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 faced an important increment of usage 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 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. For this reason, string obfuscation (e.g., string splitting) is becoming one of the most common obfuscation techniques in JavaScript malware, 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. All 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, may fail to precisely analyze strings in dynamic contexts. For instance, in the example, existing static
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);
ad = tt. substring (tt. indexOf ("O") , 3);
ad += 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.

...
The programming language we denote by $\text{FA}$ is a tuple $\langle Q, \Sigma, \delta, F \rangle$, where $Q$ is a finite set of states, $\Sigma$ 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 $\delta : Q \times \Sigma \rightarrow Q$ is a function then $\delta$ is called deterministic FA (DFA).

Regular languages and finite state automata. We follow \cite{28} for automata notation. A finite state automaton (FA) is a tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$ where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $\Sigma$ 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. The class of languages recognized by FAs is the class of regular languages. We denote the set of all DFAs as $\text{DFA}$. Given an automaton $\mathcal{A}$, we denote the language accepted by $\mathcal{A}$ as $\mathcal{L}(\mathcal{A})$. A language $L$ is regular iff there exists a FA $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$. From the Myhill-Nerode theorem \cite{19}, 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 $L$, we denote by $\text{Min}(L)$ the minimum DFA $\mathcal{A}$ s.t. $L = \mathcal{L}(\mathcal{A})$.

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 \cite{41}. Other JavaScript string operations can be modeled by composition of the given string operations or as particular cases of them. Primitive values are $\mathcal{V} = S \cup \mathbb{Z} \cup \mathbb{B} \cup \{\text{NaN}\}$ with $S \overset{\text{def}}{=} \Sigma^*$ (strings on the alphabet $\Sigma$), $\mathbb{B} \overset{\text{def}}{=} \{\text{true}, \text{false}\}$ and $\text{NaN}$ a special value denoting not-a-number.

Implicit type conversion. In order to properly capture the semantics of the language IMP, inspired by the JavaScript semantics, we need to deal with implicit type conversion \cite{2}. 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 identity when applied to values not needing conversion, e.g., $\text{toInt}$ on integers. Then, $\text{toString} : \mathcal{V} \rightarrow S$ maps any input value to its string representation; $\text{toInt} : \mathcal{V} \rightarrow \mathbb{Z} \cup \{\text{NaN}\}$ returns the integer corresponding to a value, when it is possible: For true and false it returns respectively 1 and 0, for strings in $\Sigma$ it returns the corresponding integer, while all the other values are converted to NaN. For instance, $\text{toInt}("42") = 42$, $\text{toInt}("42hello") = \text{NaN}$. Finally, $\text{toBool} : \mathcal{V} \rightarrow \mathbb{B}$ returns false when the input is 0, and true for all the other non boolean primitive values.

Semantics. Program states are partial maps from identifiers to primitive values, i.e., $\text{States} : \text{Id} \rightarrow \mathcal{V}$. The concrete big-step semantics \cite{1} : $\text{Stmt} \times \text{States} \rightarrow \text{States}$ is quite standard, and it includes dynamic typing and implicit type conversion. Also the expression semantics, $\llbracket \cdot \rrbracket : \text{Exp} \times \text{States} \rightarrow \mathcal{V}$, is standard; we only provide the formal and precise semantics of the IMP string operations. Let $\sigma, \sigma' \in S$ and $i, j \in \mathbb{Z}$ (values which are not strings or numbers respectively,

\footnote{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.}

$\sigma_1 \cdots \sigma_{j-1}$, and we denote it by $\text{substring} (\sigma, i, j)$. 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. $\mathcal{I} : \Sigma_{\mathbb{Z}} \rightarrow \mathbb{Z}$ maps numeric strings to the corresponding integers. Dually, we define the function $\mathcal{S} : \mathbb{Z} \rightarrow \Sigma_{\mathbb{Z}}$ that maps each integer to its minimal numeric string representation (e.g., 1 is mapped to the string "1", and not "+1").
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 \( SS : S \times Z \times Z \to S \) defined as:

\[
SS(\sigma, i, j) = \begin{cases} 
SS(\sigma, j, i) & j < i \\
\text{substring}(\sigma, i, \max(j, |\sigma|)) & \text{otherwise}
\end{cases}
\]

**charAt**: It returns the character at a specified index. The semantics is the function \( CA : S \times Z \to S \) defined as follows:

\[
CA(\sigma, i) = \begin{cases} 
\sigma_i & 0 \leq i < |\sigma| \\
\varepsilon & \text{otherwise}
\end{cases}
\]

**indexOf**: It returns the position of the first occurrence of a given substring. The semantics is the function \( IO : S \times S \to Z \) defined as follows:

\[
IO(\sigma, \sigma') = \begin{cases} 
\min\{i \mid \sigma_i \ldots \sigma_j = \sigma'\} & \exists i, j. \sigma_i \ldots \sigma_j = \sigma' \\
-1 & \text{otherwise}
\end{cases}
\]

**length**: It returns the length of a string \( \sigma \in S \). Its semantics is the function \( LE : S \to Z \) defined as \( LE(\sigma) = |\sigma| \). The concrete semantics of these functions is given in Sect. 2.2 (i.e., \( CC(\sigma, \sigma') = \sigma \cdot \sigma' \)).

### 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 \( \wp(\Sigma^*) \). In particular, our aim is that of underlying the well known theoretical foundations of regular languages (and therefore of DFA) characterizing automata as a domain for abstracting the computation program semantics in the abstract interpretation framework. The exploited idea is that of approximating strings as regular languages represented by the minimum DFAs \([19]\) recognizing them. In general, we have more DFAs than regular languages, hence the domain of automata is indeed the quotient \( DFA_{/\equiv} \) w.r.t. the equivalence relation induced by language equality: \( \forall A_1, A_2 \in DFA_{/\equiv} A_1 \equiv A_2 \Leftrightarrow \mathcal{L}(A_1) = \mathcal{L}(A_2) \). Hence, any equivalence class is composed by the automata that recognize the same regular language. We abuse notation by representing equivalence classes in the domain \( DFA_{/\equiv} \) w.r.t. \( \equiv \) by one of its automata (usually the minimum), i.e., when we write \( A \in DFA_{/\equiv} \) we mean \( [A]_{/\equiv} \). The partial order \( \sqsubseteq_{\text{dra}} \) induced by language inclusion is \( \forall A_1, A_2 \in DFA_{/\equiv} A_1 \sqsubseteq_{\text{dra}} A_2 \Leftrightarrow \mathcal{L}(A_1) \subseteq \mathcal{L}(A_2) \), which is well defined since automata in the same \( \equiv \)-equivalence class recognize the same language.

The least upper bound (lub) \( \sqcup_{\text{dra}} : DFA_{/\equiv} \times DFA_{/\equiv} \to DFA_{/\equiv} \) on the domain \( DFA_{/\equiv} \), corresponds to the standard union between automata: \( \forall A_1, A_2 \in DFA_{/\equiv} A_1 \sqcup_{\text{dra}} A_2 = \text{Min}(\mathcal{L}(A_1) \cup \mathcal{L}(A_2)) \). It
Figure 4: (a) $A_1$ (b) $A_2$ (c) $\text{Min}(A_1 \cup A_2)$

Figure 5: (a) $A_1$ s.t. $\mathcal{L}(A_1) = \{\epsilon, a\}$ (b) $A_2$ s.t. $\mathcal{L}(A_2) = \{a, aa\}$ (c) $A_1 \triangledown_1 A_2$

is the minimum automaton recognizing the union of the languages $\mathcal{L}(A_1)$ and $\mathcal{L}(A_2)$. This is a well-defined notion since regular languages are closed under union. As example, consider Fig. 4, where the automaton in Fig. 4c is the lub of $A_1$ and $A_2$ given in Fig. 4a and Fig. 4b, respectively. 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: $\forall A_1, A_2 \in \text{Dfa}_/\equiv, A_1 \sqcap_{\text{Dfa}} A_2 \overset{\text{def}}{=} \text{Min}(\mathcal{L}(A_1) \cap \mathcal{L}(A_2))$.

**Theorem 1.** $\langle \text{Dfa}_/\equiv, \subseteq_{\text{Dfa}}, \sqcup_{\text{Dfa}}, \sqcap_{\text{Dfa}}, \text{Min}(\emptyset), \text{Min}(\Sigma^*) \rangle$ 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 still ensuring soundness \[15\]. A well known example is the convex polyhedra domain \[18\].

**Widening.** The domain $\text{Dfa}_/\equiv$ is an infinite domain, and it is not ACC, i.e., does not contain finite ascending chains. For instance, consider the set of languages $\{\{a^j b^i \mid 0 \leq j \leq i \}\}_{i \geq 0} \subseteq \wp(\Sigma^*)$ 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 \[15\]. Most of the proposed abstract domains for strings \[12,29,31,32\] trivially satisfy ACC being finite, but they may lose precision during the abstract computation \[16\]. 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 \[16\]. As far as automata are concerned, existing widenings are defined in terms of a state equivalence relation merging states recognizing the same language, up to a fixed length $n$ (set as parameter for tuning the widening precision) \[6,21\]. We denote this parametric widening with $\triangledown_n, n \in \mathbb{N}$ \[21\].

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

```java
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. 5a we report the abstract value of the variable $\text{str}$ at the beginning of the second iteration of the loop, while in Fig. 5b the abstract value of the variable $\text{str}$ at the end of the second iteration. Before starting a new iteration, in the example, we apply $\triangledown_1$ between two automata, namely we merge together all the states having the same outgoing character. The minimization of the obtained automaton is reported in Fig. 5c. The next iteration will reach the fix-point, guaranteeing soundness and termination.

\(^2\)Note that, some works have studied automatic procedures to compute, given an input language $L$, the regular cover of $L$ \[20\] (i.e., an automaton containing the language $L$). In particular, \[10,20\] have studied regular covers guaranteeing that the automaton obtained is the best w.r.t. a minimal relation (but not minimum).
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 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 \( V = \mathbb{Z} \cup \mathbb{B} \cup S \cup \{\text{NaN}\} \). 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 \( V \). Consider, for each primitive value \( D \), an abstract domain \( D^\# \) (we denote the domain \( D^\# \) without bottom as \( D^\# \), equipped with an abstraction \( \alpha_D : D \to D^\# \) and a concretization \( \gamma_D : D^\# \to D \) forming a Galois insertion \([14]\).

Coalesced sum. One way to merge domains is the coalesced sum \([13]\). 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 \( \mathbb{Z}^\#, S^\# \) and \( \mathbb{B}^\# \), the coalesced sum is the abstraction of \( \wp(\{\text{NaN}\}) \) depicted in Fig. 6. 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 \( \text{IMP} \) fragment: \( \text{if} (y < 5) x = "42"; \text{else} x = \text{true}; \). 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 \( \alpha_S("42") \uplus \alpha_B(\text{true}) = \top \).

Cartesian product. In order to catch union types, without losing too much precision, we need to complete \([23,25]\) 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 \([22]\). Hence, the complete abstract domain w.r.t. dynamic typing and implicit type conversion is: \( \mathbb{Z}^\# \times \mathbb{B}^\# \times S^\# \times \wp(\{\text{NaN}\}) \), abstraction of \( \wp(V) \). In this combining abstract domain, the value of \( x \) after the if-execution is precisely \( (\bot, \alpha_B(\text{true}), \alpha_S("42"), \bot) \), now an element of the domain, inferring that the value of \( x \) can be \( \alpha_S(\text{true}) \) or \( \alpha_S("42") \), but surely not an abstract integer or NaN.

In the following, we consider the abstract domain \( V^\# \) for string analysis obtained as cartesian product of the following abstractions: \( \mathbb{Z}^\#, \mathbb{B}^\#, S^\#, \wp(\{\text{true, false}\}) \).

4 The \( \text{IMP} \) abstract semantics

In this section, we define the abstract semantics of the language \( \text{IMP} \) on the abstract domain \( V^\# \). In particular, we have to define the expressions abstract semantics \([1]^{\#} : \text{Exp} \times \text{States} \to V^\# \), 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 [19]). Let $L \in \wp(\Sigma^*)$ be a regular language. The suffixes of $L$ are $SU(L) \defeq \{ y \in \Sigma^* \mid \exists x \in \Sigma^*, xy \in L \}$, and the prefixes of $L$ are $PR(L) \defeq \{ x \in \Sigma^* \mid \exists y \in \Sigma^*, xy \in L \}$.

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

**Definition 2** (Right quotient [19]). Let $L_1, L_2 \in \Sigma^*$ be regular languages. The right quotient of $L_1$ w.r.t. $L_2$ is $RQ(L_1, L_2) \defeq \{ x \in \Sigma^* \mid \exists y \in L_2, xy \in L_1 \}$.

For example, let $L_1 = \{xab, yab\}$ and $L_2 = \{b, ab\}$. The right quotient of $L_1$ w.r.t. $L_2$ is $RQ(L_1, L_2) = \{xa, ya, x, y\}$.

**Definition 3** (Substrings/Factors [7]). Let $L \in \wp(\Sigma^*)$ be a regular language. The set of its substrings/factors is $FA(L) \defeq \{ y \in \Sigma^* \mid \exists x, z \in \Sigma^*, xyz \in L \}$.

These operations are all defined as transformations of regular languages. In [19] the corresponding algorithms on FA are provided. In particular, let $A, A_1 \in DFA/\equiv$ and $i \in \mathbb{N}$, then $SU(A), PR(A), SU(A,i), FA(A)$ and $RQ(A,A_1)$ are the algorithms corresponding to the transformations $SU(L(A)), PR(L(A)), SU(L(A,i)), FA(L(A))$ and $RQ(L(A), L(A_1))$, respectively. Namely, $\forall A, A_1 \in DFA/\equiv$, $i \in \mathbb{N}$, the following facts holds:

$$SU(L(A)) = L(SU(A)), PR(L(A)) = L(PR(A)), FA(L(A)) = L(FA(A))$$

$$RQ(L(A), L(A_1)) = L(RQ(A, A_1)), SU(L(A), i) = L(SU(A, i))$$

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 $SS^3 : \overline{DFA/\equiv} \times \mathbb{N} \times \mathbb{N} \to \overline{DFA/\equiv}$, 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 $SS^3$ 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 $A \in \overline{DFA/\equiv}$, $i, l \in \mathbb{Z} \cup \{-\infty\}$, $j, k \in \mathbb{Z} \cup \{+\infty\}$ (for the sake of readability we denote by $\sqcup$ the automata lub $\sqcup_{DFA}$ and by $\cap$ the glb $\cap_{DFA}$), 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., $SS(L(A), [i,j], [l,k])$.

For example, let $L = \{a\}^* \cup \{hello, bc\}$, the set of its substrings from 1 to 3 is $SS(L, [1, 1], [3, 3]) = \{c, a, aa, el, c\}$. The automaton accepting this language is computed by the operator

$$SS(A, [i,j], [l,k]) \defeq \bigcup_{a \in [i,j], b \in [l,k]} (RQ(SU(A, a), SU(A, b)) \cap \text{Min}(\Sigma^{b-a})) \sqcup (SU(A, a) \sqcup \text{Min}(\Sigma^{<b-a}))$$

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


Theorem 3. \( SS^i \) is sound and complete: \( \forall A \in \text{DFA} /_=} , I, J \in \text{Int}, \text{SS}^i(A, I, J) = \mathcal{L}(\text{SS}^i(A, I, J)) \).

### 4.2 Abstract semantics of \texttt{charAt}

The abstract semantics of \texttt{charAt} should return the automaton accepting the language of all the characters of strings accepted by an automaton \( A \), in a position inside a given interval \([i, j]\): This
The abstract semantics of \texttt{length} should return the interval of all the possible string lengths in an automaton, i.e., it is \( \text{LE}^2 : \text{DFA}_/\equiv \times \text{Int} \rightarrow \text{Int} \) computed by Alg. \ref{algo:le}, where \( \text{minPath}, \text{maxPath} : \text{DFA}_/\equiv \times Q \times Q \rightarrow \wp(Q) \) return the minimum and the maximum paths between two states of the input automaton, respectively. \( \text{len} : \wp(Q) \rightarrow \mathbb{N} \) returns the size of a path, and \( \text{hasCycle} : \text{DFA}_/\equiv \rightarrow \{\text{true, false}\} \) 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. \ref{fig:8a} 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. \ref{fig:8b} the length interval is \([3, +\infty]\).

**Theorem 5.** \( \text{LE}^2 \) is sound but not complete: \( \forall A \in \text{DFA}_/\equiv, I \in \text{Int}, \text{CA}(\mathcal{L}(A), I) = \mathcal{L}(\text{CA}^2(A, I)) \).

### 4.4 Abstract semantics of \texttt{index0f}

The abstract semantics of \texttt{index0f} is \( \text{IO}^2 : \text{DFA}_/\equiv \times \text{DFA}_/\equiv \rightarrow \text{Int} \) and should return the interval of possible positions of strings in a language inside strings of another language. Consider for instance, the automaton \( A \) in Fig. \ref{fig:9a} and suppose to call \( \text{IO}^2(A, A') \) where \( A' = \text{Min}(\{bc\}) \). The idea is that of building, for each state \( q \) in \( A \), the automaton \( A_q \) which is \( A \) where all the states are

\[\text{CA}^2(A_q) = \begin{cases} \cup_{i \in [l, h]} \text{SS}(A, [i, i], [i + 1, i + 1]) & l, h \in \mathbb{Z} \\ \text{CA}^2(A)[0, h] \sqcup \text{Min}(\{\epsilon\}) & l = -\infty, h \in \mathbb{Z}, h \geq 0 \\ \text{Min}(\{\epsilon\}) & l = -\infty, h \in \mathbb{Z}, l < 0 \\ \text{Min}(\text{chars}(\text{SU}(A, l))) \sqcup \text{Min}(\{\epsilon\}) & l \in \mathbb{Z}, l \geq 0, h = +\infty \\ \text{Min}(\text{chars}(A)) \sqcup \text{Min}(\{\epsilon\}) & l = -\infty \text{ or } l \in \mathbb{Z}, l < 0, h = +\infty \end{cases}\]
Theorem 7. \(\exists \mathcal{A}, \mathcal{A}' \in \mathsf{DFA}_{/\equiv}. \mathsf{CC}(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}')) = \mathsf{CC}^2(\mathcal{A}, \mathcal{A}').\)

4.5 Abstract semantics of concatenation

The abstract semantics of string concatenation is \(\mathsf{CC}^2 : \mathsf{DFA}_{/\equiv} \times \mathsf{DFA}_{/\equiv} \rightarrow \mathsf{DFA}_{/\equiv}\) and returns the concatenation between two automata. Since regular languages are closed under the concatenation operation, even finite state automata do. In Fig. 10, we report an example of concatenation between two automata. Hence, \(\mathsf{CC}^2\) exactly implements the standard concatenation operation between automata. Given the closure property on automata, the following result holds.

Theorem 6. \(\mathsf{IO}^2\) is sound but not complete: \(\forall \mathcal{A}, \mathcal{A}' \in \mathsf{DFA}_{/\equiv}. \mathsf{IO}(\mathcal{L}(\mathcal{A}), \mathcal{L}(\mathcal{A}')) \subset \mathsf{IO}^2(\mathcal{A}, \mathcal{A}').\)

As a counterexample to completeness, consider the automaton \(\mathcal{A}'\) in Fig. 9b. \(\mathsf{IO}^2(\mathcal{A}', \mathsf{Min}(\{b\})) = [-1, 3] \not\subseteq \mathsf{IO}(\mathcal{L}(\mathcal{A}''), \{b\}) = \{0, 3\}.\) 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.

Algorithm 1: \(\mathsf{LE}^2 : \mathsf{DFA}_{/\equiv} \rightarrow \) Int alg.

Input: \(\mathcal{A} = (Q, \Sigma, \delta, q_0, F)\)

Output: \(\mathsf{LE}^2(\mathcal{A})\)

1. \(P\_\text{len} \leftarrow 0; P\_\text{len} \leftarrow \infty\)
2. if hasCycle(\(\mathcal{A}\)) then
3. \hspace{1em} foreach \(q_f \in F\) do
4. \hspace{2em} \(p \leftarrow \min\text{Path}(\mathcal{A}, q_0, q_f)\);  
5. \hspace{2em} if \(\text{len}(p) < P\_\text{len}\) then \(P\_\text{len} \leftarrow \text{len}(p)\); 
6. \hspace{2em} return \([P\_\text{len}, +\infty]\);
9. else
10. \hspace{1em} foreach \(q_f \in F\) do
11. \hspace{2em} \(P \leftarrow \max\text{Path}(\mathcal{A}, q_0, q_f)\);
12. \hspace{2em} if \(\text{len}(P) < P\_\text{len}\) then \(P\_\text{len} \leftarrow \text{len}(P)\); 
13. \hspace{2em} if \(\text{len}(P) > P\_\text{len}\) then \(P\_\text{len} \leftarrow \text{len}(P)\);
14. \hspace{1em} return \([P\_\text{len}, P\_\text{len}]\);
15. end
16. end

Algorithm 2: \(\mathsf{IO}^2 : \mathsf{DFA}_{/\equiv} \times \mathsf{DFA}_{/\equiv} \rightarrow \) Int alg.

Input: \(\mathcal{A} = (Q, \Sigma, \delta, q_0, F), \mathcal{A}' = (Q', \Sigma, \delta', q_0', F')\)

Output: \(\mathsf{IO}^2(\mathcal{A}, \mathcal{A}')\)

1. indexesOf \(\leftarrow \emptyset\)
2. foreach \(q \in Q\) do
3. \hspace{1em} \(a_q \leftarrow (Q, \Sigma, \delta, q, Q)\);
4. \hspace{2em} if \(a_q \cap \mathcal{A}' \neq \emptyset\) then
5. \hspace{3em} indexesOf \(\leftarrow\) indexesOf \(\cup\) \(\{\text{len}(\min\text{Path}(\mathcal{A}, q_0, q))\}\);
6. \hspace{3em} if \(\exists p = \text{path}(q_0, q)\) s.t. hasCycle(\(p\)) then
7. \hspace{4em} indexesOf \(\leftarrow\) indexesOf \(\cup\) \([+\infty]\);
8. \hspace{1em} end
9. else
10. \hspace{1em} indexesOf \(\leftarrow\) indexesOf \(\cup\) \([-1]\);
11. end
12. end
13. if \(|\mathcal{L}(\mathcal{A})| = |\mathcal{L}(\mathcal{A}')| = 1\) then
14. \hspace{1em} return \([\min\{\text{indexesOf}\}, \min\{\text{indexesOf}\}]\);
15. else
16. \hspace{1em} return \([\min\{\text{indexesOf}\}, \max\{\text{indexesOf}\}]\);
17. end

Figure 10: (a) \(\mathcal{A}, \mathcal{L}(\mathcal{A}) = \{a^n \mid n \in \mathbb{N}\} \cup \{b\}\) (b) \(\mathcal{A}', \mathcal{L}(\mathcal{A}') = \{cd^n \mid n \in \mathbb{N}\}\) (c) \(\mathcal{A}'' = \mathsf{CC}^2(\mathcal{A}, \mathcal{A}')\)

final and the initial state is \(q\). Hence, we check whether \(a_q \cap \mathcal{A}'\) is non empty and we collect the size of the minimum path from \(q_0\) to \(q\) in \(\mathcal{A}\). If there exists at least one state from which any string accepted by \(\mathcal{A}'\) cannot be read, we collect -1. In the example, \(a_{q_0}\) adds \(\{0\}\), \(a_{q_1}\) adds \(\{1\}\), while all the other states add \(\{-1\}\). Finally, we return the interval \([\min\{-1, 1, 0\}, \max\{-1, 1, 0\}] = [-1, 1]\). The full algorithm is reported in Alg. 2.
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, being the conversions concerning booleans, not-a-number and intervals straightforward. Let $\text{toBool}^\sharp : \mathbb{V}^\sharp \to \mathbb{B}^\sharp$ be applied to $A \in \text{DFA}_{/=}$: if $A \cap \text{Min}({\varepsilon}) = \emptyset$, it returns $\{\text{true}\}$, when $A = \text{Min}({\varepsilon})$ the function returns $\{\text{false}\}$, otherwise the function returns $\{\text{true, false}\}$. Implicit type conversion to $\text{DFA}_{/=}$ is handled by the function $\text{toStr}^\sharp : \mathbb{V}^\sharp \to \text{DFA}_{/=}$. As far as numeric strings are concerned, $\text{toStr}^\sharp$ returns $\text{Min}({\text{NaN}})$. If the input is the boolean value $\text{true}$ $\{\text{false}\}$ it returns $\text{Min}({\text{true}})$ $\text{Min}({\text{false}})$, otherwise it returns $\text{Min}({\text{true}}) \sqcup \text{Min}({\text{false}})$. Converting intervals to FA is more tricky. If $l, h \in \mathbb{Z}$, the conversion to automata is simply $\{\bigcup_{i \in [l, h]} \text{Min}({\text{S}(i)})\}$. The interval-to-automaton conversion for $[0, +\infty]$ and $[-\infty, 0]$ are respectively shown in Fig. 11a and Fig. 11b. Other unbounded intervals, $[+l, +\infty]$ and $[-l, +\infty]$ ($l > 0$), are converted to $\text{toStr}^\sharp([0, +\infty]) \setminus \text{toStr}^\sharp([0, l])$ and $\text{toStr}^\sharp([-l, 0]) \sqcup \text{toStr}^\sharp([0, +\infty])$, respectively. Conversions of intervals $[-\infty, l]$ and $[l, -\infty]$ ($l > 0$) are analogous, while, $\text{toStr}^\sharp([-\infty, +\infty]) = \text{Min}(\Sigma_Z)$. Finally, $\text{tolInt}^\sharp : \mathbb{V}^\sharp \to \text{Int} \sqcup \{\text{NaN}\}$ handles conversion to intervals. Given an automaton $A$, if $A \cap \text{Min}(\Sigma_Z) = \emptyset$, the automaton is precisely converted to $\text{NaN}$, otherwise, if $A \subseteq_{\text{DFA}} \text{Min}(\Sigma_Z)$ it means that $\Sigma^\sharp(A)$ contains only numeric strings. For the sake of precision, we check whether $A$ 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\footnote{Available at \url{http://www.github.com/SPY-Lab/fsa} and the IMP static analyzer at \url{http://www.github.com/SPY-Lab/mu-js}}. 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 $\sqcup_{\text{DFA}}$, $\cap_{\text{DFA}}$, and a parametric widening for tuning precision and forcing convergence. The library is suitable and easily pluggable into existing static analyzers, such as $\{29, 31, 32, 37\}$. The bottleneck of our library is the determinization operation, having exponential complexity $\{28\}$ (we rely on determinization in the minimization algorithm, in order to keep the automata arising during the abstract computations minimum and deterministic). It worth noting that, as reported in Thm. 1, $\varphi(\Sigma^\sharp)$ (string concrete domain) and $\text{DFA}_{/=}$ (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 conditions that, in our string abstract domain, is expressible by $\text{Min}(\varphi(\Sigma^\sharp))$, 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
the `eval` call, is the automaton \( A_d \) in Fig. 12. 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 \( A_d \) contain a string corresponding to an assignment to an ActiveXObject?* We can simply answer by checking the predicate \( A_d \cap \text{Min}(\{\text{new ActiveXObject}\} \cdot \Sigma^* \cdot \{\}) \neq \emptyset \), checking whether \( A_d \) 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 \( A_d \) contain `eval` string?* We can answer by checking whether \( A_d \cap \text{Min}(\{\text{eval}\}) \neq \emptyset \), that is false and enforce 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. 12 due to the widening application. Nevertheless, it worth noting that such result is obtained without any precision improvement on fix-point computations, such as loop unrolling or loop-sensitiveness analyses. We think these analyses will drastically decrease false positives of the proposed string analysis but we will address this topic for future work.

5 Discussion and related work

In this paper, we have proposed an abstract semantics for a toy imperative language \( \text{IMP} \), 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 formal 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.

**Analysis vs verification.** Even if several solutions, also involving finite state machines, have been proposed for string solving and verification [2, 33, 43], it worth noting that our approach is placed instead in the context of string static analysis. Among the years, there has always been the intuition that program analysis was *harder* than verification: given a program, the aim of the former is to derive invariants for each program point, the one of the latter is instead to check whether a certain property holds for the given input program. Recently, this concept has been formalized from a computability point of view [17], confirming this belief. Hence, our approach, placed in the context of static analysis of string manipulation programs, has goals that are hardly comparable with the solutions proposed in the context of verification, such those cited above.

**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 verificator 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 only 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 to 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 one but pursues a different task and do 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 lattice-based 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 [3,8,27]. 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.

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 [29,31,32,37]. TAJS [29] 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 [30]. 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 [32], 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.

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 aim is to involve out 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 involved 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

A Appendix: Algorithms

Alg. 3 computes the right quotient between two automata, $A_1$ and $A_2$. For each state $q$ of $A_1$, we build a new automaton $A_i$, equals to $A_1$, except that the only initial state is $q$ (line 3). If $A_i$ recognizes strings of $A_2$, i.e., $A_i \cap A_2 \neq \emptyset$, the algorithm collects $q$ in $F_{RQ}$ (lines 4-5). Finally, the result is an automaton equals to $A_1$, except that the set of final states is $F_{LQ}$. We abuse notation of Min by denoting the minimization operation on automata.

Algorithm 3: $RQ : \text{Dfa}_/\equiv \times \text{Dfa}_/\equiv \rightarrow \text{Dfa}_/\equiv$ algorithm

Input: $A_1,A_2 \in \text{Dfa}_/\equiv$ s.t. $A_1 = (Q_1,q_0^1,\Sigma,\delta_1,F_1), A_2 = (Q_2,q_0^2,\Sigma,\delta_2,F_2)$
Output: $RQ(A_1,A_2)$
1 $F_{RQ} \leftarrow \emptyset$
2 foreach $q \in Q_1$ do
3 $A_i \leftarrow (Q_1,q,\Sigma,\delta_1,F_1)$
4 if $A_2 \cap A_i \neq \emptyset$ then
5 $F_{RQ} \leftarrow F_{RQ} \cup \{q\}$
6 end
7 end
8 return Min($((Q_1,q_0^1,\Sigma,\delta_1,F_{RQ}))$);

Alg. 4 computes the suffix automata of $A$. For each state $q$, the algorithm checks if there exists a path from $q$ to a final states (line 3). If it is the case (line 4), $q$ is collected in $I_{SU}$. Finally, the result is the (minimum) automaton equals to $A$, except that the set of the initial states is $I_{SU}$. Dually, Alg. 5 computes the prefix automata of $A$.

Algorithm 4: Algorithm of $SU : \text{Dfa}_/\equiv \rightarrow \text{Dfa}_/\equiv$

Input: $A \in \text{Dfa}_/\equiv$ s.t. $A = (Q,q_0,\Sigma,\delta,F)$
Output: $SU(A)$
1 $I_{SU} \leftarrow \emptyset$
2 foreach $q \in Q$ do
3 if $\exists p, p = \text{path}(q_0,q)$ then
4 $I_{SU} \leftarrow I_{SU} \cup \{q\}$
5 end
6 end
7 return Min($((Q,I_{SU},\Sigma,\delta,F))$);

Algorithm 5: Algorithm of $PR : \text{Dfa}_/\equiv \rightarrow \text{Dfa}_/\equiv$

Input: $A \in \text{Dfa}_/\equiv$ s.t. $A = (Q,q_0,\Sigma,\delta,F)$
Output: $PR(A)$
1 $F_{PR} \leftarrow \emptyset$
2 for $q \in Q$ do
3 if $\exists p, p = \text{path}(q_0,q)$ then
4 $F_{PR} \leftarrow F_{PR} \cup \{q\}$
5 end
6 end
7 $A_{PR} = \text{Min}(((Q,q_0,\Sigma,\delta,F_{PR}))$;

B Appendix: Selected proofs

Proof. (of Theorem 4) Consider the family of languages on $\Sigma = \{a,b\}$

$$L_i \overset{\text{def}}{=} \{ a^n b^n \mid n \leq i \} \cup \{ a^n b^m \mid n, m > i \}$$
These languages are trivially regular since we require the same number of \(a\) and \(b\) only up to a fixed bound, the parameter \(i\), for all the strings with a number of \(a\) and \(b\) greater than \(i\) we do not fix any relation between the lengths.

We can prove that the intersection of all these languages is a context free, not regular, language. Namely we have that \(\bigcap_{i \leq n} L_i = \{ a^n b^n | n \in \mathbb{N} \} \). In particular, consider \(a^k b^k\), then \(\forall i, k > i\) we have that \(a^k b^k \in \{ a^n b^m | n, m > i \} \subseteq L_i\), while \(\forall i, k \leq i\) we have \(a^k b^k \in \{ a^n b^n | n \leq i \} \subseteq L_i\), hence \(a^k b^k \in \bigcap_i L_i\). Consider now \(a^j b^k \in \bigcap_i L_i\), then \(j = k\) since otherwise (suppose without losing generality that \(j \leq k\)) \(\forall i, j \leq i\) we have \(a^j b^k \notin L_i\). Therefore \(a^j b^k = a^k b^k \in \{ a^n b^n | n \in \mathbb{N} \} \). Hence, we have the equality of the intersection with a well-known not regular language.

\[ \text{Proposition 1. For all } L \text{ in } \varphi(\Sigma^\ast), \text{ for all } i, j \in \mathbb{Z}, \text{ we have } \text{SS}(L, i, j) = \text{Nps}(L, i, j) \cup \text{Ps}(L, i, j), \]\n
where \(\text{Nps}(L, i, j) = \text{Su}(L, i) \cap \Sigma^{<j-i} \text{ and } \text{Ps}(L, i, j) = \text{Rq}(\text{Su}(L, i), \text{Su}(L, j)) \cap \Sigma^j \).\(^6\)

\[ \text{Proof. For the sake of simplicity, we assume that } i \text{ and } j \text{ are positive and } j \geq i, \text{ since our rewriting handles these corner cases. We separately prove that the new definition exactly computes these}\]

\(^6\)In order to be coherent with the IMP semantics of substring, we have that if \(i\) and \(j\) are negative then they are treated as zero, and if \(j < i\) the values are swapped (the substring is always computed from the smaller to the greater value).
two classes of partition.

\[
\begin{align*}
NPS(L,i,j) &= \{ \text{substring}(\sigma,i,n) \mid j > n = |\sigma|, \sigma \in L \} \\
&= \{ y \mid \exists x \in \Sigma^*, xy \in L, j > |xy|, |x| = i \} \\
&= \{ y \mid \exists x \in \Sigma^*, xy \in L, j > |x| + |y|, |x| = i \} \\
&= \{ y \mid \exists x \in \Sigma^*, xy \in L, j > |y|, |y| = i \} \\
&= \{ y \mid \exists x \in \Sigma^*, xy \in L, |x| = i \} \cap \{ y \mid j > i + |y| \} \\
&= Su(L,i) \cap \{ y \mid |y| > i \} \\
&= Su(L,i) \cap \Sigma^{j-i}
\end{align*}
\]

\[
Ps(L,i,j) = \{ \text{substring}(\sigma,i,j) \mid j \leq n = |\sigma|, \sigma \in L \} \\
\]

Lemma 1. Let \( L \in \wp(\Sigma^*) \) be a regular language, \( i,j \in \mathbb{Z} \). Then

\[ SS^{-}(L,i,j) = Rq(Su(L,i),Su(Su(L,j))) \]

Proof.

\[
SS^{-}(L,i,j) = \{ SS(\sigma,i,k) \mid \sigma \in L, k \geq j \} \\
\]

\[ = \{ y \mid \exists x, z \in \Sigma^*, xz \in L, |x| = i, |x| = j, k \geq j, xyz \in L \} \\
\]

\[ = \{ y \mid \exists z \in Su(L,k), k \geq j, yz \in Su(L,i) \} \\
\]

\[ = \{ y \mid \exists z \in Su(Su(L,j)), yz \in Su(L,i) \} \]

\[ \text{(*)} \]

\[ = Rq(Su(L,i),Su(Su(L,j))) \]

where (*) holds since we can prove that \( \bigcup_{k \geq j} Su(L,k) = Su(Su(L,j)) \). By definition \( Su(Su(L,j)) = \{ z \in \Sigma^* \mid \exists x, y \in \Sigma^*, |x| = j, xyz \in L \} \), while \( Su(L,k) = \{ z \in \Sigma^* \mid \exists w \in \Sigma^*, |w| = k, wz \in L \} \). Hence, if \( z \in Su(Su(L,j)) \) then we have that \( \exists k \geq j, |xy| = k \), namely \( \exists k \geq j, z \in Su(L,k) \). On the other hand, if \( z \) is in the union above then \( \exists k \geq j, z \in Su(L,k) \), but there exists \( x, y \in \Sigma^* \) such that \( w = xy \) \( (|w| = k) \) with \( |x| = j \) and \( |y| = k - j \), but then by definition \( yz \in Su(L,j) \), and therefore \( z \in Su(Su(L,j)) \).

Lemma 2. Let \( L \) be a regular language, \( i,j \in \mathbb{Z} \). The following fact holds

\[ SS^{+}(L,i) = FA(Su(L,i)) \]
Proof.

\[ \text{SS}^{\rightarrow}(L, i) = \{ \text{SS}(\sigma, l, k) \mid \sigma \in L, l, k \geq i \} \]
\[ = \{ y \mid \exists x, z \in \Sigma^*, |x| = l, |xy| = k, l, k \geq i, xyz \in L \} \]
\[ = \{ y \mid \exists x, z \in \Sigma^*. yz \in \text{SU}(L, l), |xy| = k, l, k \geq i, xyz \in L \} \]
\[ = \{ y \mid \exists x, z \in \Sigma^*. yz \in \text{SU}(\text{SU}(L, i)), |xy| = k, l, k \geq i, xyz \in L \} \]
\[ = \{ y \mid \exists x, z \in \Sigma^*. yz \in \text{SU}(\text{SU}(L, i)), z \in \text{SU}(\text{SU}(L, i)), xyz \in L \} \]
\[ = \{ y \mid \exists x, z \in \Sigma^*. yz \in \text{SU}(\text{SU}(L, i)), xyz \in L \} \]
\[ = \text{PR}(\text{SU}(\text{SU}(L, i))) \]
\[ = \text{FA}(\text{SU}(L, i)) \]

\[ \square \]

Theorem 8. Let \( A \in \text{DFA}_{\equiv}, i, j \in \mathbb{Z} \). The following facts holds

\[ \mathcal{L}(\text{SS}(A, i, j)) = \text{SS}(\mathcal{L}(A), i, j) \]
\[ \mathcal{L}(\text{SS}^{\rightarrow}(A, i, j)) = \text{SS}^{\rightarrow}(\mathcal{L}(A), i, j) \]
\[ \mathcal{L}(\text{SS}^{\leftarrow}(A, i, j)) = \text{SS}^{\leftarrow}(\mathcal{L}(A), i, j) \]
\[ \mathcal{L}(\text{SS}^{\leftrightarrow}(A, i, j)) = \text{SS}^{\leftrightarrow}(\mathcal{L}(A), i, j) \]

Proof. By definition and by Proposition 1, Lemma 1 and Lemma 2. \( \square \)

Of Thm. 4 For space limitations, we report the proof for the case \( l \in \mathbb{Z}, l \geq 0, h = +\infty \). The other cases are straightforward.

\[ \text{CA}(\mathcal{L}(A), [l, +\infty)) \]
\[ = \{ \text{CA}(\sigma, i) \mid \sigma \in \mathcal{L}(A), i \in [l, +\infty) \} \]
\[ = \{ y \mid \exists x, z \in \Sigma^*. |x| = i, i \in [l, +\infty], |y| \leq 1, xyz \in \mathcal{L}(A) \} \]
\[ = \{ y \mid \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(A), i), i \in [l, +\infty], |y| \leq 1, xyz \in \mathcal{L}(A) \} \]
\[ = \{ y \mid \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(A), i), i \in [l, +\infty], xyz \in \mathcal{L}(A) \} \cap \{ y \mid |y| \leq 1 \} \]
\[ = \{ y \mid \exists z \in \Sigma^*. yz \in \text{SU}(\text{SU}(\mathcal{L}(A), i)) \} \cap \Sigma^{\leq 1} \]
\[ = \text{PR}(\text{SU}(\text{SU}(\mathcal{L}(A), i))) \cap \Sigma^{\leq 1} \]
\[ = \mathcal{L}(\text{PR}(\text{SU}(\text{SU}(\mathcal{L}(A), i))) \cap_{\text{DFA}} \text{Min}(\Sigma^{\leq 1})) \]
\[ = \mathcal{L}(\text{FA}(\text{SU}(\mathcal{L}(A), i))) \cap_{\text{DFA}} \text{Min}(\Sigma^{\leq 1}) \]
\[ = \mathcal{L}(\text{CA}^{\rightarrow}(A, [l, +\infty])) \]

\[ \square \]

Of Thm. 5 \( \text{LE}^3 \) is not complete, i.e. \( (\text{LE}^3(A) \not\subset \text{LE}(\mathcal{L}(A)) \). As a counterexample, consider the automaton \( A_2 \) in Fig. 5B.

\[ \text{LE}^3(A) = [3, +\infty] = \{ n + 3 \mid n \in \mathbb{N} \} \not\subset \text{LE}(\mathcal{L}(A)) = \{3, 5\} \cup \{3n + 1 \mid n > 0\} \]

As far as soundness is concern, we argue \( \forall \sigma \in \mathcal{L}(A), |\sigma| \in \gamma(\text{LE}^3(A)) \). Let consider the following cases:
A has cycle: if $\sigma$ is the minimum string accepted by $A$, its length is computed by searching for the minimum path from the initial to final states (lines 4-9 and lines 15-17), hence, the resulting interval takes into account $|\sigma|$ (line 10 and line 22). Strings of length greater than the minimum, it is contained in the resulting interval, since it is positive unbounded.

A has not cycle: if $\sigma$ is the minimum string accepted by $A$, its length is contained in the resulting interval as explained in the previous case. If $\sigma$ is the maximum string accepted by $L(A)$, Alg. 1 searches for the maximum length path from the initial to final states (lines 18-20) and the returning interval contains $|\sigma|$ (line 22); Otherwise, if $\sigma$ is not the minimum or the maximum string accepted by $L(A)$, it trivially belongs to the resulting interval, since the interval goes from minimum string length and maximum string length.