Static Program Analysis for String Manipulation Languages

Vincenzo Arceri
University of Verona, Verona, Italy
vincenzo.arceri@univr.it

Isabella Mastroeni
University of Verona, Verona, Italy
isabella.mastroeni@univr.it

In recent years, dynamic languages, such as JavaScript or Python, have been increasingly used in a wide range of fields and applications. Their tricky and misunderstood behaviors pose a hard challenge for static analysis of these programming languages. A key aspect of any dynamic language program is the multiple usage of strings, since they can be implicitly converted to another type value, transformed by string-to-code primitives or used to access an object-property. Unfortunately, string analyses for dynamic languages still lack precision and do not take into account some important string features. Moreover, string obfuscation is very popular in the context of dynamic language malicious code, for example, to hide code information inside strings and then to dynamically transform strings into executable code. In this scenario, more precise string analyses become a necessity. This paper is placed in the context of static string analysis by abstract interpretation and proposes a new semantics for string analysis, placing a first step for handling dynamic languages string features.

1 Introduction

Dynamic languages, such as JavaScript or Python, have faced an important increment of usage in a very wide range of fields and applications. Common features in dynamic languages are dynamic typing (typing occurs during program execution, at run-time) and implicit type conversion [38], lightening the development phase and allowing not to block the program execution in presence of unexpected or unpredictable situations. Moreover, one important aspect of dynamic languages is the way strings may be used. In JavaScript, for example, strings can be either used to access property objects or transformed into executable code, by using the global function `eval`. In this way, dynamic languages provide multiple string features that simplify writing programs, allowing, at the same time, statically unpredictable executions which may make programs harder to understand [38]. For this reason, string obfuscation (e.g., string splitting) is becoming one of the most common obfuscation techniques in JavaScript malware [42], making hard to statically analyze code. Consider, for example, the JavaScript program fragment in Fig. 1 where strings are manipulated, de-obfuscated, combined together into the variable `d` and finally transformed into executable code, the statement `ws = new ActiveXObject(WScript.Shell)`. This command, in Internet Explorer, opens a shell which may execute malicious commands. The command is not hard-coded in the fragment but it is built at run-time and the initial values of `i`,`j` and `k` are unknown, such as the number of iterations of the loops in the fragment. These observations suggest us that, in order to statically understand statements dynamically generated and executed, it may be extremely useful to statically analyze the string value of `d`. Unfortunately, existing static analyzers for dynamic languages [27, 30, 32, 33], may fail to precisely analyze strings in dynamic contexts. For instance, in the example, existing static analyzers [30, 32, 33] lose precision on the `eval` input value, losing any information about it. Namely, the issue of analyzing dynamic

```
vd, ac, la = "";
v = "wZsZ"; m = "AYcYtYiYvYeYXY";
tt = "AObyaSZjectB";
l = "WYSYcYrYiYpYtY.YSYhYeYlYlY";

while (i+=2 < v.length)
  vd = vd + v.charAt(i);

while (j+=2 < m.length)
  ac = ac + m.charAt(j);
```

```
ac += tt.substring(tt.indexOf("O"), 3);
ac += tt.substring(tt.indexOf("j"), 11)
  ;

while (k+=2 < l.length)
  la = la + l.charAt(k);

d = vd + "=new " + ac + "(" + la + ")";
eval(d);
```

Figure 1: A potentially malicious obfuscated JavaScript program.

languages, even if tackled by sophisticated tools as the cited ones, still lacks formal approaches for handling the more dynamic features of string manipulation, such as dynamic typing, implicit type conversion and dynamic code generation.

Contributions. In this paper, we focus on the characterization of an abstract interpretation-based [15] formal framework for handling dynamic typing and implicit type conversion, by defining an abstract semantics able to capture these dynamic features. Even if we do not tackle the problem of analyzing dynamically generated code (meaning that we do not analyze its *behavior*), we strongly believe that such a semantics is a necessary step towards a sufficiently precise analysis of dynamically generated code, being able to reason about a class of string manipulation programs (as far as string values are concerned) that state-of-art static analyzers would fail to precisely analyze. Indeed, the domain we propose allows us to *collect* (and potentially approximate) the set of all the string values that a variable may receive during computation (at each program point). It should be clear that, in order to analyze *what* an `eval` statement may execute, we surely need to (over-)approximate the set of precise values that its parameter may have. Hence, we propose an approach aiming at defining a collecting semantics for strings. With this task in mind, we first discuss how to combine abstract domains of primitive types (strings, integers and booleans) in order to capture dynamic typing. Once we have such an abstract domain, we define on it an abstract semantics for a toy language, augmented with implicit type conversion, dynamic typing and some interesting string operations, whose concrete semantics is inspired by the JavaScript one. In particular, for each one of these operations we provide the algorithm computing its abstract semantics and we discuss their soundness and completeness.

Paper structure. In Sect. 2 we recall relevant notions on finite state automata and the core language we adopt for this paper and the finite state automata domain, highlighting some important operations and theoretical results, respectively. In Sect. 3 we discuss and present two ways of combining abstract domains (for primitive types) suitable for dynamic languages. Then, In Sect. 4 we present the novel abstract semantics for string manipulation programs. Finally, in Sect. 5 we discuss the related work compared to this paper and we conclude the paper.

2 Background

2.1 Basic notations and concepts

String notation. We denote by Σ a finite alphabet of symbols, its Kleene-closure by Σ^* and a string element by $\sigma \in \Sigma^*$. If $\sigma = \sigma_0\sigma_1 \cdots \sigma_n$, the length of σ is $|\sigma| = n+1$ and the element in the i-th position is σ_i. Given two strings $\sigma, \sigma' \in \Sigma^*$, $\sigma\sigma'$ is their concatenation. A language is a set of strings, i.e., $\mathtt{L} \in \wp(\Sigma^*)$. We use the following notations: $\Sigma^i \overset{\text{def}}{=} \{\, \sigma \in \Sigma^* \mid |\sigma| = i \,\}$ and $\Sigma^{<i} \overset{\text{def}}{=} \bigcup_{j<i} \Sigma^j$. Given $\sigma \in \Sigma^*$, $i, j \in \mathbb{N}$ ($i \leq j \leq |\sigma|$) the substring between i and j of σ is the

Exp ::= **Id** | v $\in \mathbb{V}$ | **Exp** + **Exp** | **Exp** - **Exp** | **Exp** * **Exp** | **Exp** / **Exp** | **Exp** && **Exp**
| **Exp** || **Exp** | ! **Exp** | **Exp** > **Exp** | **Exp** < **Exp** | **Exp** == **Exp** | **Exp**.`substring`(**Exp**,**Exp**)
| **Exp**.`charAt`(**Exp**) | **Exp**.`indexOf`(**Exp**) | **Exp**.`length`

Block ::= { } | { **Stmt** }

Stmt ::= **Id** = **Exp**; | `if` (**Exp**) **Block** `else` **Block** | `while` (**Exp**) **Block** | **Block** | **Stmt Stmt** | ;

Figure 2: IMP syntax

string $\sigma_i \cdots \sigma_{j-1}$, and we denote it by *substring(σ,i,j)*. Let $\mathbb{Z}$ be the set of integers. We denote by $\Sigma^*_{\mathbb{Z}} \overset{\text{def}}{=} \{+,-,\epsilon\} \cdot \{0,1,\ldots,9\}^+$ the set of *numeric strings*, i.e., strings corresponding to integers. $\mathcal{I} : \Sigma^*_{\mathbb{Z}} \to \mathbb{Z}$ maps numeric strings to the corresponding integers. Dually, we define the function $\mathcal{S} : \mathbb{Z} \to \Sigma^*_{\mathbb{Z}}$ that maps each integer to its numeric string representation (e.g., 1 is mapped to the string `"1"`, not `"+1"`, -5 is mapped to `"-5"`).

Regular languages and finite state automata. We follow [29] for automata notation. A finite state automaton (FA) is a tuple $\mathtt{A} = (Q, q_0, \Sigma, \delta, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is the set of final states. In particular, if $\delta : Q \times \Sigma \to Q$ is a function then $\mathtt{A}$ is called deterministic FA (DFA).[1] The class of languages recognized by FAs is the class of regular languages. We denote the set of all DFAs as DFA. Given an automaton $\mathtt{A}$, we denote the language accepted by $\mathtt{A}$ as $\mathscr{L}(\mathtt{A})$. A language $\mathtt{L}$ is regular iff there exists a FA $\mathtt{A}$ such that $\mathtt{L} = \mathscr{L}(\mathtt{A})$. From the Myhill-Nerode theorem [20], for each regular language there uniquely exists a minimum automaton, i.e., with the minimum number of states, recognizing the language. Given a regular language $\mathtt{L}$, we denote by $\mathsf{Min}(\mathtt{L})$ the minimum DFA $\mathtt{A}$ s.t. $\mathtt{L} = \mathscr{L}(\mathtt{A})$.

The programming language. We consider an IMP language (Fig. 2) that contains representative string operations taken from the set of methods offered by the JavaScript built-in class `String` [41]. Other JavaScript string operations can be modeled by composition of the given string operations or as particular cases of them. Primitive values are $\mathbb{V} = \mathbb{S} \cup \mathbb{Z} \cup \mathbb{B} \cup \{\mathtt{NaN}\}$ with $\mathbb{S} \overset{\text{def}}{=} \Sigma^*$ (strings on the alphabet Σ), $\mathbb{B} \overset{\text{def}}{=} \{\mathtt{true}, \mathtt{false}\}$ and `NaN` a special value denoting not-a-number.

Implicit type conversion. In order to capture the semantics of the language IMP, inspired by the JavaScript semantics, we need to deal with *implicit type conversion* [4]. For each primitive value, we define an auxiliary function converting primitive values to other primitive values (Fig. 3). Note that all the functions behave like the identity when applied to values not needing conversion, e.g., `toInt` on integers. Then, $\mathtt{toStr} : \mathbb{V} \to \mathbb{S}$ maps any input value to its string representation; $\mathtt{toInt} : \mathbb{V} \to \mathbb{Z} \cup \{\mathtt{NaN}\}$ returns the integer corresponding to a value, when it is possible: For `true` and `false` it returns respectively 1 and 0, for strings in $\Sigma^*_{\mathbb{Z}}$ it returns the corresponding integer, while all the other values are converted to `NaN`. For instance, `toInt`("42") = 42, `toInt`("42*hello*") = `NaN`. Finally, $\mathtt{toBool} : \mathbb{V} \to \mathbb{B}$ returns `false` when the input is 0, and `true` for all the other non boolean primitive values. For example, implicit type conversion is applied when the guards of `while` and `if` statements do not evaluate to booleans (e.g., `while (1)` {x=x+1;}, the guard is implicitly converted to `true`).

Semantics. Program states are partial maps from identifiers to primitive values, i.e., STATES : **Id** $\to \mathbb{V}$. The concrete big-step semantics $[\![\cdot]\!]$: **Stmt** $\times$ STATES $\to$ STATES follows [4], and it

[1] We consider DFA also those FAs which are not complete, namely such that a transition for each pair (q,a) $(q \in Q,\ a \in \Sigma)$ does not exists. They can be easily transformed in a DFA by adding a sink state receiving all the missing transitions.

$$\texttt{toStr}(v) = \begin{cases} v & v \in \mathbb{S} \\ \text{“}\texttt{NaN}\text{”} & v = \texttt{NaN} \\ \text{“}\texttt{true}\text{”} & v = \texttt{true} \\ \text{“}\texttt{false}\text{”} & v = \texttt{false} \\ \mathcal{S}(v) & v \in \mathbb{Z} \end{cases} \quad \texttt{toInt}(v) = \begin{cases} v & v \in \mathbb{Z} \\ 1 & v = \texttt{true} \\ 0 & v = \texttt{false} \vee v = \texttt{NaN} \\ \mathcal{I}(v) & v \in \mathbb{S} \wedge v \in \Sigma^*_{\mathbb{Z}} \\ \texttt{NaN} & v \in \mathbb{S} \wedge v \notin \Sigma^*_{\mathbb{Z}} \end{cases} \quad \texttt{toBool}(v) = \begin{cases} v & v \in \mathbb{B} \\ \texttt{true} & v \in \mathbb{Z} \smallsetminus \{0\} \vee v \in \mathbb{S} \smallsetminus \{\epsilon\} \\ \texttt{false} & v = 0 \vee v = \epsilon \vee v = \texttt{NaN} \end{cases}$$

Figure 3: IMP implicit type conversion functions.

includes dynamic typing and implicit type conversion. Also the expression semantics, $[\![\cdot]\!] : \mathbf{Exp} \times \textsc{States} \to \mathbb{V}$, is standard; we only provide the formal and precise semantics of the IMP string operations. Let $\sigma, \sigma' \in \mathbb{S}$ and $i, j \in \mathbb{Z}$ (values which are not strings or numbers respectively, are converted by the implicit type conversion primitives. Negative values are treated as zero).

`substring`: It extracts substrings from strings, i.e., all the characters between two indexes. The semantics is the function $\textsc{Ss}\colon \mathbb{S} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{S}$ defined as:

$$\textsc{Ss}(\sigma, i, j) \overset{\text{def}}{=} \begin{cases} \textsc{Ss}(\sigma, j, i) & j < i \\ \textit{substring}(\sigma, i, \mathsf{max}(j, |\sigma|)) & \text{otherwise} \end{cases}$$

`charAt`: It returns the character at a specified index. The semantics is the function $\textsc{Ca}\colon \mathbb{S} \times \mathbb{Z} \to \mathbb{S}$ defined as follows:

$$\textsc{Ca}(\sigma, i) \overset{\text{def}}{=} \begin{cases} \sigma_i & 0 \le i < |\sigma| \\ \epsilon & \text{otherwise} \end{cases}$$

`indexOf`: It returns the position of the first occurrence of a given substring. The semantics is the function $\textsc{Io}\colon \mathbb{S} \times \mathbb{S} \to \mathbb{Z}$ defined as follows:

$$\textsc{Io}(\sigma, \sigma') \overset{\text{def}}{=} \begin{cases} \mathsf{min}\{\, i \mid \sigma_i \ldots \sigma_j = \sigma' \,\} & \exists i, j.\, \sigma_i \ldots \sigma_j = \sigma' \\ -1 & \text{otherwise} \end{cases}$$

`length`: It returns the length of a string $\sigma \in \mathbb{S}$. Its semantics is the function $\textsc{Le}\colon \mathbb{S} \to \mathbb{Z}$ defined as $\textsc{Le}(\sigma) \overset{\text{def}}{=} |\sigma|$.

`concat`: The string concatenation is handled by IMP plus operator (+). The concrete semantics relies on the concatenation operator reported in Sect. 2, i.e., $\textsc{Cc}(\sigma, \sigma') = \sigma\sigma'$.

2.2 The finite state automata domain for strings

In this section, we describe the automata abstract domain for strings [11, 36, 43], namely the domain of regular languages over Σ^*. In particular, our aim is that of characterize automata as a domain for abstracting the computation of program semantics in the abstract interpretation framework. The exploited idea is that of approximating strings as regular languages represented by the minimum DFAs [20] recognizing them. In general, we have more DFAs that recognize a regular language, hence the domain of automata is indeed the quotient $\textsc{Dfa}_{/\equiv}$ w.r.t. the equivalence relation induced by language equality: $\forall \mathtt{A}_1, \mathtt{A}_2 \in \textsc{Dfa}.\, \mathtt{A}_1 \equiv \mathtt{A}_2 \Leftrightarrow \mathscr{L}(\mathtt{A}_1) = \mathscr{L}(\mathtt{A}_2)$. Hence, any equivalence class $[\mathtt{A}]_\equiv$ is composed by the automata that recognize the same regular language. We abuse notation by representing equivalence classes in the domain $\textsc{Dfa}_{/\equiv}$ w.r.t. $\equiv$ by one of its automata (usually the minimum), i.e., when we write $\mathtt{A} \in \textsc{Dfa}_{/\equiv}$ we mean $[\mathtt{A}]_\equiv$. The partial order $\sqsubseteq_{\textsc{Dfa}}$ induced by language inclusion is $\forall \mathtt{A}_1, \mathtt{A}_2 \in \textsc{Dfa}_{/\equiv} .\, \mathtt{A}_1 \sqsubseteq_{\textsc{Dfa}} \mathtt{A}_2 \Leftrightarrow \mathscr{L}(\mathtt{A}_1) \subseteq \mathscr{L}(\mathtt{A}_2)$, which is well defined since automata in the same $\equiv$-equivalence class recognize the same language.

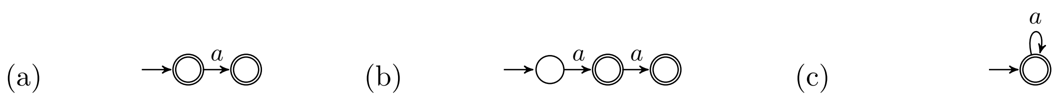


Figure 4: (a) $\mathtt{A}_1$ s.t. $\mathscr{L}(\mathtt{A}_1) = \{\epsilon, a\}$ (b)$\mathtt{A}_2$ s.t. $\mathscr{L}(\mathtt{A}_2) = \{a, aa\}$ (c) $\mathtt{A}_1 \nabla_1 \mathtt{A}_2$

The least upper bound (lub) $\sqcup_{\text{DFA}} : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \to \text{DFA}_{/\equiv}$ on the domain $\text{DFA}_{/\equiv}$, corresponds to the standard union between automata: $\forall \mathtt{A}_1, \mathtt{A}_2 \in \text{DFA}_{/\equiv}.\ \mathtt{A}_1 \sqcup_{\text{DFA}} \mathtt{A}_2 \overset{\text{def}}{=} \mathsf{Min}(\mathscr{L}(\mathtt{A}_1) \cup \mathscr{L}(\mathtt{A}_2))$. It is the minimum automaton recognizing the union of the languages $\mathscr{L}(\mathtt{A}_1)$ and $\mathscr{L}(\mathtt{A}_2)$. This is a well-defined notion since regular languages are closed under union. The greatest lower bound $\sqcap_{\text{DFA}} : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \to \text{DFA}_{/\equiv}$ corresponds to automata intersection, since regular languages are closed under finite intersection: $\forall \mathtt{A}_1, \mathtt{A}_2 \in \text{DFA}_{/\equiv}.\ \mathtt{A}_1 \sqcap_{\text{DFA}} \mathtt{A}_2 \overset{\text{def}}{=} \mathsf{Min}(\mathscr{L}(\mathtt{A}_1) \cap \mathscr{L}(\mathtt{A}_2))$.

Theorem 1. $\langle \text{DFA}_{/\equiv}, \sqsubseteq_{\text{DFA}}, \sqcup_{\text{DFA}}, \sqcap_{\text{DFA}}, \mathsf{Min}(\varnothing), \mathsf{Min}(\Sigma^*)\rangle$ *is a sub-lattice but not a complete meet-sub-semilattice of* $\wp(\Sigma^*)$.

In other words, there exists no Galois connections between $\text{DFA}_{/\equiv}$ and $\wp(\Sigma^*)$, i.e., there may exist no minimal automaton abstracting a language.[2] However, this is not a concern, since the relation between concrete semantics and abstract semantics can be weakened while still ensuring soundness [16]. A well known example is the convex polyhedra domain [19].

Widening. The domain $\text{DFA}_{/\equiv}$ is an infinite domain, and it is not ACC, i.e., it contains infinite ascending chains. For instance, consider the set of languages $\{\{\ a^j b^j \mid 0 \leq j \leq i\ \}\}_{i \geq 0} \subseteq \wp(\Sigma^*)$ forming an infinite ascending chain, then also the set of the corresponding minimal automata forms an ascending chain on $\text{DFA}_{/\equiv}$. This clearly implies that any computation on $\text{DFA}_{/\equiv}$ may lose convergence [16]. Most of the proposed abstract domains for strings [13, 30, 32, 33] trivially satisfy ACC by being finite, but they may lose precision during the abstract computation [17]. In these cases, domains must be equipped with a widening operator approximating the lub in order to force convergence (by necessarily losing precision) for any increasing chain [17]. As far as automata are concerned, existing widenings are defined in terms of a state equivalence relation merging states that recognize the same language, up to a fixed length n (set as parameter for tuning the widening precision) [6, 22]. We denote this parametric widening with ∇_n, $n \in \mathbb{N}$ [22].

Example 1. *Consider the following* IMP *fragment*

```
str = ""; while (x++ < 100) { str += "a"; }
```

Since the value of the variable `x` *is unknown, also the number of iterations of the* `while`*-loop is unknown. In these cases, in order to guarantee soundness and termination, we apply the widening operator. In Fig. 4a we report the abstract value of the variable* `str` *at the beginning of the second iteration of the loop, while in Fig. 4b the abstract value of the variable* `str` *at the end of the second iteration is reported. Before starting a new iteration, in the example, we apply* ∇_1 *between two automata, namely we merge all the states having the same outgoing character. The minimization of the obtained automaton is reported in Fig. 4c. The next iteration will reach the fix-point, guaranteeing soundness and termination.*

3 An abstract domain for string manipulation

In this section, we discuss how to design an abstract domain for string manipulation dealing also with other primitive types, namely able to combine different abstractions of different primitive

[2]Note that, some works have studied automatic procedures to compute, given an input language L, the *regular cover* of L [21] (i.e., an automaton containing the language L). In particular, [10, 21] have studied regular covers guaranteeing that the automaton obtained is the best w.r.t. a *minimal relation* (but not minimum).

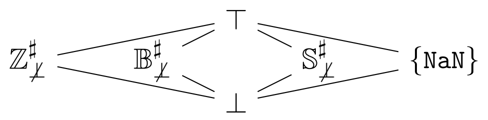

$\top$
$\mathbb{Z}^\sharp_{\not\perp}$
$\mathbb{B}^\sharp_{\not\perp}$
$\mathbb{S}^\sharp_{\not\perp}$
$\{\mathtt{NaN}\}$
$\perp$

Figure 5: Coalesced sum abstract domain for IMP

types. In particular, since operations on strings combine strings also with other values (e.g., integers), an abstract domain for string analysis equipped with dynamic typing must include all the possible primitive values, i.e., the whole $\mathbb{V} = \mathbb{Z} \cup \mathbb{B} \cup \mathbb{S} \cup \{\mathtt{NaN}\}$. The idea is to consider an abstract domain for each type of primitive value and to combine these abstract domains in a unique abstract domain for $\mathbb{V}$. Consider, for each primitive value $\mathbb{D}$, an abstract domain $\mathbb{D}^\sharp$ (we denote the domain $\mathbb{D}^\sharp$ without bottom as $\mathbb{D}^\sharp_{\not\perp}$), equipped with an abstraction $\alpha_{\mathbb{D}} : \mathbb{D} \to \mathbb{D}^\sharp$ and a concretization $\gamma_{\mathbb{D}} : \mathbb{D}^\sharp \to \mathbb{D}$ forming a Galois insertion [15].

Coalesced sum. One way to merge domains is the *coalesced sum* [14]. The resulting domain contains all the non-bottom elements of the domains, together with a new top and a new bottom, covering all the elements and covered by all the elements, respectively. In our case, if we consider the abstract domains $\mathbb{Z}^\sharp$, $\mathbb{S}^\sharp$ and $\mathbb{B}^\sharp$, the coalesced sum is the abstraction of $\wp(\mathbb{V})$ depicted in Fig. 5. This is the simplest choice, but unfortunately this is not suitable for dynamic languages, and in particular for dealing with dynamic typing and implicit type conversion. The problem is that the type of variables is inferred at run-time and may change during execution. For example, consider the following IMP fragment: $\mathtt{if}\ (\mathtt{y} < 5)\ \{\mathtt{x} = \text{“}\mathtt{42}\text{”};\}\ \mathtt{else}\ \{\mathtt{x} = \mathtt{true};\}$. The value of the variable `y` is statically unknown hence, in order to guarantee soundness, we must take into account both the branches, meaning that `x` may be both a string and a boolean value, after the `if` statement. On the coalesced sum domain, the analysis would lose any precision w.r.t. collecting semantics by returning $\alpha_{\mathbb{S}}(\text{“}\mathtt{42}\text{”}) \sqcup \alpha_{\mathbb{B}}(\mathtt{true}) = \top$.

Cartesian product. In order to catch union types, without losing too much precision, we need to *complete* [24–26] the above domain in order to observe collections of values of different types. In order to define this combination, we rely on the cartesian product, following [23]. Hence, the complete abstract domain w.r.t. dynamic typing and implicit type conversion is: $\mathbb{Z}^\sharp \times \mathbb{B}^\sharp \times \mathbb{S}^\sharp \times \wp(\{\mathtt{NaN}\})$, abstraction of $\wp(\mathbb{V})$. In this combining abstract domain, the value of `x` after the `if`-execution is precisely $(\perp, \alpha_{\mathbb{B}}(\mathtt{true}), \alpha_{\mathbb{S}}(\text{“}\mathtt{42}\text{”}), \perp)$, now an element of the domain, inferring that the value of `x` can be $\alpha_{\mathbb{B}}(\mathtt{true})$ or $\alpha_{\mathbb{S}}(\text{“}\mathtt{42}\text{”})$, but definitely not an abstract integer or `NaN`.

In the following, we consider the abstract domain $\mathbb{V}^\sharp$ for string analysis obtained as cartesian product of the following abstractions: $\mathbb{Z}^\sharp = \mathsf{Int}$ (the well-known abstract domain of intervals [15]), $\mathbb{S}^\sharp = \textsc{Dfa}_{/\equiv}$, $\mathbb{B}^\sharp = \wp(\{\mathtt{true}, \mathtt{false}\})$.

4 The IMP abstract semantics

In this section, we define the abstract semantics of the language IMP on the abstract domain $\mathbb{V}^\sharp$. In particular, we have to define the expressions abstract semantics $[\![\cdot]\!]^\sharp : \mathbf{Exp} \times \textsc{States} \to \mathbb{V}^\sharp$, which is standard except for the string operations that will be explicitly provided by describing the algorithm for computing them. Let us first recall some important notions on regular languages, useful for the algorithms we will provide.

Definition 1 (Suffixes and prefixes [20])**.** *Let* $\mathtt{L} \in \wp(\Sigma^*)$ *be a regular language. The suffixes of* $\mathtt{L}$ *are* $\textsc{Su}(\mathtt{L}) \overset{def}{=} \{\, y \in \Sigma^* \mid \exists x \in \Sigma^*. xy \in \mathtt{L} \,\}$*, and the prefixes of* $\mathtt{L}$ *are* $\textsc{Pr}(\mathtt{L}) \overset{def}{=} \{\, x \in \Sigma^* \mid \exists y \in \Sigma^*. xy \in \mathtt{L} \,\}$*.*

We can define the suffixes from a position, namely given $i \in \mathbb{N}$, the set of suffixes from i is $\textsc{Su}(\mathtt{L}, i) \overset{\text{def}}{=} \{\, y \in \Sigma^* \mid \exists x \in \Sigma^*.\, xy \in \mathtt{L}, |x| = i \,\}$. For instance, let $\mathtt{L} = \{abc, hello\}$, then $\textsc{Su}(\mathtt{L}, 2) = \{c, llo\}$.

Definition 2 (Right quotient [20])**.** *Let* $\mathtt{L}_1, \mathtt{L}_2 \in \Sigma^*$ *be regular languages. The right quotient of* $\mathtt{L}_1$ *w.r.t.* $\mathtt{L}_2$ *is* $\textsc{Rq}(\mathtt{L}_1, \mathtt{L}_2) \overset{\text{def}}{=} \{\, x \in \Sigma^* \mid \exists y \in \mathtt{L}_2.\, xy \in \mathtt{L}_1 \,\}$.

For example, let $\mathtt{L}_1 = \{xab, yab\}$ and $\mathtt{L}_2 = \{b, ab\}$. The right quotient of $\mathtt{L}_1$ w.r.t. $\mathtt{L}_2$ is $\textsc{Rq}(\mathtt{L}_1, \mathtt{L}_2) = \{xa, ya, x, y\}$.

Definition 3 (Factors [7])**.** *Let* $\mathtt{L} \in \wp(\Sigma^*)$ *be a regular language. The set of its factors is* $\textsc{Fa}(\mathtt{L}) \overset{\text{def}}{=} \{\, y \in \Sigma^* \mid \exists x, z \in \Sigma^*.\, xyz \in \mathtt{L} \,\}$.

These operations are all defined as transformations of regular languages. In [20] the corresponding algorithms on FA are provided. In particular, let $\mathtt{A}, \mathtt{A}_1 \in \textsc{Dfa}_{/\equiv}$ and $i \in \mathbb{N}$, then $\mathsf{SU}(\mathtt{A})$, $\mathsf{PR}(\mathtt{A})$, $\mathsf{SU}(\mathtt{A}, i)$, $\mathsf{FA}(\mathtt{A})$ and $\mathsf{RQ}(\mathtt{A}, \mathtt{A}_1)$ are the algorithms corresponding to the transformations $\textsc{Su}(\mathtt{L}(\mathtt{A}))$, $\textsc{Pr}(\mathtt{L}(\mathtt{A}))$, $\textsc{Su}(\mathscr{L}(\mathtt{A}), i)$, $\textsc{Fa}(\mathscr{L}(\mathtt{A}))$ and $\textsc{Rq}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}_1))$, respectively. Namely, $\forall \mathtt{A}, \mathtt{A}_1 \in \textsc{Dfa}_{/\equiv}$, $i \in \mathbb{N}$, the following facts hold:

$$\begin{gathered}\textsc{Su}(\mathscr{L}(\mathtt{A})) = \mathscr{L}(\mathsf{SU}(\mathtt{A})),\ \textsc{Pr}(\mathscr{L}(\mathtt{A})) = \mathscr{L}(\mathsf{PR}(\mathtt{A})),\ \textsc{Fa}(\mathscr{L}(\mathtt{A})) = \mathscr{L}(\mathsf{FA}(\mathtt{A})) \\ \textsc{Rq}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}_1)) = \mathscr{L}(\mathsf{RQ}(\mathtt{A}, \mathtt{A}_1)),\ \textsc{Su}(\mathscr{L}(\mathtt{A}), i) = \mathscr{L}(\mathsf{SU}(\mathtt{A}, i))\end{gathered}$$

As far as (state) complexity is concerned [44], prefix and right quotient operations have linear complexity, while suffix and factor operations, in general, are exponential [39, 44].

4.1 Abstract semantics of `substring`

In this section, we define the abstract semantics of `substring`, i.e., we define the operator $\mathsf{SS}^\sharp : \textsc{Dfa}_{/\equiv} \times \mathsf{Int} \times \mathsf{Int} \to \textsc{Dfa}_{/\equiv}$, starting from an automaton, an interval $[i, j]$ of initial indexes and an interval $[l, k]$ of final indexes for substrings, and computing the automaton recognizing the set of all substrings of the input automata language between the indexes in the two intervals. Hence, since the abstract semantics has to take into account the swaps when the initial index is greater than the final one, several cases arise handling (potentially unbounded) intervals. Tab. 1 reports the abstract semantics of $\mathsf{SS}^\sharp$ when $i, j \leq l$ (hence $i \leq k$). The definition of this semantics is by recursion with four base cases (the other cases are recursive calls splitting and rewriting the input intervals in order to match or to get closer to base cases) for which we describe the algorithmic characterization. Consider $\mathtt{A} \in \textsc{Dfa}_{/\equiv}$, $i, l \in \mathbb{Z} \cup \{-\infty\}$, $j, k \in \mathbb{Z} \cup \{+\infty\}$ (for the sake of readability we denote by $\sqcup$ the automata lub $\sqcup_{\textsc{Dfa}}$, and by $\sqcap$ the glb $\sqcap_{\textsc{Dfa}}$), the base cases are

1. If $i, j, l, k \in \mathbb{Z}$ (first row, first column of Tab. 1) we have to compute the language of all the substrings between an initial index in $[i, j]$ and a final index in $[l, k]$, i.e., $\textsc{Ss}(\mathscr{L}(\mathtt{A}), [i, j], [l, k])$[3]. For example, let $\mathtt{L} = \{a\}^* \cup \{hello, bc\}$, the set of its substrings from 1 to 3 is $\textsc{Ss}(\mathtt{L}, [1, 1], [3, 3])$ $= \{\epsilon, a, aa, el, c\}$. The automaton accepting this language is computed by the operator

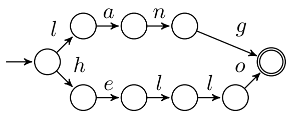
$$\mathsf{SS}(\mathtt{A}, [i, j], [l, k]) \overset{\text{def}}{=} \bigsqcup_{a \in [i, j], b \in [l, k]} (\mathsf{RQ}(\mathsf{SU}(\mathtt{A}, a), \mathsf{SU}(\mathtt{A}, b)) \sqcap \mathsf{Min}(\Sigma^{b-a})) \sqcup (\mathsf{SU}(\mathtt{A}, a)) \sqcap \mathsf{Min}(\Sigma^{<b-a})$$

2. When both intervals correspond to $[-\infty, +\infty]$, the result is the automaton of all possible factors of $\mathtt{A}$ (last row, last column), i.e., $\mathsf{FA}(\mathtt{A})$;

[3] We abuse notation by denoting with $\textsc{Ss}$ also the additive lift to languages and to sets of indexes: $\textsc{Ss} : \wp(\Sigma^*) \times \wp(\mathbb{Z}) \times \wp(\mathbb{Z}) \to \wp(\Sigma^*)$ defined as $\textsc{Ss}(\mathtt{L}, I, J) = \{\, \textsc{Ss}(\mathtt{L}, i, j) \mid i \in I, j \in J \,\} = \{\, \textsc{Ss}(\sigma, i, j) \mid \sigma, \in \mathtt{L}, i \in I, j \in J \,\}$.

$\mathsf{SS}^\sharp(\mathtt{A},[i,j],[l,k])$ $i,j \le l$ $(i \le k)$	$l,k \in \mathbb{Z}$	$l=-\infty,\ k \in \mathbb{Z}$	$l \in \mathbb{Z},\ k=+\infty$	$l=-\infty,\ k=+\infty$
$i,j \in \mathbb{Z}$	$\mathsf{SS}(\mathtt{A},[i,j],[l,k])$	$\mathsf{SS}^\sharp(\mathtt{A},[i,j],[0,k])$	$\mathsf{SS}^\rightarrow(\mathtt{A},[i,j],l)$	$\mathsf{SS}^\sharp(\mathtt{A},[i,j],[0,+\infty])$
$i=-\infty,\ j \in \mathbb{Z}$	$\mathsf{SS}^\sharp(\mathtt{A},[0,j],[l,k])$	$\mathsf{SS}^\sharp(\mathtt{A},[0,j],[0,k])$	$\mathsf{SS}^\sharp(\mathtt{A},[0,j],[l,+\infty])$	$\mathsf{SS}^\sharp(\mathtt{A},[0,j],[0,+\infty])$
$i \in \mathbb{Z},\ j=+\infty$	$\mathsf{SS}^\sharp(\mathtt{A},[l,k],[k,+\infty])$ $\sqcup \mathsf{SS}^\sharp(\mathtt{A},[i,k],[l,k])$	$\mathsf{SS}^\sharp(\mathtt{A},[i,+\infty],[0,k])$	$\mathsf{SS}^\rightarrow(\mathtt{A},[i,l],l) \sqcup \mathsf{SS}^\leftrightarrow(\mathtt{A},l)$	$\mathsf{SS}^\sharp(\mathtt{A},[i,+\infty],[0,+\infty])$
$i=-\infty,\ j=+\infty$	$\mathsf{SS}^\sharp(A,[0,+\infty],[l,k])$	$\mathsf{SS}^\sharp(\mathtt{A},[0,+\infty],[0,k])$	$\mathsf{SS}^\sharp(\mathtt{A},[0,+\infty],[l,+\infty])$	$\mathsf{FA}(\mathtt{A})$

Table 1: Definition of $\mathsf{SS}^\sharp$ when $i,j \le l$ (and thus $i \le k$)

(a)


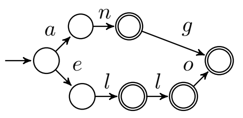
(b)


Figure 6: (a) $\mathtt{A}$, $\mathscr{L}(\mathtt{A}) = \{lang, hello\}$. (b) $\mathtt{A}' = \mathsf{SS}^\sharp(\mathtt{A},[1,1],[3,+\infty])$, $\mathscr{L}(\mathtt{A}') = \{an, ang, el, ell, ello\}$.

3. If $[i,j]$ is defined and the interval of final indexes is unbounded, i.e., $[l,+\infty]$ (first row, third column), we have to compute the automaton recognizing the following language

$$\mathrm{Ss}^\rightarrow(\mathscr{L}(\mathtt{A}),[i,j],l) \overset{\text{def}}{=} \bigcup_{a \in [i,j]} \{\, \mathrm{Ss}(\sigma,a,k) \mid \sigma \in \mathscr{L}(\mathtt{A}),\ k \ge l \,\}$$

   i.e., all the strings between a finite interval of initial indexes and an unbounded final index. The automaton accepting this language is computed by

$$\mathsf{SS}^\rightarrow(\mathtt{A},[i,j],l) \overset{\text{def}}{=} \bigsqcup_{a \in [i,j]} \mathsf{RQ}(\mathsf{SU}(\mathtt{A},a),\mathsf{SU}(\mathsf{SU}(\mathtt{A},l)))$$

   The abstract semantics returns the least upper bound of all the automata of substrings from a in $[i,j]$ to an unbounded index greater than or equal to l;
4. When both intervals are unbounded ($[i,+\infty]$ and $[l,+\infty]$, third row, third column of Tab. 1), we split the language to accept. In particular, we compute the substrings between $[i,l]$ and $[l+\infty]$ (and this has been considered in case 3), and the automaton recognizing the language of all substrings with both initial and final index with any value greater than l, i.e., the language $\mathrm{Ss}^\leftrightarrow(\mathscr{L}(\mathtt{A}),l) \overset{\text{def}}{=} \{\, \mathrm{Ss}(\sigma,a,b) \mid \sigma \in \mathscr{L}(\mathtt{A}),\ a,b \ge l \,\}$. This latter set is computed by the algorithm $\mathsf{SS}^\leftrightarrow(\mathtt{A},l) \overset{\text{def}}{=} \mathsf{FA}(\mathsf{SU}(\mathtt{A},l))$

Here we show the table only for the case $i,j \le l$ (and thus $i \le k$). Only few cases are not considered and they are not reported for space limitations. Anyway, they are amenable to Tab. 1. In Fig. 6 we report an example obtained applying the rules in the tables.

Theorem 2 (Termination of $\mathsf{SS}^\sharp$)**.** *For each* $\mathtt{A} \in \mathrm{DFA}_{/\equiv}, I, J \in \mathsf{Int}.\, \mathsf{SS}^\sharp(\mathtt{A},I,J)$ *performs at most three recursive calls, before reaching a base case.*

Theorem 3. $\mathsf{SS}^\sharp$ *is sound and complete:* $\forall \mathtt{A} \in \mathrm{DFA}_{/\equiv}, I, J \in \mathsf{Int}.\, \mathrm{Ss}(\mathscr{L}(\mathtt{A}),I,J) = \mathscr{L}(\mathsf{SS}^\sharp(\mathtt{A},I,J))$.

4.2 Abstract semantics of `charAt`

The abstract semantics of `charAt` should return the automaton accepting the language of all the characters of strings accepted by an automaton $\mathtt{A}$, in a position inside a given interval $[i,j]$: This

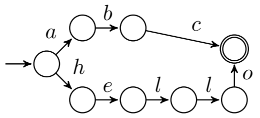
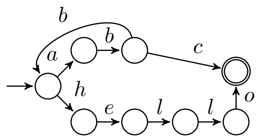
(a) 
(b) 

Figure 7: (a) $\mathtt{A}_1$, $\mathscr{L}(\mathtt{A}_1) = \{abc, hello\}$. (b) $\mathtt{A}_2$, $\mathscr{L}(\mathtt{A}_2) = \{abc, hello\} \cup \{\, (abb)^n c \mid n > 0 \,\}$.

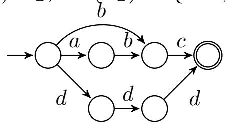
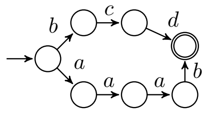
(a) 
(b) 

Figure 8: (a) $\mathtt{A}$, $\mathscr{L}(\mathtt{A}) = \{ddd, abc, bc\}$. (b) $\mathtt{A}''$, $\mathscr{L}(\mathtt{A}'') = \{bcd, aaab\}$

is computed by $\mathsf{CA}^\sharp : \text{DFA}_{/\equiv} \times \mathsf{Int} \to \text{DFA}_{/\equiv}$

$$\mathsf{CA}^\sharp(\mathtt{A}, [l,h]) \overset{\text{def}}{=} \begin{cases} \bigsqcup_{i \in [l,h]} \mathsf{SS}(\mathtt{A}, [i,i], [i+1, i+1]) & l, h \in \mathbb{Z} \\ \mathsf{CA}^\sharp(\mathtt{A}, [0,h]) \sqcup \mathsf{Min}(\{\epsilon\}) & l = -\infty, h \in \mathbb{Z}, h \geq 0 \\ \mathsf{Min}(\{\epsilon\}) & l = -\infty, h \in \mathbb{Z}, h < 0 \\ \mathsf{Min}(\mathtt{chars}(\mathsf{SU}(\mathtt{A}, l))) \sqcup \mathsf{Min}(\{\epsilon\}) & l \in \mathbb{Z}, l \geq 0, h = +\infty \\ \mathsf{Min}(\mathtt{chars}(\mathtt{A})) \sqcup \mathsf{Min}(\{\epsilon\}) & l = -\infty \text{ or } l \in \mathbb{Z}, l < 0, h = +\infty \end{cases}$$

We call SS (defined before) when the interval index $[l,h]$ is finite. In the last two cases, we use the function $\mathtt{chars} : \text{DFA}_{/\equiv} \to \wp(\Sigma)$, returning the set of characters read in any transition of an automaton. When $l \in \mathbb{Z}, h = +\infty$, we return the characters starting from l together with $\mathsf{Min}(\{\epsilon\})$ while, when $l = -\infty$, we simply return the characters of the automaton together with $\mathsf{Min}(\{\epsilon\})$.

Theorem 4. $\mathsf{CA}^\sharp$ *is sound and complete:* $\forall \mathtt{A} \in \text{DFA}_{/\equiv}, I \in \mathsf{Int}$, $\text{CA}(\mathscr{L}(\mathtt{A}), I) = \mathscr{L}(\mathsf{CA}^\sharp(\mathtt{A}, I))$.[4]

4.3 Abstract semantics of `length`

The abstract semantics of `length` should return the interval of all the possible string lengths in an automaton, i.e., it is $\mathsf{LE}^\sharp : \text{DFA}_{/\equiv} \to \mathsf{Int}$ computed by Alg. 1, where $\mathsf{minPath}, \mathsf{maxPath} : \text{DFA}_{/\equiv} \times Q \times Q \to \wp(Q)$ return the minimum and the maximum paths between two states of the input automaton, respectively. $\mathsf{len} : \wp(Q) \to \mathbb{N}$ returns the size of a path, and $\mathsf{hasCycle} : \text{DFA}_{/\equiv} \to \{\mathtt{true}, \mathtt{false}\}$ checks whether the automaton contains cycles.
The idea is to compute the minimum and the maximum path reaching each final state in the automaton (in Fig. 7a, we obtain 3 and 5). Then, we abstract the set of lengths obtained so far into intervals (in the example, $[3,5]$). Problems arise when the automaton contains cycles. In this case, we simply return the undefined interval starting from the minimum path, to a final state, to $+\infty$. For example, in the automaton in Fig. 7b, the length interval is $[3, +\infty]$.

Theorem 5. $\mathsf{LE}^\sharp$ *is sound but not complete:* $\forall \mathtt{A} \in \text{DFA}_{/\equiv}$ $\text{LE}(\mathscr{L}(\mathtt{A})) \subset \mathsf{LE}^\sharp(\mathtt{A})$.

4.4 Abstract semantics of `indexOf`

The abstract semantics of `indexOf` is $\mathsf{IO}^\sharp : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \to \mathsf{Int}$ and should return the interval of any possible positions of strings in a language inside strings of another language. Consider for instance the automaton $\mathtt{A}$ in Fig. 8a and suppose to call $\mathsf{IO}^\sharp(\mathtt{A}, \mathtt{A}')$ where $\mathtt{A}' = \mathsf{Min}(\{bc\})$. The idea is that of building, for each state q in $\mathtt{A}$, the automaton $\mathtt{A}_q$ which is $\mathtt{A}$ where all the states are final

[4]In the following, for all the string semantics, we abuse notation for the additive lift to languages and intervals.

Algorithm 1: $\mathsf{LE}^\sharp : \mathrm{DFA}_{/\equiv} \rightarrow \mathsf{Int}$ alg.

```
    Input: $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$
    Output: $\mathsf{LE}^\sharp(\mathtt{A})$
 1  $\mathsf{P\_len} \leftarrow 0$; $\mathsf{p\_len} \leftarrow \infty$
 2  if $\mathsf{hasCycle}(\mathtt{A})$ then
 3      foreach $q_f \in F$ do
 4          $\mathsf{p} \leftarrow \mathsf{minPath}(\mathtt{A}, q_0, q_f)$;
 5          if $\mathsf{len(p)} < \mathsf{p\_len}$ then
              $\mathsf{p\_len} \leftarrow \mathsf{len(p)}$ ;
 6      end
 7      return $[\mathsf{p\_len}, +\infty]$;
 8  else
 9      foreach $q_f \in F$ do
10          $\mathsf{p} \leftarrow \mathsf{minPath}(\mathtt{A}, q_0, q_f)$;
11          $\mathsf{P} \leftarrow \mathsf{maxPath}(\mathtt{A}, q_0, q_f)$;
12          if $\mathsf{len(p)} < \mathsf{p\_len}$ then
              $\mathsf{p\_len} \leftarrow \mathsf{len(p)}$ ;
13          if $\mathsf{len(P)} > \mathsf{P\_len}$ then
              $\mathsf{P\_len} \leftarrow \mathsf{len(P)}$ ;
14      end
15      return $[\mathsf{p\_len}, \mathsf{P\_len}]$;
16  end
```

Algorithm 2: $\mathsf{IO}^\sharp : \mathrm{DFA}_{/\equiv} \times \mathrm{DFA}_{/\equiv} \rightarrow \mathsf{Int}$ alg.

```
    Input: $\mathtt{A} = (Q, \Sigma, \delta, q_0, F), \mathtt{A}' = (Q', \Sigma, \delta', q_0', F')$
    Output: $\mathsf{IO}^\sharp(\mathtt{A}, \mathtt{A}')$
 1  $\mathsf{indexesOf} \leftarrow \varnothing$
 2  foreach $q \in Q$ do
 3      $\mathtt{A}_q \leftarrow (Q, \Sigma, \delta, q, Q)$;
 4      if $\mathtt{A}_q \sqcap_{\mathrm{DFA}} \mathtt{A}' \neq \varnothing$ then
 5          $\mathsf{indexesOf} \leftarrow$
              $\mathsf{indexesOf} \cup \{\mathsf{len}(\mathsf{maxPath}(\mathtt{A}, q_0, q))\}$;
 6          if $\exists p = \mathsf{path}(q_0, q)$ s.t. $\mathsf{hasCycle}(p)$ then
 7              $\mathsf{indexesOf} \leftarrow \mathsf{indexesOf} \cup \{+\infty\}$
 8          end
 9      else
10          $\mathsf{indexesOf} \leftarrow \mathsf{indexesOf} \cup \{-1\}$;
11      end
12  end
13  if $|\mathscr{L}(\mathtt{A})| == |\mathscr{L}(\mathtt{A}')| == 1$ then
14      return $[\mathsf{min(indexesOf)}, \mathsf{min(indexesOf)}]$;
15  else
16      return $[\mathsf{min(indexesOf)}, \mathsf{max(indexesOf)}]$;
17  end
```

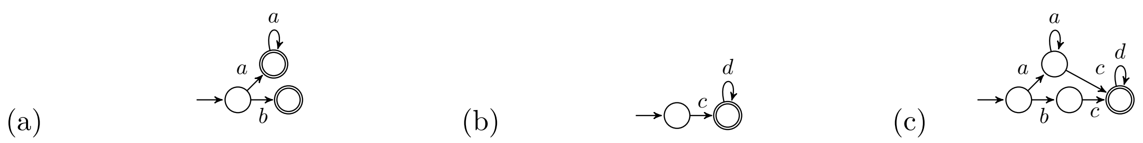


Figure 9: (a) $\mathtt{A}$, $\mathscr{L}(\mathtt{A}) = \{\, a^n \mid n > 0 \,\} \cup \{b\}$ (b) $\mathtt{A}'$, $\mathscr{L}(\mathtt{A}') = \{\, cd^n \mid n \in \mathbb{N} \,\}$ (c) $\mathtt{A}'' = \mathsf{CC}^\sharp(\mathtt{A}, \mathtt{A}')$

and the initial state is q. Hence, we check whether $\mathtt{A}_q \sqcap \mathtt{A}'$ is non empty and we collect the size of the maximum path from q_0 to q in $\mathtt{A}$. If there exists at least one state from which any string accepted by $\mathtt{A}'$ cannot be read, we collect -1. In the example, $\mathtt{A}_{q_0}$ adds $\{0\}$, $\mathtt{A}_{q_1}$ adds $\{1\}$, while all the other states add $\{-1\}$. Finally, we return the interval $[\mathsf{min}\{-1,1,0\}, \mathsf{max}\{-1,1,0\}] = [-1,1]$. The full algorithm is reported in Alg. 2.

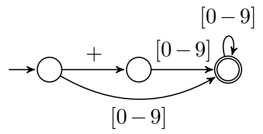
Theorem 6. $\mathsf{IO}^\sharp$ *is sound but not complete:* $\forall \mathtt{A}, \mathtt{A}' \in \mathrm{DFA}_{/\equiv} .\ \mathrm{IO}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) \subset \mathsf{IO}^\sharp(\mathtt{A}, \mathtt{A}')$.

As a counterexample to completeness, consider the automaton $\mathtt{A}''$ in Fig.8b: $\mathsf{IO}^\sharp(\mathtt{A}'', \mathsf{Min}(\{b\})) = [-1,3] \not\subset \mathrm{IO}(\mathscr{L}(\mathtt{A}''), \{b\}) = \{0,3\}$. The interval $[-1,3]$ contains also indexes where the string b is not recognized (e.g., 2), but it also contains the information (-1) meaning that there exists at least one accepted string without b as substring, which is not true.

4.5 Abstract semantics of concatenation

The abstract semantics of string concatenation is $\mathsf{CC}^\sharp : \mathrm{DFA}_{/\equiv} \times \mathrm{DFA}_{/\equiv} \rightarrow \mathrm{DFA}_{/\equiv}$ and returns the concatenation between two automata. Since regular languages are closed under concatenation, the property also holds on automata. In Fig. 9, we report an example of concatenation between two automata. Hence, $\mathsf{CC}^\sharp$ exactly implements the standard concatenation operation between automata. Given the closure property on automata, the following result holds.

Theorem 7. $\mathsf{CC}^\sharp$ *is sound and complete:* $\forall \mathtt{A}, \mathtt{A}' \in \mathrm{DFA}_{/\equiv} . \mathrm{CC}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \mathsf{CC}^\sharp(\mathtt{A}, \mathtt{A}')$.

(a) 

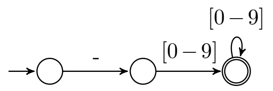
(b) 

Figure 10: (a) $\mathsf{toStr}^\sharp([0,+\infty])$. (b) $\mathsf{toStr}^\sharp([-\infty,0])$

4.6 Concerning abstract implicit type conversion

In this section, we discuss the abstraction of the implicit type conversion functions. For space limitations, we will focus only on the conversion of automata into other values, since the conversions concerning booleans, not-a-number and intervals are standard. Let $\texttt{toBool}^\sharp : \mathbb{V}^\sharp \rightarrow \mathbb{B}^\sharp$ be applied to $\mathtt{A} \in \textsc{Dfa}_{/\equiv}$: If $\mathtt{A} \sqcap \mathsf{Min}(\{\epsilon\}) = \varnothing$, it returns $\{\mathtt{true}\}$, when $\mathtt{A} = \mathsf{Min}(\{\epsilon\})$ the function returns $\{\mathtt{false}\}$, otherwise the function returns $\{\mathtt{true},\mathtt{false}\}$. Implicit type conversion to $\textsc{Dfa}_{/\equiv}$ is handled by the function $\texttt{toStr}^\sharp : \mathbb{V}^\sharp \rightarrow \textsc{Dfa}_{/\equiv}$. As far as non numeric strings are concerned, $\texttt{toStr}^\sharp$ returns $\mathsf{Min}(\{\mathtt{NaN}\})$. If the input is the boolean value $\mathtt{true}$ [$\mathtt{false}$] it returns $\mathsf{Min}(\{\mathtt{true}\})$ [$\mathsf{Min}(\{\mathtt{false}\})$], otherwise it returns $\mathsf{Min}(\{\mathtt{true}\}) \sqcup \mathsf{Min}(\{\mathtt{false}\})$. Converting intervals to FA is more tricky. If $l, h \in \mathbb{Z}$, the conversion to automata is simply $\bigsqcup_{i\in[l,h]} \mathsf{Min}(\{\mathcal{S}(i)\})$. The interval-to-automaton conversion for $[0,+\infty]$ and $[-\infty,0]$ are respectively shown in Fig. 10a and Fig. 10b. Other unbounded intervals, $[+l,+\infty]$ and $[-l,+\infty]$ $(l > 0)$, are converted in $\mathsf{toStr}^\sharp([0,+\infty]) \smallsetminus \mathsf{toStr}^\sharp([0,l])$ and $\mathsf{toStr}^\sharp([-l,0]) \sqcup \mathsf{toStr}^\sharp([0,+\infty])$, respectively. Conversions of intervals $[-\infty,l]$ and $[-\infty,-l]$ $(l > 0)$ are analogous, while, $\mathsf{toStr}^\sharp([-\infty,+\infty]) = \mathsf{Min}(\Sigma_\mathbb{Z})$. Finally, $\mathsf{toInt}^\sharp : \mathbb{V}^\sharp \rightarrow \mathsf{Int} \cup \{\mathtt{NaN}\}$ handles conversion to intervals. Given an automaton $\mathtt{A}$, if $\mathtt{A} \sqcap \mathsf{Min}(\Sigma_\mathbb{Z}) = \varnothing$, the automaton is precisely converted to $\mathtt{NaN}$, otherwise, if $\mathtt{A} \sqsubseteq_{\textsc{Dfa}} \mathsf{Min}(\Sigma_\mathbb{Z})$ it means that $\mathscr{L}(\mathtt{A})$ contains only numeric strings. For the sake of precision, we check whether $\mathtt{A}$ recognizes positive numeric strings (checking if the initial state reads only $+$ or number symbols), negative numeric strings (checking if the initial state reads only $-$ or 0 symbols) or both. In the first case, we return $[0,+\infty]$, in the second $[-\infty,0]$ and in the last $[-\infty,+\infty]$.

The abstract interpreter for the abstract semantics so far defined has been tested by means of the implementation of an automata library[5]. This library includes the implementation of all the algorithms concerning the finite state automata domain and provide well-known operations on automata such as suffix, right quotient, and abstract domain-related operations, such as $\sqcup_{\textsc{Dfa}}$, $\sqcap_{\textsc{Dfa}}$, and a parametric widening for tuning precision and forcing convergence. The library is suitable and easily pluggable into existing static analyzers, such as [30,32,33,37]. The bottleneck of our library is the determinization operation, having exponential complexity [29] (we rely on determinization in the minimization algorithm, in order to preserve the automata arising during the abstract computations minimum and deterministic). It is worth noting that, as reported in Thm. 1, $\wp(\Sigma^*)$ (string concrete domain) and $\textsc{Dfa}_{/\equiv}$ (abstract string domain) do not form a Galois connection but, nevertheless, this is not a concern. We have shown, for the core language we adopted, that the abstract semantics we have defined for string operations guarantee soundness hence, if the abstract interpreter starts from regular initial conditions (i.e., constraints expressible as finite state automata) it will always compute regular invariants. Indeed, it is sound to start from $\top$ initial condition that, in our string abstract domain, is expressible by $\mathsf{Min}(\wp(\Sigma^*))$, that it is regular.

Example: Obfuscated malware. Consider the fragment reported in Fig. 1 in the introduction. By computing the abstract semantics of this code, we obtain that the abstract value of `d`, at

[5]Available at `www.github.com/SPY-Lab/fsa` and the IMP static analyzer at `www.github.com/SPY-Lab/mu-js`

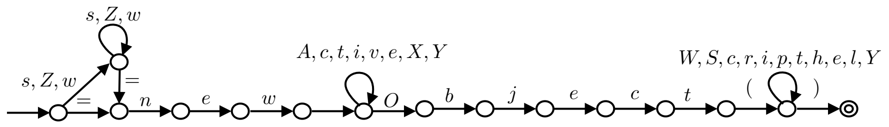


Figure 11: $\mathtt{A}_d$ abstract value of `d` before `eval` call of the program in Fig. 1

the `eval` call, is the automaton $\mathtt{A}_d$ in Fig. 11. The cycles are caused by the widening application in the `while` computation. From this automaton we are able to retrieve some important and non-trivial information. For example, we are able to answer to the following question: *May* $\mathtt{A}_d$ *contain a string corresponding to an assignment to an ActiveXObject?* We can simply answer by checking the predicate $\mathtt{A}_d \sqcap \mathsf{Min}(\mathbf{Id} \cdot \{new\ ActiveXObject(\} \cdot \Sigma^* \cdot \{)\}) \neq \varnothing$, checking whether $\mathtt{A}_d$ recognizes strings that are concatenations of any identifier with the string *new ActiveXObject*, followed by any possible string. In the example, the predicate returns `true`. Another interesting information could be: *May* $\mathtt{A}_d$ *contain* `eval` *string?* We can answer by checking whether $\mathtt{A}_d \sqcap \mathsf{Min}(\{eval\}) \neq \varnothing$, that is false and enforces that any explicit call to `eval` cannot occur.
We observe that such analysis may lose precision during fix-point computations, causing the cycles in the automaton in Fig. 11, due to the widening application. Nevertheless, it is worth noting that this result is obtained without any precision improvement on fix-point computations, such as loop unrolling or widening with thresholds. We think these analyses will drastically decrease false positives of the proposed string analysis but we will address this topic in future work.

5 Discussion and related work

In this paper, we have proposed an abstract semantics for a toy imperative language IMP, augmented with string manipulation operations, expressive enough to handle dynamic typing and implicit type conversion. In our abstract semantics, we have combined the DFA domain with abstract domains for the other primitive types, necessary to deal with static analysis of programs with dynamic typing. The proposed framework allows us to formally prove soundness and to study precision of the abstract semantics of each string operation: Depending on the property of interest, one can tune the degree of precision, namely the completeness of any string operation.
Main related work. The issue of analyzing strings is a widely studied problem, and it has been tackled in the literature from different points of view. Before discussing the most related works, we can observe what makes our approach original w.r.t. all the existing ones: (1) We provide a modular abstract domain parametric on the the abstractions of the different primitive types, this allows us both to obtain a tunable semantics precision and to handle dynamic typing for operation having both integer and string parameters, e.g., `substring`; (2) Our focus is on the characterization of a formal abstract interpretation-based framework where it is possible to prove soundness and to analyze completeness of string operations, in order to understand where it is possible to tune precision versus efficiency.
The main feature we have in common with existing works is the use of DFA (regular expressions) for abstracting strings. In [43], the authors propose symbolic string verifier for PHP based on finite state automata, represented by a particular form of binary decision diagrams, the MBDD. Even if it could be interesting to understand whether this representation of DFAs may be used also for improving our algorithms, their work only considers operations exclusively involving strings (not also integers such as `substring`) and therefore it provides a solution for different string manipulations. In [11], the authors propose an abstract interpretation-based string analyzer approximating strings into a subset of regular languages, called *regular strings* and they

define the abstract semantics for four string operations of interest together with a widening. This is the most related work, but our approach is strictly more general, since we do not introduce any restriction of regular languages and we abstract integers on intervals instead of on constants (meaning that our domain is strictly more precise). In [36], the authors propose a scalable static analysis for jQuery that relies on a novel abstract domain of regular expressions. The abstract domain in [36] contains the finite state automata domain but pursues a different task and does not provide semantics for string manipulations. Surely it may be interesting to integrate our library for string manipulation operators into SAFE. Finally, [35] proposes a generalization of regular expression, formally illustrating a parametric abstract domain of regular expressions starting from a complete lattice of reference. However, this work does not tackle the problem of analyzing string manipulations, since it instantiates the parametric abstract domain in the network communication environment, analyzing the exchanged messages as regular expressions. Finite state machines (transducer and automata) have found a critical application also in model checking both for enforcing string constraints and to model infinite transition systems [34]. For example, the authors of [1] define a sound decision procedure for a regular language-based logic for verification of string properties. The authors of [9] propose an automata abstraction in the context of regular model checking to tackle the well-known problem of state space explosion. Moreover, other formal systems, similar to DFA, have been proposed in the context of string analysis [2, 8, 28]. As future work, it can be interesting to study the relation between standard DFA and the other existing formal models, such as logics or other forms of FA. An interesting recent work is reported in [12], where the authors propose M-String, a parametric string abstract domain that extends the segmentation approach proposed in [18] for C strings. M-String uses an abstract domain for the content of a string and an abstract domain for expression, inferring when a string index position corresponds to an expression of the considered abstract domain. As future work, it could be interesting studying how to involve the finite state automata abstract domain into M-String, as abstraction of the string content.
In the context of JavaScript, several static analyzers have been proposed, pushed by the wide range of applications and the security issues related to the language [30,32,33,37]. TAJS [30] is a static analyzer based on abstract interpretation for JavaScript. The authors focus on allocation site abstraction, plugging in the static analyzer the *recency abstraction* [5], decreasing the number of false positives when objects are accessed. Upon TAJS, the authors have defined a sound way to statically analyze a large range of non-trivial `eval` patterns [31]. In [37], the authors define the Loop-Sensitive Analysis (LSA) that distinguishes loop iterations using *loop strings*, in the same way *call strings* distinguish function calls from different call sites in k-CFA [40]. The authors have implemented LSA into SAFE [33], a JavaScript web applications static analyzer. As future work, it may be interesting to combine LSA with our abstract semantics for decreasing the false positives introduced by the widening during fix-point computations. Finally, in [3], the authors extend SAFE defining a formal framework to combine multiple existing string abstract domains for the analysis of JavaScript programs, showing that combinations of simple abstract domains out-perform the precision of the existing state-of-art static analyzers, comparing their approach with SAFE, TAJS and JSAI.
Future ideas. In this paper we have proposed string static program analysis for a set of relevant string manipulation operations, whose semantics is inspired by the JavaScript behaviors. We are currently working on extending our framework in order to fully cover the JavaScript `String` built-in global object, formally defining the remaining methods contained in it. Afterwards, the first goal is to integrate our abstract semantics into a static analyzer for JavaScript, that uses

finite state automata to approximate strings. In order to decrease the number of false positives in our string approximation in presence of loops, several techniques will be involved, such as loop unrolling and LSA [37]. The domain described in this paper has been equipped only with a widening, to enforce termination in fix-point computations, that may lead to a big loss of precision. A narrowing will be studied and introduced in our static analyzer in order to retrieve some precision lost when widening is applied.
We conclude by observing that we are strongly confident that an important future application of our semantics may be the string-to-code primitives analysis. Consider, for instance, in JavaScript programs, the `eval` function, transforming strings into code. As already observed, our semantics is sound and precise enough for answering to some non-trivial property of interest. Hence, we think this semantics for strings can be a good starting point for a sound and *precise enough* analysis of `eval`, for example in JavaScript, which is still an open problem in static analysis.

References

[1] P. Abdulla, M. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer & J. Stenman (2014): *String Constraints for Verification.* In: *CAV'14.*

[2] R. Alur & P. Madhusudan (2004): *Visibly pushdown languages.* In: *STOC'04.*

[3] R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey & C. Zhang (2017): *Combining String Abstract Domains for JavaScript Analysis: An Evaluation.* In: *TACAS'17.*

[4] V. Arceri & S. Maffeis (2017): *Abstract Domains for Type Juggling. ENTCS* 331.

[5] G. Balakrishnan & T. Reps (2006): *Recency-Abstraction for Heap-Allocated Storage.* In: *SAS'06.*

[6] C. Bartzis & T. Bultan (2004): *Widening Arithmetic Automata.* In: *CAV'04.*

[7] H. Bordihn, M. Holzer & M. Kutrib (2009): *Determination of finite automata accepting subregular languages. Theor. Comput. Sci.* 410(35).

[8] A. Bouajjani, P. Habermehl, L. Holík, T. Touili & T. Vojnar (2008): *Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata.* In: *CIAA'08.*

[9] A. Bouajjani, P. Habermehl & T. Vojnar (2004): *Abstract Regular Model Checking.* In: *CAV'04.*

[10] C. Câmpeanu, A. Paun & S. Yu (2002): *An Efficient Algorithm for Constructing Minimal Cover Automata for Finite Languages. Int. J. Found. Comput. Sci.* 13(1).

[11] T. Choi, O. Lee, H. Kim & K. Doh (2006): *A Practical String Analyzer by the Widening Approach.* In: *APLAS'06.*

[12] Agostino Cortesi & Martina Olliaro (2018): *M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs.* In: *TASE'18*, pp. 1–8.

[13] G. Costantini, P. Ferrara & A. Cortesi (2015): *A suite of abstract domains for static analysis of string values. Softw., Pract. Exper.* 45(2).

[14] P. Cousot (1997): *Types as Abstract Interpretations.* In: *POPL'97.*

[15] P. Cousot & R. Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In: *POPL'77.*

[16] P. Cousot & R. Cousot (1992): *Abstract Interpretation Frameworks. J. Log. Comput.* 2(4).

[17] P. Cousot & R. Cousot (1992): *Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation.* In: *PLILP'92.*

[18] P. Cousot, R. Cousot & F. Logozzo (2011): *A parametric segmentation functor for fully automatic and scalable array content analysis.* In: *POPL'11*, pp. 105–118.

[19] P. Cousot & N. Halbwachs (1978): *Automatic Discovery of Linear Restraints Among Variables of a Program.* In: *POPL'78.*

[20] M. D. Davis, R. Sigal & E. J. Weyuker (1994): *Computability, Complexity, and Languages: Fund. of Theor. CS.* Academic Press Professional, Inc.

[21] M. Domaratzki, J. Shallit & S. Yu (2001): *Minimal Covers of Formal Languages.* In: *DLT'01.*

[22] V. D'Silva (2006): *Widening for Automata.* Diploma Thesis, Institut Fur Informatick, UZH.

[23] Aymeric Fromherz, Abdelraouf Ouadjaout & Antoine Miné (2018): *Static Value Analysis of Python Programs by Abstract Interpretation.* In: *NFM'18.*

[24] R. Giacobazzi & I. Mastroeni (2016): *Making abstract models complete. MSCS* 26(4).

[25] R. Giacobazzi & E. Quintarelli (2001): *Incompleteness, counterexamples and refinements in abstract model-checking.* In: *SAS'01.*

[26] R. Giacobazzi, F. Ranzato & F. Scozzari. (2000): *Making Abstract Interpretation Complete. JACM* 47(2).

[27] D. Hauzar & J. Kofron (2015): *Framework for Static Analysis of PHP Applications.* In: *ECOOP'15.*

[28] L. Holík, P. Janku, A. Lin, P. Rümmer & T. Vojnar (2018): *String constraints with concatenation and transducers solved efficiently.*

[29] J. Hopcroft & J. Ullman (1979): *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

[30] S. Jensen, A. Møller & P. Thiemann (2009): *Type Analysis for JavaScript.* In: *SAS'09.*

[31] S. H. Jensen, P. A. Jonsson & A. Møller (2012): *Remedying the eval that men do.* In: *ISSTA'12.*

[32] V. Kashyap, K. Dewey, E. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann & B. Hardekopf (2014): *JSAI: a static analysis platform for JavaScript.* In: *FSE'14.*

[33] H. Lee, S. Won, J. Jin, J. Cho & S. Ryu (2012): *SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript.* In: *FOOL'12.*

[34] A. Widjaja Lin & P. Barceló (2016): *String solving with word equations and transducers: towards a logic for analysing mutation XSS.* In: *POPL'16.*

[35] J. Midtgaard, F. Nielson & H. R. Nielson (2016): *A Parametric Abstract Domain for Lattice-Valued Regular Expressions.* In: *SAS'16.*

[36] C. Park, H. Im & S. Ryu (2016): *Precise and scalable static analysis of jQuery using a regular expression domain.* In: *DLS'16.*

[37] C. Park & S. Ryu (2015): *Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity.* In: *ECOOP'15.*

[38] M. Pradel & K. Sen (2015): *The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript.* In: *ECOOP'15.*

[39] E. Pribavkina & E. Rodaro (2010): *State Complexity of Prefix, Suffix, Bifix and Infix Operators on Regular Languages.* In: *DLT'10.*

[40] M Sharir & A Pnueli (1978): *Two approaches to interprocedural data flow analysis.* NYU CS, NY.

[41] W3S: *JS String Ref.* `www.w3schools.com/jsref/jsref_obj_string.asp`. Accessed 16-06-2018.

[42] W. Xu, F. Zhang & S. Zhu (2012): *The power of obfuscation techniques in malicious JavaScript code: A measurement study.* In: *MALWARE'12.*

[43] F. Yu, T. Bultan, M. Cova & O. H. Ibarra (2008): *Symbolic String Verification: An Automata-Based Approach.* In: *SPIN'08.*

[44] S. Yu, Q. Zhuang & K. Salomaa (1994): *The State Complexities of Some Basic Operations on Regular Languages. Theor. Comput. Sci.* 125(2).