{Min}(\mathsf{Sb}(\mathtt{r}_1 || \mathtt{r}_2, i, j - i))) && \wr \S 21 \int
\end{aligned}
$$

$\triangleright$ $\mathtt{r} = \mathtt{r}_1 \mathtt{r}_2$: let $\mathscr{L}, \mathscr{L}_1, \mathscr{L}_2 \in \wp(\Sigma^*)$ be the languages recognized by $\mathtt{r}$, $\mathtt{r}_1$ and $\mathtt{r}_2$, respectively. The concrete semantics is the union of two sets: $\llbracket \texttt{substr}(\mathscr{L}_1, \{i\}, \{j\}) \rrbracket$ (i.e., substrings that are fully contained in $\mathscr{L}_1$), and $\llbracket \texttt{substr}(\mathscr{L}_1 \cdot \mathscr{L}_2, \{i\}, \{j\}) \rrbracket$ (i.e., substrings that straddle $\mathscr{L}_1$ and $\mathscr{L}_2$). We prove soundness assuming the inductive hypotheses $\llbracket \texttt{substr}(\mathscr{L}_1, \{i\}, \{j\}) \rrbracket = \gamma(\mathsf{Min}(\mathsf{Sb}(\mathtt{r}_1, i, j - i)))$ and $\llbracket \texttt{substr}(\mathscr{L}_2, \{i\}, \{j\}) \rrbracket = \gamma(\mathsf{Min}(\mathsf{Sb}(\mathtt{r}_2, i, j - i)))$:

$$
\begin{aligned}
\llbracket \texttt{substr}(\mathscr{L}, \{i\}, \{j\}) \rrbracket &= \llbracket \texttt{substr}(\mathscr{L}_1, \{i\}, \{j\}) \rrbracket \cup \llbracket \texttt{substr}(\mathscr{L}_1 \cdot \mathscr{L}_2, \{i\}, \{j\}) \rrbracket \\
&= \gamma(\mathsf{Min}(\mathsf{Sb}(\mathtt{r}_1, i, j - i))) \cup \gamma(\mathsf{Min}(\{(\sigma_1^1 \cdot \sigma_2^1, i_2^1, j_2^1), \ldots, (\sigma_1^n \cdot \sigma_2^n, i_2^n, j_2^n)\})) && \wr \text{ind. hp.} \int \\
&= \gamma(\mathsf{Min}(\mathsf{Sb}(\mathtt{r}_1 \mathtt{r}_2, i, j - i))) && \wr \S 1, \S 16, \S 18 \int
\end{aligned}
$$

where, for the sake of clarity, strings returned by Sb are split in two sets, the first $(\mathsf{Sb}(\mathtt{r}_1, i, j - i))$ corresponding to substrings that entirely contained into $\mathtt{r}_1$, the second one $(\{(\sigma_1^1 \cdot \sigma_2^1, i_2^1, j_2^1), \ldots, (\sigma_1^n \cdot \sigma_2^n, i_2^n, j_2^n)\})$ that models substrings straddling $\mathtt{r}_1$ and $\mathtt{r}_2$, where $\forall i \,.\, (\sigma_1^i, i_1^i, j_1^i) \in \mathsf{Sb}(\mathtt{r}_1, i, j - i), j_1^i \neq 0 \wedge (\sigma_2^i, i_2^i, j_2^i) \in \mathsf{Sb}(\mathtt{r}_2, i_1^i, j_1^i)$. Strings in the latter set are built by offsetting substrings of $\mathtt{r}_2$ by the length of the substrings of $\mathtt{r}_1$.
$\triangleright$ $\mathtt{r} = (\mathtt{r}_1)^*$. The proof of this case is similar to the one for concatenation, since $(\mathtt{r}_1)^*$ can be seen as an (undefined) concatenation of the regular expression $\mathtt{r}_1$, and is thus left implicit. □

Theorem 10. $\llbracket\ \mathit{charAt}\ (\mathsf{s}, \mathsf{a})\ \rrbracket^\sharp$ *is a sound and complete abstraction of* $\llbracket\ \mathit{charAt}\ (\mathsf{s}, \mathsf{a})\ \rrbracket$. *Formally:*

$$
\forall \mathsf{m}^\sharp \in \mathbb{M}^\sharp, \forall \mathsf{s} \in \textsc{se}, \forall \mathsf{a} \in \textsc{ae} \,.\, \llbracket\ \mathit{charAt}\ (\mathsf{s}, \mathsf{a})\ \rrbracket \gamma(\mathsf{m}^\sharp) = \gamma(\llbracket\ \mathit{charAt}\ (\mathsf{s}, \mathsf{a})\ \rrbracket^\sharp \mathsf{m}^\sharp)
$$

Proof. Since the abstract semantics of `charAt` relies on the one of `substr`, soundness and completeness come from Thm 9. □

A.9 String Equality

We report the soundness and completeness proof of the abstract semantics of string equality. First, we prove the soundness of the eq function (whose algorithm is reported in Alg. 1).

Theorem 11. *Given $\sigma_1, \sigma_2 \in \Sigma^* \cup \{\mathsf{T}\}$, $[\![\ \mathsf{Flat}(\sigma_1) == \mathsf{Flat}(\sigma_2)\ ]\!]$ is a sound and complete approximation of $\mathsf{eq}(\sigma_1, \sigma_2)$. Formally:*

$$\forall \sigma_1, \sigma_2 \in \Sigma^* \cup \{\mathsf{T}\}\ .\ \mathsf{Flat}(\sigma_1) == \mathsf{Flat}(\sigma_2) = \gamma(\mathsf{eq}(\sigma_1, \sigma_2))$$

where we abuse notation denoting by == also the collecting semantics of string equality.

Proof. The proof is done by natural induction over the length of σ_1 and σ_2.
Base cases
$\triangleright\ |\sigma_1| = 0 \iff \sigma_1 = \epsilon$

- $|\sigma_2| = 0 \iff \sigma_2 = \epsilon$. In this case, $\mathsf{Flat}(\sigma_1) == \mathsf{Flat}(\sigma_2) = \{\mathtt{true}\}$ that is equal to the result returned by eq in Alg. 1 when both strings are empty (lines 1-2).
- $|\sigma_2| = 1 \iff \sigma_2 = c \in \Sigma \cup \{\mathsf{T}\}$. We can split the proof in two cases:
  - $c = \mathsf{T}$: the empty string may be equal to T, thus $\mathsf{Flat}(\sigma_1) == \mathsf{Flat}(\sigma_2) = \{\mathtt{true}, \mathtt{false}\}$, that is equal to the result returned by eq in Alg. 1 when one of the string is empty and the other is equal to T (lines 3-4).
  - $c \neq \mathsf{T}$: the empty string is not equal to any string, thus $\mathsf{Flat}(\sigma_1) == \mathsf{Flat}(\sigma_2) = \{\mathtt{false}\}$ that is equal to the result returned by eq in Alg. 1 when one of string is empty and the other has a single character not equal to T (lines 5-6).

$\triangleright\ |\sigma_1| = 1 \iff \sigma_1 = c \in \Sigma \cup \{\mathsf{T}\}$. We can split the proof in two case

- $c = \mathsf{T}$.
  - $|\sigma_2| = 0 \iff \sigma_2 = \epsilon$. This case is analogous to the second point of the previous base case.
  - $|\sigma_1| = 1 \iff \sigma_2 = c' \in \Sigma \cup \{\mathsf{T}\}$. We have two cases
    - $c' = \mathsf{T}$. Two strings just made of a singleton T character may be equal, thus $\mathsf{Flat}(\sigma_1) == \mathsf{Flat}(\sigma_2) = \{\mathtt{true}, \mathtt{false}\}$ that is equal to the result returned by eq in Alg. 1 when both strings have a single character equal to T (lines 7-8).
    - $c' \neq \mathsf{T}$. A string just made of a singleton T may be equal to $c \in \Sigma \cup \{\mathsf{T}\}$, thus $\mathsf{Flat}(\sigma_1) == \mathsf{Flat}(\sigma_2) = \{\mathtt{true}, \mathtt{false}\}$, that is equal to the result returned by eq in this case (lines 7-8) Alg. 1
- $c \neq \mathsf{T}$.
  - $|\sigma_2| = 0 \iff \sigma_2 = \epsilon$. This case is analogous to the second case of the first base case.
  - $|\sigma_2| = 1 \iff \sigma_2 = c' \in \Sigma \cup \{\mathsf{T}\}$. This case is analogous to the previous case ($c = \mathsf{T}$).

Inductive steps Let $n \in \mathbb{N}$ and let $\sigma_1, \sigma_2 \in (\Sigma \cup \{\mathsf{T}\})^*$ such that $|\sigma_1| \leq n$, $|\sigma_2| \leq n$. For inductive hypothesis the following holds

$$\mathsf{Flat}(\sigma_1) == \mathsf{Flat}(\sigma_2) \subseteq \gamma(\mathsf{eq}(\sigma_1, \sigma_2))$$

Given $\rho_1, \rho_2 \in (\Sigma \cup \{\mathsf{T}\})^*$ such that $|\rho_1| > n$, $|\rho_2| > n$ we prove that

$$\mathsf{Flat}(\rho_1) == \mathsf{Flat}(\rho_2) \subseteq \gamma(\mathsf{eq}(\rho_1, \rho_2))$$

Let us consider $\rho_1 = c\sigma_1$ and $\rho_2 = c'\sigma_2$. We split the proof in the following cases.

$\triangleright\, c \neq \mathsf{T}, c' \neq \mathsf{T}$

$$\begin{aligned}
&\mathsf{Flat}(\rho_1) == \mathsf{Flat}(\rho_2) = \\
&\qquad = \mathsf{Flat}(c\sigma_1) == \mathsf{Flat}(c'\sigma_2) && \wr\text{def. } \rho_1 \text{ and } \rho_2 \wr \\
&\qquad = \{\ cs == c's' \mid s \in \mathsf{Flat}(\sigma_1), s' \in \mathsf{Flat}(\sigma_2)\ \} && \wr\text{def. } == \wr
\end{aligned}$$

We split the proof in the following cases.

- $c = c'$

$$\begin{aligned}
&\{\ cs == c's' \mid s \in \mathsf{Flat}(\sigma_1), s' \in \mathsf{Flat}(\sigma_2)\ \} = \\
&\qquad = \{\ s == s' \mid s \in \mathsf{Flat}(\sigma_1), s' \in \mathsf{Flat}(\sigma_2)\ \} && \wr c = c' \wr \\
&\qquad = \gamma(\mathsf{eq}(\sigma_1, \sigma_2)) && \wr\text{ind. hp.}\wr \\
&\qquad = \gamma(\mathsf{eq}(\rho_1, \rho_2)) && \wr\text{lines 11-12}\wr
\end{aligned}$$

- $c \neq c'$

$$\begin{aligned}
&\{\ cs == c's' \mid s \in \mathsf{Flat}(\sigma_1), s' \in \mathsf{Flat}(\sigma_2)\ \} = \\
&\qquad = \{\texttt{false}\} && \wr c \neq c' \wr \\
&\qquad = \gamma(\mathsf{eq}(\rho_1, \rho_2)) && \wr\text{lines 9-10}\wr
\end{aligned}$$

$\triangleright\, c = \mathsf{T}, c' \neq \mathsf{T}$

$$\begin{aligned}
&\mathsf{Flat}(\rho_1) == \mathsf{Flat}(\rho_2) = \\
&\qquad = \mathsf{Flat}(\mathsf{T}\sigma_1) == \mathsf{Flat}(c'\sigma_2) && \wr\text{def. } \rho_1 \text{ and } \rho_2 \wr \\
&\qquad = \{\ ts == c's' \mid s \in \mathsf{Flat}(\sigma_1), s' \in \mathsf{Flat}(\sigma_2), t \in \Sigma^*\ \} && \wr\text{def. } == \wr \\
&\qquad = \{\ s == c's' \mid s \in \mathsf{Flat}(\sigma_1), s' \in \mathsf{Flat}(\sigma_2), t \in \Sigma^*\ \} \\
&\qquad \cup \{\ cs == cs' \mid s \in \mathsf{Flat}(\sigma_1), s' \in \mathsf{Flat}(\sigma_2)\ \} \\
&\qquad \cup \{\ ts == c's' \mid s \in \mathsf{Flat}(\sigma_1), s' \in \mathsf{Flat}(\sigma_2), t \in \Sigma^* \smallsetminus \{\epsilon, c\}\ \} && \wr\text{set union def.}\wr \\
&\qquad = \gamma(\mathsf{eq}(\rho_1, \rho_2[1:])) && \wr\text{ind. hp.}\wr \\
&\qquad \sqcup \gamma(\mathsf{eq}(\rho_1[1:], \rho_2[1:]) && \wr\text{ind. hp.}\wr \\
&\qquad \sqcup \gamma(\mathsf{eq}(\rho_1, \rho_2[1:]) \sqcup \texttt{false} && \wr\text{ind. hp., } t = \epsilon \vee t \neq c\wr \\
&\qquad = \gamma(\mathsf{eq}(\rho_1, \rho_2)) && \wr\text{lines 13-14}\wr
\end{aligned}$$

$\triangleright\, c = \mathsf{T}, c' = \mathsf{T}$. The proof is analogous to the previous case.
$\triangleright\, c \neq \mathsf{T}, c' = \mathsf{T}$. The proof is analogous to the previous cases. □

Theorem 12. $[\![\ \mathsf{s} == \mathsf{s}'\ ]\!]^\sharp$ *is a sound and complete abstraction of* $[\![\ \mathsf{s} == \mathsf{s}'\ ]\!]$. *Formally:*

$$\forall \mathfrak{m}^\sharp \in \mathbb{M}^\sharp, \forall \mathsf{s}, \mathsf{s}' \in \textsc{se}\,.\ [\![\ \mathsf{s} == \mathsf{s}'\ ]\!]\gamma(\mathfrak{m}^\sharp) \subseteq \gamma([\![\ \mathsf{s} == \mathsf{s}'\ ]\!]^\sharp \mathfrak{m}^\sharp)$$

Proof. In the first case of the abstract semantics of string equality, soundness and completeness are trivially met, while soundness and completeness of the third case follow from Thm. 11. Let us focus on the second case, let $\mathsf{s}, \mathsf{s}' \in \textsc{se}$ and suppose $[\![\ \mathsf{s}\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A} \in \mathcal{T}\mathrm{FA}_{/\equiv}$, $[\![\ \mathsf{s}'\ ]\!]^\sharp \mathfrak{m}^\sharp = \mathtt{A}' \in \mathcal{T}\mathrm{FA}_{/\equiv}$. We prove that if either $\mathtt{A}$ or $\mathtt{A}'$ are cyclic, $[\![\ \mathsf{s} == \mathsf{s}'\ ]\!]\gamma(\mathfrak{m}^\sharp) = \{\texttt{true}, \texttt{false}\}$, the same result returned by $\gamma([\![\ \mathsf{s} == \mathsf{s}'\ ]\!]^\sharp \mathfrak{m}^\sharp)$, proving completeness. Note that $[\![\ \mathsf{s} == \mathsf{s}'\ ]\!]\gamma(\mathfrak{m}^\sharp)$ cannot be $\{\texttt{false}\}$ since this case is treated in the first case of the abstract semantics of string equality. By contradiction, let either $\mathtt{A}$ or $\mathtt{A}'$ be cyclic and let us suppose that $[\![\ \mathsf{s} == \mathsf{s}'\ ]\!]\gamma(\mathfrak{m}^\sharp) = \{\texttt{true}\}$.

$[\![\ \mathsf{s} == \mathsf{s}'\ ]\!]\gamma(\mathfrak{m}^\sharp) = \{\texttt{true}\}$

$$\iff \forall\sigma \in \gamma(\mathtt{A})\forall\sigma' \in \gamma(\mathtt{A}') \,.\, \sigma == \sigma' \qquad \wr\text{def. } [\![\, \mathsf{s} == \mathsf{s}' \,]\!]\wr$$

$$\implies |\gamma(\mathtt{A})| = |\gamma(\mathtt{A}')| = 1 \qquad \wr\text{set theory, def. } ==\wr$$

We supposed that either $\mathtt{A}$ or $\mathtt{A}'$ were cyclic, reaching a contradiction. Thus, if either $\mathtt{A}$ or $\mathtt{A}'$ are cyclic, $[\![\, \mathsf{s} == \mathsf{s}' \,]\!]\gamma(\mathbb{m}^\sharp) = \{\mathtt{false}, \mathtt{true}\} = \gamma([\![\, \mathsf{s} == \mathsf{s}' \,]\!]^\sharp \mathbb{m}^\sharp)$, proving completeness. $\square$