applied sciences

MDPI

Article

Static Analysis for ECMAScript String Manipulation Programs

Vincenzo Arceri [1,2,*], Isabella Mastroeni [1] and Sunyi Xu [1]

[1] Department of Computer Science, University of Verona, 37134 Verona, Italy; isabella.mastroeni@univr.it (I.M.); sunyi1908@gmail.com (S.X.)

[2] Ca' Foscari University, Department of Environmental Sciences, Informatics and Statistics, 30170 Venice, Italy

* Correspondence: vincenzo.arceri@univr.it

Received: 25 April 2020; Accepted: 16 May 2020; Published: 20 May 2020

check for updates

Abstract: 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 great challenge for static analysis of these 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. In this scenario, more precise string analyses become a necessity. The goal of this paper is to place a first step for precisely handling dynamic language string features. In particular, we propose a new abstract domain approximating strings as finite state automata and an abstract interpretation-based static analysis for the most common string manipulating operations provided by the ECMAScript specification. The proposed analysis comes with a prototype static analyzer implementation for an imperative string manipulating language, allowing us to show and evaluate the improved precision of the proposed analysis.

Keywords: string static analysis; abstract interpretation; abstract semantics

1. Introduction

Dynamic languages, for instance JavaScript or Python, have seen an important growth in a very wide range of fields and applications. Common features in these languages are dynamic typing (typing occurs during program execution, at run-time) and implicit type conversion [1], which lighten the development phase and allow programs not to block execution in the 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 the writing of programs, allowing, at the same time, statically unpredictable executions which might make them harder to understand [1]. For this reason, string obfuscation (e.g., string splitting) is becoming one of the most common obfuscation techniques in JavaScript malwares [2], making it hard to statically analyze code. Consider, for example, the JavaScript program fragment in Figure 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, as is the number of iterations of the loops.

```
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.

All these observations suggest that, in order to statically understand statements which are dynamically generated and executed, it may be extremely useful to statically analyze the string value of `d`. Unfortunately existing static analyzers for dynamic languages [3–6], might fail to precisely analyze strings in dynamic contexts. For instance, in the example above, TAJS [3], JSAI [4] and SAFE [5], lose precision on the `eval` input value and any information gathered so far about it. Namely, the issue of analyzing dynamic languages, even if tackled by sophisticated tools as the cited ones, still lacks formal approaches for handling the dynamic features of string manipulation, such as dynamic typing, implicit type conversion and dynamic code generation. Instead, in [7], a new approach for dynamic language analysis is proposed based on finite state automata for abstracting strings, coming with both a precise string abstraction able to infer string properties in general and a sound abstract interpreter for dynamically-generated code.

Contributions

In this paper (This is an extended and revised version of [8] integrated with a more complete range of string operations, detailed proofs of the results presented (proofs are reported in Appendix A) and an improved implementation that will be discussed in Section 6.), we focus on the characterization of an abstract interpretation-based [9] formal framework, capable of handling dynamic typing and implicit type conversion, by defining an abstract semantics able to (precisely, when possible) capture the previously mentioned dynamic features. Even if we do not tackle the problem of analyzing dynamically generated code (meaning that we do not analyze its behavior), as highlighted in [7], such semantics is a necessary step towards a sufficiently precise analysis for it, since it is 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 possible 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 might execute, we surely need to (over-)approximate the set of precise string values of its input. Hence we propose an approach defining a collecting semantics for strings. With this task in mind, we will 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 will define on it an abstract semantics for a μJS language, augmented with implicit type conversion, dynamic typing and several interesting string operations taken from the official ECMAScript language specification [10], namely the JavaScript language specification, whose concrete semantics is inspired by the JavaScript one. In particular, for each one of these operations we will provide the algorithm computing its abstract semantics and we will discuss their soundness and completeness.

Paper structure

In Section 2 we recall relevant notions on finite state automata and the core language adopted for this paper is established in Section 3. In Section 4.1 we define the finite state automata domain, highlighting some important operations and theoretical results. In Section 4 we discuss and present two ways of combining abstract domains (for primitive types) suitable for dynamic languages. Then, In Section 5, we present the new abstract semantics for string manipulating operations. In Section 6 we examine and evaluate the precision of the string static analyzer based on the above semantics. Finally, in Section 7, we discuss and compare this paper to the most related works and we draw our conclusions.

2. Background

In this section, we recall some basic notations and notions that will be used in the rest of the paper.

2.1. String Notation

We denote by Σ a finite non-empty 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 $\sigma, \sigma' \in \Sigma^*$, $\sigma \cdot \sigma'$ 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 string $\sigma_i \cdots \sigma_{j-1}$. 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"`, and not `"+1"`). Given $\sigma \in \Sigma^*$ and $n \in \mathbb{N}$, we denote with σ^n the n-times concatenation of σ. Given a symbol $c \in \Sigma$ we denote with $\mathsf{toLowerCase}(c)$ its corresponding lower-case symbol, if it is a capital letter, otherwise c is returned. We abuse notation denoting by $\mathsf{toLowerCase}(\sigma)$ the string σ where at each position any upper-case symbol is replaced with the corresponding lower-case symbol.

2.2. Regular Languages and Finite State Automata

We follow [11] for automata notation. A finite state automaton (FA) is a tuple $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite non-empty 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). We consider DFA also those FA 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. The class of languages recognized by FA is the class of regular languages. We denote the set of all DFA as $\textsc{Dfa}$. 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 [12], for each regular language 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. $\mathtt{L} = \mathscr{L}(\mathtt{A})$. Given an automaton $\mathtt{A}$, we denote by $\mathsf{Kleene}(\mathtt{A})$ the automaton that recognizes the language corresponding to the Kleene-closure of $\mathscr{L}(\mathtt{A})$, namely the automaton $\mathtt{A}'$ s.t. $\mathscr{L}(\mathtt{A}') = \mathscr{L}(\mathsf{Kleene}(\mathtt{A}))) = \{\ \sigma^n \mid \sigma \in \mathscr{L}(\mathtt{A}), n \in \mathbb{N}\ \}$. Moreover, given an automaton $\mathtt{A}$, we rely on the predicate $\mathsf{hasCycle}(\mathtt{A})$ that checks whether $\mathtt{A}$ is cyclic.

2.3. Abstract Interpretation

Abstract interpretation establishes a correspondence between a concrete semantics and an approximated one called abstract semantics [9,13]. In a Galois Connection framework, if C and A are complete lattices, a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ forms a Galois Connection

(GC for short) between C and A if for every $x \in C$ and $y \in A$ we have $\alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. α and γ are called abstraction function and concretization function, respectively.

Let L be a complete lattice. $X \subseteq L$ is a Moore family of L if $X = \mathcal{M}(X) \triangleq \{ \bigwedge S \mid S \subseteq X \}$ and $\top$ (top element) $\in \mathcal{M}(X)$. If any concrete object in C has a best abstraction in the abstract domain A implies that A is a Moore family of C and so there exists a Galois connection between C and A.

Weaker forms of correspondence are possible, e.g., when A is not a complete lattice or when only γ exists [14]. In all cases, relative precision in A is given by comparing the meaning of abstract objects in C, i.e., $x_1 \leq_A x_2$ if $\gamma(x_1) \leq_C \gamma(x_2)$. If $f : C \rightarrow C$ is a continuous function and A is an abstraction of C by means of the GC $\langle \alpha, \gamma \rangle$, then f always has a best correct approximation in A, $f^A : A \rightarrow A$, defined as $f^A \triangleq \alpha \circ f \circ \gamma$. Any approximation $f^\sharp : A \rightarrow A$ of f in A is sound if $f^A \sqsubseteq f^\sharp$.

In abstract interpretation, there exist two notions of completeness: *backward completeness* and forward completeness. The former is the best known form of completeness and focuses on complete abstractions of the inputs, while the latter is forward completeness [15–17] and it focuses on complete abstractions of the outputs, both w.r.t. an operation of interest. When we do not have a GC, namely when only the concretization γ exists, we need to focus only on forward completeness, as we will do in this paper. Given a GC $\langle \alpha, \gamma \rangle$, a concrete function $f : C \rightarrow C$ and an abstract function $f^\sharp : A \rightarrow A$, $f^\sharp$ is forward complete w.r.t. f if $\forall a \in A.\, f(\gamma(a)) = \gamma(f^\sharp(a))$.

A satisfies the ascending chain condition (ACC) if all ascending chains are finite. When A is not ACC convergence to the limit of the fix-point iterations can be ensured through widening operators. A widening operator $\nabla : A \times A \rightarrow A$ approximates the least upper bounds, i.e., $\forall x, y \in A \,.\, x, y \leq_A (x \nabla y)$ and it is such that for any increasing chain $x_1 \leq x_2 \leq \cdots \leq x_n \leq \ldots$ the increasing chain $w_0 = \bot$ and $w_{i+1} = w_i \nabla xi$ is finite.

3. The Core Language

In this paper, we consider a JavaScript core language, reported in Figure 2, that we call μJS, containing several representative string operations taken from the set of methods offered by the JavaScript built-in class `String`, detailed in the ECMAScript language specification [10]. Even though we have decided to focus on a core of the operations, note that the missing methods (e.g., `indexOf` or `endsWith`) can be easily modeled as composition of our chosen string methods or as particular cases of them. Nevertheless, as we will discuss in Section 6, these operations have been implemented and tested.

$$
\begin{array}{lll}
\mathbf{Exp} & ::= & v \in \mathbb{V} \;\mid\; \mathbf{Id} \in \textsc{Id} \;\mid\; \mathbf{Exp}\ \texttt{+}\ \mathbf{Exp} \;\mid\; \mathbf{Exp}\ \texttt{-}\ \mathbf{Exp} \;\mid\; \mathbf{Exp}\ \texttt{*}\ \mathbf{Exp} \;\mid\; \mathbf{Exp}\ \texttt{/}\ \mathbf{Exp} \\
 & \mid & \mathbf{Exp}\ \texttt{\&\&}\ \mathbf{Exp} \;\mid\; \mathbf{Exp}\ \texttt{||}\ \mathbf{Exp} \;\mid\; \texttt{!}\ \mathbf{Exp} \;\mid\; \texttt{length(}\mathbf{Exp}\texttt{)} \\
 & \mid & \texttt{startsWith(}\mathbf{Exp}\texttt{,}\mathbf{Exp}\texttt{)} \;\mid\; \texttt{substr(}\mathbf{Exp}\texttt{,}\mathbf{Exp}\texttt{,}\mathbf{Exp}\texttt{)} \;\mid\; \texttt{charAt(}\mathbf{Exp}\texttt{,}\mathbf{Exp}\texttt{)} \\
 & \mid & \texttt{concat(}\mathbf{Exp}\texttt{,}\mathbf{Exp}\texttt{)} \;\mid\; \texttt{includes(}\mathbf{Exp}\texttt{,}\mathbf{Exp}\texttt{, }\mathbf{Exp}\texttt{)} \\
 & \mid & \texttt{trim(}\mathbf{Exp}\texttt{)} \;\mid\; \texttt{repeat(}\mathbf{Exp}\texttt{,}\mathbf{Exp}\texttt{)} \\
\mathbf{Block} & \texttt{::=} & \texttt{\{ \}} \;\mid\; \texttt{\{}\ \mathbf{Stmt}\ \texttt{\}} \\
\mathbf{Stmt} & \texttt{::=} & \mathbf{Id}\ \texttt{=}\ \mathbf{Exp}\texttt{;} \;\mid\; \texttt{if (}\mathbf{Exp}\texttt{)}\ \mathbf{Block}\ \texttt{else}\ \mathbf{Block} \\
 & \mid & \texttt{while (}\mathbf{Exp}\texttt{)}\ \mathbf{Block} \;\mid\; \mathbf{Block} \;\mid\; \mathbf{Stmt}\ \ \mathbf{Stmt} \;\mid\; \texttt{;}
\end{array}
$$

Figure 2. μJS syntax.

μJS Semantics

In μJS the primitive values are $\mathbb{V} = \mathbb{S} \cup \mathbb{Z} \cup \mathbb{B} \cup \{\texttt{NaN}\}$ with $\mathbb{S} \overset{\text{def}}{=} \Sigma^*$ (strings on the alphabet Σ), $\mathbb{B} \overset{\text{def}}{=} \{\texttt{true}, \texttt{false}\}$ and `NaN` a special value denoting not-a-number.

Program states are partial maps from identifiers to primitive values, i.e., $\textsc{States} : \textsc{Id} \rightarrow \mathbb{V}$. The concrete big-step semantics $[\![\cdot]\!] : \textsc{Stmt} \times \textsc{States} \rightarrow \textsc{States}$ is standard and follows [18], and it includes dynamic typing and implicit type conversion. In addition, the expression semantics, $(\!|\cdot|\!) :$

$\textsc{Exp} \times \textsc{States} \to \mathbb{V}$, is standard and follows [18]; we only provide the formal and precise semantics of the μJS 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, moreover, negative values are treated as zero).

`substring`: It extracts the substring between two indexes from a string. The semantics is defined by the function $\textsc{Ss}$: $\mathbb{S} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{S}$ as:

$$\textsc{Ss}(\sigma, i, j) \stackrel{\text{def}}{=} \begin{cases} \textsc{Ss}(\sigma, j, i) & j < i \\ \sigma_i \dots \sigma_j & j < |\sigma| \wedge i \leq j \\ \sigma_i \dots \sigma_n & j \geq n = |\sigma| \wedge i \leq j \end{cases}$$

`charAt`: It returns the character, i.e., the string of unitary length, at a specified index in a string σ. The semantics is the function $\textsc{Ca}$: $\mathbb{S} \times \mathbb{Z} \to \mathbb{S}$ defined as follows:

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

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

`concat`: It returns the concatenation between two strings and its concrete semantics $\textsc{Cc} : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ relies on the concatenation operator reported in Section 2.

$$\textsc{Cc}(\sigma, \sigma') = \sigma \cdot \sigma'$$

`startsWith`: It determines whether a specified string σ starts with σ'. The semantics is the function $\textsc{Sw} : \mathbb{S} \times \mathbb{S} \to \mathbb{B}$ defined as:

$$\textsc{Sw}(\sigma, \sigma') \stackrel{\text{def}}{=} \begin{cases} \texttt{true} & \exists \sigma'' \in \Sigma^*. \sigma = \sigma' \cdot \sigma'' \\ \texttt{false} & \text{otherwise} \end{cases}$$

`repeat`: It returns the given string repeated n times. The semantics is the function $\textsc{Rt} : \mathbb{S} \times \mathbb{Z} \to \mathbb{S}$ defined as $\textsc{Rt}(\sigma, n) \stackrel{\text{def}}{=} \sigma^n$.

`includes`: It determines whether a string σ' is a substring of σ. The semantics is the function $\textsc{In}$: $\mathbb{S} \times \mathbb{S} \to \mathbb{B}$ defined as:

$$\textsc{In}(\sigma, \sigma') \stackrel{\text{def}}{=} \begin{cases} \texttt{true} & \exists \phi, \psi \in \Sigma^*. \sigma = \phi \cdot \sigma' \cdot \psi \\ \texttt{false} & \text{otherwise} \end{cases}$$

`toLowerCase`: It returns the given string in all lowercase letters. The semantics is the function $\textsc{Lc}$: $\mathbb{S} \to \mathbb{S}$ defined as $\textsc{Lc}(\sigma) \stackrel{\text{def}}{=} \text{toLowerCase}(\sigma)$.

`trimLeft`: It removes all the white-spaces at the beginning of a string. The semantics is the function $\textsc{Tl} : \mathbb{S} \to \mathbb{S}$ defined as:

$$\textsc{Tl}(\sigma) \stackrel{\text{def}}{=} \sigma' \quad \text{where } \psi = \max\{\ \psi' \in (\text{␣})^* \mid \sigma = \psi' \cdot \sigma'\ \} \wedge \sigma = \psi \cdot \sigma'$$

`trimRight`: It removes all the white-spaces at the end of a string. The semantics is the function $\textsc{Tr}$: $\mathbb{S} \to \mathbb{S}$ defined as:

$$\textsc{Tr}(\sigma) \stackrel{\text{def}}{=} \sigma' \quad \text{where } \psi = \max\{\ \psi' \in (\text{␣})^* \mid \sigma = \sigma' \cdot \psi'\ \} \wedge \sigma = \sigma' \cdot \psi$$

`trim`: It removes all the white-spaces at the end and beginning of a string. The semantics is the function $\textsc{Tm} : \mathbb{S} \to \mathbb{S}$ defined as: $\textsc{Tm}(\sigma) \stackrel{\text{def}}{=} \textsc{Tr}(\textsc{Tl}(\sigma))$.

Implicit Type Conversion

In order to properly capture the semantics of the language μJS, inspired by the JavaScript semantics, we need to deal with implicit type conversion [18]. For each primitive value, we define an auxiliary function converting it to other primitive values (Figure 3). Note that all the functions behave like identity when applied to values not needing conversion, e.g., `toInt` on integers. Then, $\texttt{toString} : \mathbb{V} \rightarrow \mathbb{S}$ maps any input value to its string representation; $\texttt{toInt} : \mathbb{V} \rightarrow \mathbb{Z} \cup \{\texttt{NaN}\}$ 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, $\texttt{toInt}(\text{“}42\text{”}) = 42$, $\texttt{toInt}(\text{“}42\mathit{hello}\text{”}) = \texttt{NaN}$. Finally, $\texttt{toBool} : \mathbb{V} \rightarrow \mathbb{B}$ returns `false` when the input is 0, and `true` for all the other non boolean primitive values. It is worth noting that the auxiliary functions defined in Figure 3 do not correspond to explicit casting but they model the implicit type conversion implemented by JavaScript. In particular, these functions cannot be directly called by a programmer since they are exclusively used internally (indeed implicitly) by the semantics when a type value of an expression operand is required.

$$\texttt{toString}(v) = \begin{cases} v & v \in \mathbb{S} \\ \text{“}\texttt{NaN}\text{”} & v = \texttt{NaN} \\ \text{“}\texttt{true}\text{”} & v = \texttt{true} \\ \text{“}\texttt{false}\text{”} & v = \texttt{false} \\ \mathcal{S}(v) & v \in \mathbb{Z} \end{cases} \qquad \texttt{toInt}(v) = \begin{cases} v & v \in \mathbb{Z} \\ 1 & v = \texttt{true} \\ 0 & v = \texttt{false} \,\vee\, v = \texttt{NaN} \\ \mathcal{I}(v) & v \in \mathbb{S} \wedge v \in \Sigma_{\mathbb{Z}} \\ \texttt{NaN} & v \in \mathbb{S} \wedge v \notin \Sigma_{\mathbb{Z}} \end{cases}$$

$$\texttt{toBool}(v) = \begin{cases} v & v \in \mathbb{B} \\ \texttt{true} & v \in \mathbb{Z} \smallsetminus \{0\} \vee v \in \mathbb{S} \smallsetminus \{\epsilon\} \\ \texttt{false} & v = 0 \vee v = \epsilon \vee v = \texttt{NaN} \end{cases}$$

Figure 3. μJS implicit type conversion functions.

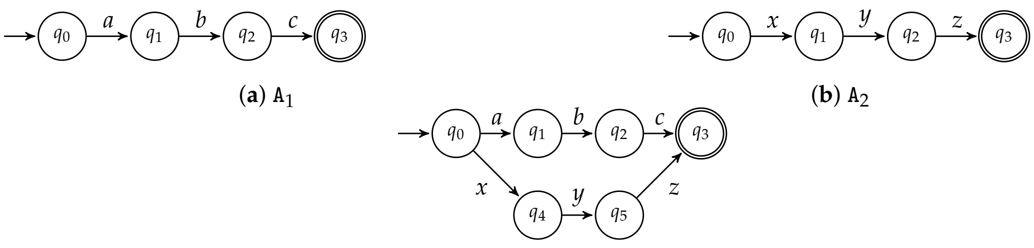
4. An Abstract Domain for String Manipulation

4.1. The Finite State Automata Abstract Domain for Strings

In this section, we describe the finite state automata abstract domain for strings [19–21], namely the domain of regular languages over $\wp(\Sigma^*)$. In particular our goal is to exploit automata, and therefore regular languages, for approximating string values collected during analysis. The idea is to approximate strings as regular languages represented by the minimum DFA [12] recognizing them. In general, we have more DFA than regular languages, hence the domain of automata is indeed the quotient $\text{DFA}_{/\equiv}$ w.r.t. the equivalence relation induced by language equality: $\forall \texttt{A}_1, \texttt{A}_2 \in \text{DFA}_{/\equiv}.\ \texttt{A}_1 \equiv \texttt{A}_2 \Leftrightarrow \mathscr{L}(\texttt{A}_1) = \mathscr{L}(\texttt{A}_2)$. Therefore any equivalence class is composed by automata that recognize the same regular language. We abuse notation by representing these classes in the domain $\text{DFA}_{/\equiv}$ w.r.t. $\equiv$ using one of its automata (usually the minimum), i.e., when we write $\texttt{A} \in \text{DFA}_{/\equiv}$ we mean $[\texttt{A}]_{\equiv}$.

The partial order $\sqsubseteq_{\text{DFA}}$ is induced by language inclusion, i.e., $\forall \texttt{A}_1, \texttt{A}_2 \in \text{DFA}_{/\equiv}\ .\ \texttt{A}_1 \sqsubseteq_{\text{DFA}} \texttt{A}_2 \Leftrightarrow \mathscr{L}(\texttt{A}_1) \subseteq \mathscr{L}(\texttt{A}_2)$, which is well defined since automata in the same $\equiv$-equivalence class recognize the same language.

The corresponding least upper bound, $\sqcup_{\text{DFA}} : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \rightarrow \text{DFA}_{/\equiv}$ on the domain $\text{DFA}_{/\equiv}$, is the standard union between automata: $\forall \texttt{A}_1, \texttt{A}_2 \in \text{DFA}_{/\equiv}.\ \texttt{A}_1 \sqcup_{\text{DFA}} \texttt{A}_2 \overset{\text{def}}{=} \mathsf{Min}(\mathscr{L}(\texttt{A}_1) \cup \mathscr{L}(\texttt{A}_2))$. It is the minimum automaton recognizing the union of the languages $\mathscr{L}(\texttt{A}_1)$ and $\mathscr{L}(\texttt{A}_2)$. This is a well-defined notion since regular languages are closed under union. As example consider Figure 4, where the automaton in Figure 4c is the least upper bound of $\texttt{A}_1$ and $\texttt{A}_2$ given in Figure 4a,b, respectively.



(**a**) $\mathtt{A}_1$

(**b**) $\mathtt{A}_2$

(**c**) $\mathsf{Min}(\mathtt{A}_1 \cup \mathtt{A}_2)$

Figure 4. Least upper bound of $\text{DFA}_{/\equiv}$.

The (finite) 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, it cannot exists a Galois connection between $\text{DFA}_{/\equiv}$ and $\wp(\Sigma^*)$, i.e., there may be no minimal automaton abstracting a language. Note that some works [22–24] have studied automatic procedures to compute, given an input language L, the regular cover of L [23] (i.e., an automaton containing the language L). Some of them [22,23] studied regular covers guaranteeing that the automaton obtained is the best w.r.t. a minimal relation (but not minimum). However this is not a concern since the relation between concrete semantics and abstract semantics can be weakened still ensuring soundness [14]. A well known example is the convex polyhedra domain [25].

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 $L_i = \{\, a^j b^j \mid 0 \leq j \leq i \,\} \subseteq \wp(\Sigma^*)$, indexed by a constant natural $i \in \mathbb{N}$, forming an infinite ascending chain of finite regular languages. The set of the corresponding minimal automata trivially forms an ascending chain on $\text{DFA}_{/\equiv}$. This clearly implies that any computation on $\text{DFA}_{/\equiv}$ may lose convergence [14] (Most of the proposed abstract domains for strings [3–5,26] trivially satisfy ACC being finite, but they may lose precision during the abstract computation [27].).

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) [28,29]. We denote this parametric widening with $\nabla_n : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \to \text{DFA}_{/\equiv}$, with $n \in \mathbb{N}$ [28] and it is defined in the following.

Let $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$ and $\mathtt{A}' = (Q', \Sigma, \delta', q_0', F')$ be two finite state automata such that $\mathscr{L}(\mathtt{A}) \subseteq \mathscr{L}(\mathtt{A}')$: the widening between $\mathtt{A}$ and $\mathtt{A}'$ is formalized in terms of a relation $R \subseteq Q \times Q'$ between the sets of states of the two automata. The relation R is used to define an equivalence relation $\equiv_R \subseteq Q' \times Q'$ over the states of $\mathtt{A}'$, such that $\equiv_R = R \circ R^{-1}$. The widening between $\mathtt{A}$ and $\mathtt{A}'$ is then given by the quotient automaton of $\mathtt{A}'$ w.r.t. the partition induced by $\equiv_R$: $\mathtt{A}' \nabla_R \mathtt{A}' = \mathtt{A}'_{\equiv_R}$ (Given $\mathtt{A} \in \text{DFA}_{/\equiv}$ and a partition π over its states, we denote as $\mathtt{A}_\pi = (Q', \delta', q_0', F', \Sigma)$ the quotient automaton [12].). Thus, the widening operator merges the states of $\mathtt{A}'$ that are equivalent by the relation $\equiv_R$. By changing the relation R, we obtain different widening operators [28]. It has been proved that convergence is guaranteed when the relation $R_n \subseteq Q \times Q'$ is such that $(q, q') \in R_n$ iff q and q' recognize the same language of strings of length at most n [28]. The parameter n therefore tunes the length of strings determining the equivalence of states used for merging them in the widening. It is worth noting that the smaller is n, the more information will be lost by widening.

In the following, given $\mathtt{A}, \mathtt{A}' \in \textsc{Dfa}_{/\equiv}$ (without any constraints on the languages they recognize), we define the widening operator on $\textsc{Dfa}_{/\equiv}$ parametric on $n \in \mathbb{N}$ as follows.

$$\mathtt{A} \nabla_n \mathtt{A}' \overset{\text{def}}{=} \mathtt{A} \nabla_{R_n} (\mathtt{A} \sqcup_{\textsc{Dfa}} \mathtt{A}')$$

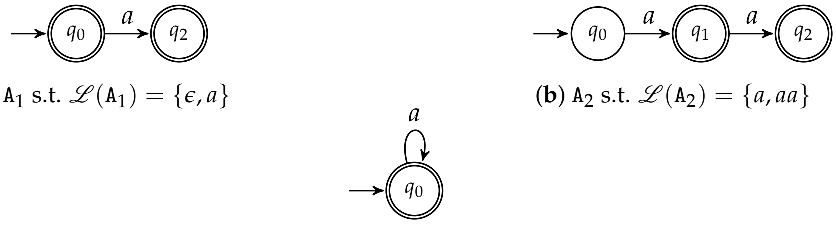
In order to show how the defined widening operator works, let us discuss the following example.

Example 1. *Consider the following μJS fragment*

```
str=""; while (x < 100) { str=str+"a"; x=x+1; }
```

The value of the variable `x` *is unknown and so is the number of iterations of the* `while`*-loop. In these cases, in order to guarantee soundness and termination, we apply the widening operator.*

In Figure 5a we report the abstract value of the variable `str` *at the beginning of the second iteration of the loop, while in Figure 5b the abstract value of the variable* `str` *at the end of the second iteration. Before starting a new iteration, in the example, we apply ∇_1 between the two automata, specifically we merge all the states having the same outgoing character. The minimization of the so obtained automaton is reported in Figure 5c. The next iteration will reach the fix-point, guaranteeing termination.*



(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) $\mathtt{A}_1 \nabla_1 \mathtt{A}_2$

Figure 5. Widening of $\textsc{Dfa}_{/\equiv}$.

4.2. An Abstract Domain for μJS

By definition, string operations in our language also involve other primitive values, such as booleans or integers, hence we need an abstract domain able to observe any possible concrete value. This is additionally necessary for dealing with implicit type conversion as we will later observe.

We therefore have to design an abstract domain for string manipulation dealing with other primitive types, namely being able to combine different abstractions of various types. In particular, 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 them in a unique abstract domain for $\mathbb{V}$. Consider, for each value $\mathbb{D}$, an abstract domain $\mathbb{D}^\sharp$ (we denote $\mathbb{D}^\sharp_{\not\perp}$ the domain $\mathbb{D}^\sharp$ without bottom), 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 [9].

4.2.1. Coalesced Sum

One way to merge domains is the coalesced sum [30]. The resulting domain contains all the non-bottom elements of the input domains, with a new top and a new bottom.

Definition 1 (Coalesced sum domain [31])**.** *Let $\langle A, \leq_A, \sqcup_A, \sqcap_A, \perp_A, \top_A \rangle$ and $\langle B, \leq_B, \sqcup_B, \sqcap_B, \perp_B, \top_B \rangle$ be two lattices abstracting the posets $\langle C, \leq_C \rangle$ and $\langle D, \leq_D \rangle$ with abstraction functions $\alpha_A : A \to C$ and $\alpha_B : B \to D$, respectively. The coalesced sum domain $A \oplus B$ is defined as:*

$$A \oplus B \overset{def}{=} \{\perp_{A \oplus B}\} \cup \{\, a \mid a \in A_{\not\perp} \,\} \cup \{\, b \mid b \in B_{\not\perp} \,\} \cup \{\top_{A \oplus B}\}$$

such that the partial order is defined as $x \leq_{A \oplus B} y \Leftrightarrow x \leq_A y \; (x, y \in A) \vee x \leq_B y \; (x, y \in B)$ *and* $\forall x \in A \oplus B.\ \bot_{A \oplus B} \leq_{A \oplus B} x \leq_{A \oplus B} \top_{A \oplus B}$*, its least upper bound is defined as:*

$$x \sqcup_{A \oplus B} y \stackrel{def}{=} \begin{cases} x \sqcup_A y & \textit{if } x, y \in A_{\not\perp} \\ x \sqcup_B y & \textit{if } x, y \in B_{\not\perp} \\ x & \textit{if } y = \bot_{A \oplus B} \\ y & \textit{if } x = \bot_{A \oplus B} \\ \top_{A \oplus B} & \textit{otherwise} \end{cases}$$

and its greatest lower bound $\sqcap_{A \oplus B}$ *can be dually defined. The abstraction functions* $\alpha_{A \oplus B} : C \cup D \to A \oplus B$ *is defined as:*

$$\alpha_{A \oplus B}(x) \stackrel{def}{=} \begin{cases} \alpha_A(x) & \textit{if } x \in C \\ \alpha_B(x) & \textit{if } x \in D \\ \top_{A \oplus B} & \textit{otherwise} \end{cases}$$

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 Figure 6.

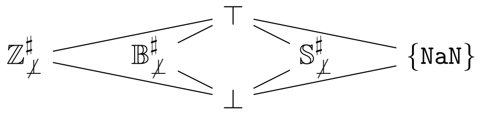

$\top$
$\mathbb{Z}^\sharp_{\not\perp}$
$\mathbb{B}^\sharp_{\not\perp}$
$\mathbb{S}^\sharp_{\not\perp}$
{NaN}
$\bot$

Figure 6. Coalesced sum abstract domain for μJS.

This is the simplest choice but unfortunately this is not suitable for dynamic languages, 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 μJS fragment `if` $(\texttt{y} < 5)$ $\{\texttt{x} = \text{“42”};\}$ `else` $\{\texttt{x} = \texttt{true};\}$. 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 $\alpha_{\mathbb{S}}(\text{“42”}) \sqcup \alpha_{\mathbb{B}}(\texttt{true}) = \top$.

4.2.2. Cartesian Product

In order to catch union types, without losing too much precision, we need to complete [15,16,32] 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 [33]. The complete abstract domain w.r.t. dynamic typing and implicit type conversion is: $\mathbb{Z}^\sharp \times \mathbb{B}^\sharp \times \mathbb{S}^\sharp \times \wp(\{\texttt{NaN}\})$, abstraction of $\wp(\mathbb{V})$. In this combined abstract domain, the value of `x` after the `if`-execution is precisely $(\bot, \alpha_{\mathbb{B}}(\texttt{true}), \alpha_{\mathbb{S}}(\text{“42”}), \bot)$, now an element of the domain, inferring that the value of `x` can be $\alpha_{\mathbb{B}}(\texttt{true})$ or $\alpha_{\mathbb{S}}(\text{“42”})$ but surely not an abstract integer of `NaN`.

In the following, we consider the abstract domain $\mathbb{V}^\sharp$ for string analysis obtained as Cartesian product of the following abstractions: $\mathbb{B}^\sharp = \wp(\{\texttt{true}, \texttt{false}\})$, $\mathbb{Z}^\sharp = \mathsf{Const} \stackrel{\text{def}}{=} \{\bot_{\mathsf{Const}}, \top_{\mathsf{Const}}\} \cup \{\ \{z\} \mid z \in \mathbb{Z}\ \}$ (the abstract domain of constant integers) and $\mathbb{S}^\sharp = \textsc{Dfa}_{/\equiv}$, .

5. Abstract Semantics of ECMAScript String Operations

In this section, we define the abstract semantics of the language μJS over the abstract domain $\mathbb{V}^\sharp$. In particular, we have to define the expressions abstract semantics $[\![\cdot]\!]^\sharp : \textsc{Exp} \times \textsc{States} \to \mathbb{V}^\sharp$, abstracting the collecting semantics (The string collecting semantics (fully reported in Appendix A) is defined lifting to $\wp(\mathbb{V})$ the concrete one reported in Section 3. For example, the collecting semantics of substring is, abusing notation, $\mathsf{Ss} : \wp(\Sigma^*) \times \wp(\mathbb{Z}) \times \wp(\mathbb{Z}) \to \wp(\Sigma^*)$ defined as

$\textsc{Ss}(\mathtt{L}, I, J) = \{\ \textsc{Ss}(\sigma, i, j) \mid \sigma, \in \mathtt{L}, i \in I, j \in J\ \}$.), which is standard except for the string operations that will be explicitly provided by describing the algorithms for computing them. Let us first recall some important notions on regular languages, useful for the algorithms we will provide.

Definition 2 (Suffixes and prefixes [12])**.** *Let* $\mathtt{L} \in \wp(\Sigma^*)$ *be a regular language. The suffixes of* $\mathtt{L}$ *are* $\textsc{Su}(\mathtt{L}) \overset{def}{=} \{\ y \in \Sigma^* \mid \exists x \in \Sigma^*.\ x \cdot y \in \mathtt{L}\ \}$*, 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 $\textsc{Su}(\mathtt{L}, i) \overset{\text{def}}{=} \{\ y \in \Sigma^* \mid \exists x \in \Sigma^*.\ x \cdot y \in \mathtt{L}, |x| = i\ \}$. For instance, let $\mathtt{L} = \{abc, hello\}$, then $\textsc{Su}(\mathtt{L}, 2) = \{c, llo\}$.

Definition 3 (Right quotient [12])**.** *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* $\textsc{Rq}(\mathtt{L}_1, \mathtt{L}_2) \overset{def}{=} \{\ x \in \Sigma^* \mid \exists y \in \mathtt{L}_2.\ x \cdot y \in \mathtt{L}_1\ \}$*.*

For example, let $\mathtt{L}_3 = \{xab, yab\}$ and $\mathtt{L}_4 = \{b, ab\}$. The right quotient of $\mathtt{L}_3$ w.r.t $\mathtt{L}_4$ is $\textsc{Rq}(\mathtt{L}_3, \mathtt{L}_4) = \{xa, ya, x, y\}$.

Definition 4 (Substrings/Factors [34])**.** *Let* $\mathtt{L} \in \wp(\Sigma^*)$ *be a regular language. The set of its substrings/factors is* $\textsc{Fa}(\mathtt{L}) \overset{def}{=} \{\ y \in \Sigma^* \mid \exists x, z \in \Sigma^*.\ x \cdot y \cdot z \in \mathtt{L}\ \}$*.*

These operations are all defined as transformations of regular languages. In [12], 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 holds:

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

5.1. *Abstract Semantics of* `Substring`

In this section we define the abstract semantics of `substring`. In particular, we define the operator $\mathsf{SS}^\sharp : \textsc{Dfa}_{/\equiv} \times \mathsf{Const} \times \mathsf{Const} \rightarrow \textsc{Dfa}_{/\equiv}$, that takes as input an automaton and two constant integer indexes $i, j \in \mathsf{Const}$, and computes the automaton recognizing the set of all substrings of the input automata language between the two provided integer indexes. Since the abstract semantics has to take into account the swaps when the initial index is greater than the final one, several cases arise when one of the two integer parameters is unknown, namely when it is equal to $\top_{\mathsf{Const}}$. Indeed, the abstract semantics $\mathsf{SS}^\sharp$ is divided in four cases that are reported in Table 1. Consider $\mathtt{A} \in \textsc{Dfa}_{/\equiv}, i, j \in \mathsf{Const}$ (for the sake of readability we denote by $\sqcup$ the automata lub $\sqcup_{\textsc{dfa}}$, and by $\sqcap$ the glb $\sqcap_{\textsc{dfa}}$). As in the concrete semantics of `substring`, negative integer values are treated as zero.

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

$$\mathsf{SS}(\mathtt{A}, i, j) \overset{\text{def}}{=} (\mathsf{RQ}(\mathsf{SU}(\mathtt{A}, i), \mathsf{SU}(\mathtt{A}, j)) \sqcap \mathsf{Min}(\Sigma^{j-i})) \sqcup (\mathsf{SU}(\mathtt{A}, i) \sqcap \mathsf{Min}(\Sigma^{<j-i}))$$

   If $j > i$, the integer arguments are simply swapped, as in the Table 1.
2. When both integer parameters correspond to $\top_{\mathsf{Const}}$, the result is the automaton of all possible factors of $\mathtt{A}$ (third row, third column), i.e., $\mathsf{FA}(\mathtt{A})$.

3. When i is defined and $j = \top_{\mathsf{Const}}$ (second row, third column), we have to compute the automaton recognizing all the substrings of $\mathscr{L}(\mathtt{A})$ from 0 to i and any substring starting from i. For example, let us consider $\mathsf{SS}^\sharp(\mathsf{Min}(\{\mathit{helloworld}\}), 5, \top_{\mathsf{Const}})$. Due to the semantics of `substring` reported in Section 3, we need to compute the substring from $a \in [0, 5]$ to 5 and then any substring with initial index equal to 5. The automata recognizing any substring starting at a specific index l is defined as $\mathsf{SS}^{\leftrightarrow}(\mathtt{A}, l) \overset{\text{def}}{=} \mathsf{FA}(\mathsf{SU}(\mathtt{A}, l))$. The abstract semantics returns the least upper bound of all the automata of substrings from a in $[0, i]$ to the automata recognizing any substring with initial index equals to i.
4. Similarly to the previous case, when j is defined and $i = \top_{\mathsf{Const}}$ (third row, second column), we have to compute the automaton recognizing all the substring of $\mathscr{L}(\mathtt{A})$ from 0 to j and any substring starting from j. Let us consider $\mathsf{SS}^\sharp(\mathsf{Min}(\{\mathit{helloworld}\}), \top_{\mathsf{Const}}, 5)$. Similarly to the previous case, we compute the substrings from $a \in [0, 5]$ to 5 and then any substring with initial index equal to 5. The abstract semantics therefore returns the least upper bound of all the automata of substrings from a in $[0, j]$ to the automata recognizing any substring with initial index equal to j.

In Figure 7 we report an example obtained applying the rules in the table.

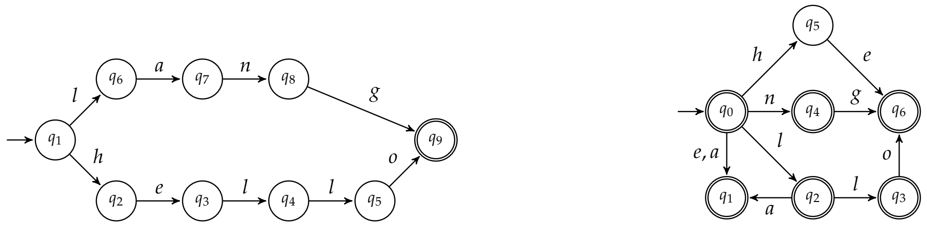
Table 1. Definition of $\mathsf{SS}^\sharp$.

$\mathsf{SS}^\sharp(\mathtt{A}, \mathbf{i}, \mathbf{j})$	$\mathbf{j} \in \mathbb{Z}$ $(\mathbf{j} \neq \top_{\mathsf{Const}})$	$\mathbf{j} = \top_{\mathsf{Const}}$
$\mathbf{i} \in \mathbb{Z}$ $(\mathbf{i} \neq \top_{\mathsf{Const}})$	$\mathsf{SS}(\mathtt{A}, \mathsf{min}(i, j), \mathsf{max}(i, j))$	$\bigsqcup_{a \in [0,i]} \mathsf{SS}(\mathtt{A}, a, i)$ $\sqcup$ $\mathsf{SS}^{\leftrightarrow}(\mathtt{A}, i)$
$\mathbf{i} = \top_{\mathsf{Const}}$	$\bigsqcup_{a \in [0,j]} \mathsf{SS}(\mathtt{A}, a, j)$ $\sqcup$ $\mathsf{SS}^{\leftrightarrow}(\mathtt{A}, j)$	$\mathsf{FA}(\mathtt{A})$

Theorem 2. $\mathsf{SS}^\sharp$ *is sound and complete. Formally,*

$$\forall \mathtt{A} \in \textsc{Dfa}_{/\equiv}, i, j \in \mathsf{Const}.\ \textsc{Ss}(\mathscr{L}(\mathtt{A}), \gamma(i), \gamma(j)) = \mathscr{L}(\mathsf{SS}^\sharp(\mathtt{A}, i, j))$$

From here on, when we say completeness we mean *forward completeness*. As highlighted in Section 2, this is the only form of completeness we can ensure in absence of a Galois connection. In particular, when an abstract operation (e.g., $\mathsf{SS}^\sharp$) is forward complete for a concrete operation (e.g., $\textsc{Ss}$) means that the computation on the abstract domain (e.g., $\textsc{Dfa}_{/\equiv}$) does not lose information due to the necessary computation only on abstract elements.



(a) **(b)**

Figure 7. **(a)** $\mathtt{A}$, $\mathscr{L}(\mathtt{A}) = \{\mathit{lang}, \mathit{hello}\}$ **(b)** $\mathtt{A}' = \mathsf{SS}^\sharp(\mathtt{A}, 2, \top_{\mathsf{Const}})$.

5.2. *Abstract Semantics of* `charAt`

The abstract semantics of `charAt` should return an automaton accepting the language of the characters at position i in the strings accepted by the given automaton. Since `charAt` is a particular case of `substring`, its abstract semantics, determined by $\mathsf{CA}^\sharp : \textsc{Dfa}_{/\equiv} \times \mathsf{Const} \rightarrow \textsc{Dfa}_{/\equiv}$, relies on the abstract semantic of `substring` previously defined. In particular,

$$\mathsf{CA}^\sharp(\mathtt{A}, i) \overset{\text{def}}{=} \begin{cases} \mathsf{SS}^\sharp(\mathtt{A}, i, i+1) & i \neq \top_{\mathsf{Const}} \\ \mathsf{Min}(\mathtt{chars}(\mathtt{A})) \sqcup \mathsf{Min}(\{\epsilon\}) & \text{otherwise} \end{cases}$$

We call $\mathsf{SS}^\sharp$ (defined before) when the index i corresponds to a determinate integer value otherwise we use the function $\mathtt{chars} : \textsc{Dfa}_{/\equiv} \rightarrow \wp(\Sigma)$, returning the set of characters read in any transition of an automaton, together with $\mathsf{Min}(\{\epsilon\})$.

Theorem 3. $\mathsf{CA}^\sharp$ *is sound and complete. Formally,*

$$\forall \mathtt{A} \in \textsc{Dfa}_{/\equiv}, i \in \mathsf{Const}.\, \textsc{Ca}(\mathscr{L}(\mathtt{A}), \gamma(i)) = \mathscr{L}(\mathsf{CA}^\sharp(\mathtt{A}, i))$$

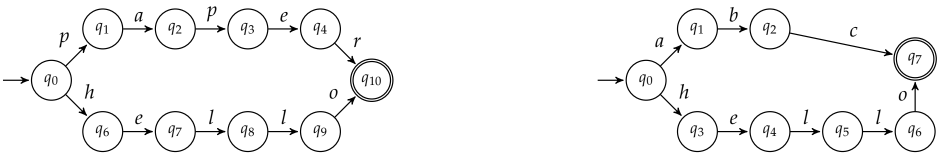
5.3. *Abstract Semantics of* `length`

The abstract semantics of `length` should return a value, of the integer domain Const, that, in a sound way, approximates the length of all the possible strings of an automaton. The abstract semantics of `length` is defined by the function $\mathsf{LE}^\sharp : \textsc{Dfa}_{/\equiv} \rightarrow \mathsf{Const}$, computed by Algorithm 1, where $\mathsf{Paths} : \textsc{Dfa}_{/\equiv} \rightarrow \wp(\wp(Q))$ returns the set of the paths from the initial state to any final state of A [35]. Given a path $\mathsf{p} \in \mathsf{Paths}(\mathtt{A})$, we denote by $|\mathsf{p}|$ the length of p.

Algorithm 1: $\mathsf{LE}^\sharp : \textsc{Dfa}_{/\equiv} \rightarrow \mathsf{Const}$ algorithm

```
   Input: $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$
   Output: $\mathsf{LE}^\sharp(\mathtt{A})$
 1 if hasCycle(A) then
 2     return $\top_{\mathsf{Const}}$;
 3 else
 4     lenghts ← $-1$;
 5     foreach p ∈ Paths(A) do
 6         if $|\mathsf{p}|$ == lenghts ∨ lenghts == $-1$ then
 7             lenghts ← $|\mathsf{p}|$;
 8         else
 9             return $\top_{\mathsf{Const}}$;
10         end
11     end
12     return lenghts;
13 end
```

If the input automaton has cycles, $\mathsf{LE}^\sharp$ returns $\top_{\mathsf{Const}}$ otherwise it checks that any path of the automaton A has the same length (lines 5–8). Whenever the algorithm finds that there exists two paths in the automaton that have different lengths, $\top_{\mathsf{Const}}$ is returned (lines 8–10). Due to the constant integers domain, the abstract semantics of `length` can give a precise answer only when any string of the automaton has precisely the same length. More accurate results can be obtained by using more precise integer abstract domains, e.g., intervals, as we will discuss in Section 6. For example, consider the automata A and A′ in Figure 8a,b, respectively. $\mathsf{LE}^\sharp(\mathtt{A})$ precisely returns 5, since all the strings recognized by A have the same length, while $\mathsf{LE}^\sharp(\mathtt{A}')$ returns $\top_{\mathsf{Const}}$.



(**a**) (**b**)

Figure 8. (**a**) $\mathtt{A}$, $\mathscr{L}(\mathtt{A}) = \{paper, hello\}$. (**b**) $\mathtt{A}'$, $\mathscr{L}(\mathtt{A}') = \{abc, hello\}$.

Theorem 4. $\mathsf{LE}^\sharp$ *is sound and complete. Formally,*

$$\forall \mathtt{A} \in \textsc{Dfa}_{/\equiv}.\ \textsc{Le}(\mathscr{L}(\mathtt{A})) = \gamma(\mathsf{LE}^\sharp(\mathtt{A}))$$

5.4. *Abstract Semantics of* `Concat`

The abstract semantics of string concatenation is $\mathsf{CC}^\sharp : \textsc{Dfa}_{/\equiv} \times \textsc{Dfa}_{/\equiv} \rightarrow \textsc{Dfa}_{/\equiv}$ and returns the concatenation between the input automata. Since regular languages are closed under the concatenation operation, so are finite state automata. Hence, $\mathsf{CC}^\sharp$ exactly implements the standard concatenation operation between automata. Given the closure property on automata, the following result holds.

Theorem 5. *The function* $\mathsf{CC}^\sharp$ *is sound and complete. Formally,* $\forall \mathtt{A}, \mathtt{A}' \in \textsc{Dfa}_{/\equiv}$.

$$\textsc{Cc}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \mathscr{L}(\mathsf{CC}^\sharp(\mathtt{A}, \mathtt{A}'))$$

As we have already mentioned before, completeness holds thanks to the closure properties of regular languages (and in turn of finite state automata).

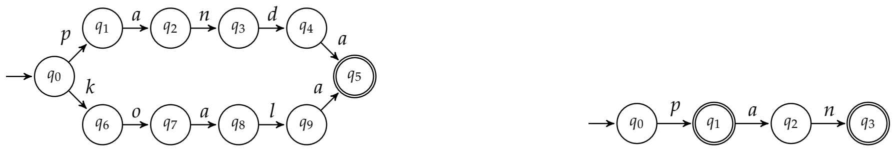
5.5. *Abstract Semantics of* `StartsWith`

The abstract semantics of `startsWith` takes as input two automata and checks whether a string of the language of the first automaton starts with a string of the language of the second one. The abstract semantics of `startsWith` is captured by the function $\mathsf{SW}^\sharp : \textsc{Dfa}_{/\equiv} \times \textsc{Dfa}_{/\equiv} \rightarrow \mathbb{B}^\sharp$, computed by Algorithm 2, where $\mathsf{maxString} : \textsc{Dfa}_{/\equiv} \rightarrow \textsc{Dfa}_{/\equiv}$ returns the (minimal) automaton recognizing the longest string of the automaton given as input and $\mathsf{isSinglePath} : \textsc{Dfa}_{/\equiv} \rightarrow \{\mathtt{true}, \mathtt{false}\}$ checks whether the input automata $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$ respect the following condition: $\delta = \bigcup_{i \in [0, |Q|]} (q_i, q_{i+1}, c)$. Informally, a single-path automaton is an automaton where, if we sort the strings of its language from the shortest to the longest, each string is a prefix of the next one. An example of a single-path automaton is reported in Figure 9b where it is graphically clear that each state, excluding the initial and last one, have one incoming and one outgoing transition. Since the longest string in a single-path automaton has, as prefix, all the others of the language, it is sufficient to check, for an automaton $\mathtt{A}$, if it starts with only the former. For example, let $\mathscr{L}(\mathtt{A}) = \{softer\}$ and $\mathscr{L}(\mathtt{A}') = \{s, so, soft\}$. The string s is prefix of so, which is in turn prefix of $soft$ so $\mathtt{A}'$ is a single-path automaton. Therefore, in this case, it is sufficient to check if $softer$ starts with only $soft$ (the longest string of $\mathscr{L}(\mathtt{A}')$) since, being $\mathtt{A}'$ single-path, the other strings (s and so) are consequently prefix of $softer$. Instead, consider $\mathscr{L}(\mathtt{A}') = \{s, no\}$. It would be impossible for a string to start with both of them since there is no prefix relation between them.

Algorithm 2 takes as input two automata denoted by $\mathtt{A}$ and $\mathtt{A}'$. Lines 1-9 handle some corner cases. If $\mathscr{L}(\mathtt{A}') = \{\varepsilon\}$, $\{\mathtt{true}\}$ is returned, since any string starts with ε (lines 1-3). If none of the prefixes of $\mathtt{A}$ is recognized by $\mathtt{A}'$, meaning that none of the strings recognized by $\mathtt{A}$ start with a string of $\mathtt{A}'$, we can safely return $\{\mathtt{false}\}$ (lines 4-6). Finally, if at least one of the input automata have cycles, we return $\{\mathtt{true}, \mathtt{false}\}$ (lines 7-9). Lines 10-17 determine if any string of $\mathtt{A}'$ is the beginning of any string of $\mathtt{A}$, otherwise $\top_{\mathsf{Bool}}$ is returned.

Algorithm 2: $\mathsf{SW}^\sharp : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \to \mathbb{B}^\sharp$ algorithm

```
    Input  : $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$, $A' = (Q', \Sigma, \delta', q'_0, F')$
    Output : $\mathsf{SW}^\sharp(\mathtt{A}, \mathtt{A}')$
 1  if $\mathtt{A}' == \mathsf{Min}(\{\epsilon\})$ then
 2      return $\{\mathtt{true}\}$;
 3  end
 4  if $\mathsf{PR}(\mathtt{A}) \sqcap_{\text{DFA}} \mathtt{A}' == \mathsf{Min}(\varnothing)$ then
 5      return $\{\mathtt{false}\}$;
 6  end
 7  if $\mathsf{hasCycle}(\mathtt{A}) \vee \mathsf{hasCycle}(\mathtt{A}')$ then
 8      return $\top_{\mathsf{Bool}}$;
 9  end
10  if $\mathsf{isSinglePath}(\mathtt{A}')$ then
11      $\mathtt{B} \leftarrow \mathsf{maxString}(\mathtt{A}')$;
12      $\mathtt{C} \leftarrow \mathsf{SS}^\sharp(\mathtt{A}, 0, \mathsf{LE}^\sharp(\mathtt{B}))$;
13      if $\mathtt{B} == \mathtt{C}$ then
14          return $\{\mathtt{true}\}$;
15      end
16  end
17  return $\top_{\mathsf{Bool}}$;
```



(**a**) (**b**)

Figure 9. (**a**) $\mathtt{A}$, $\mathscr{L}(\mathtt{A}) = \{panda, koala\}$. (**b**) $\mathtt{A}'$, $\mathscr{L}(\mathtt{A}') = \{pan, p\}$.

In order to explain our approach in lines 10-17, consider the automata $\mathtt{A}$ and $\mathtt{A}'$ reported in Figure 9. To be sure that any string recognized by $\mathtt{A}'$ is the beginning of any string recognized by $\mathtt{A}$ we need to check two conditions: (1) any string recognized by $\mathtt{A}'$ is prefix of its longest recognized string σ and (2) each string in $\mathtt{A}$ starts with σ (all strings must have a common prefix). Only if both conditions occur we can safely return $\{\mathtt{true}\}$ otherwise we return $\top_{\mathsf{Bool}}$. In particular, (1) is checked by the function $\mathsf{isSinglePath}$ at line 10 and (2) is checked at lines 11-15. It is worth noting that if an automaton is single-path, then the longest string is unique (line 11).

In our example, both the strings p and pan in $\mathscr{L}(\mathtt{A}')$ are prefixes of pan, which is the longest string recognized by $\mathtt{A}'$, so we build $\mathtt{B}$, which is the (minimal) automaton that recognizes pan and $\mathtt{C}$, $\mathscr{L}(\mathtt{C}) = \{pan, koa\}$, and compare them (line 13). We return $\{\mathtt{true}\}$ if $\mathtt{B}$ and $\mathtt{C}$ recognize the same language otherwise we return $\top_{\mathsf{Bool}}$. In the other cases, as already mentioned, we return $\{\mathtt{true}, \mathtt{false}\}$. For example, in Figure 9, $\{\mathtt{true}, \mathtt{false}\}$ is returned because, although $\mathtt{A}'$ is a single-path automaton, only the string $panda \in \mathscr{L}(\mathtt{A})$ begins with pan, namely the longest string of $\mathscr{L}(\mathtt{A}')$.

Example 2. *Consider for example* $\mathtt{A}$ *s.t.* $\mathscr{L}(\mathtt{A}) = \{panda, panem\}$, *and* $\mathtt{A}'$ *s.t.* $\mathscr{L}(\mathtt{A}') = \{p, pan\}$. $\mathsf{SW}^\sharp(\mathtt{A}, \mathtt{A}')$ *returns* $\{\mathtt{true}\}$ *since* $\mathtt{A}'$ *is a single-path automaton and both strings of* $\mathtt{A}$ *start with the longest string in* $\mathtt{A}'$, *the string pan. Consider instead the automata* $\mathtt{A}, \mathscr{L}(\mathtt{A}) = \{panda, koala\}$, *and* $\mathtt{A}, \mathscr{L}(\mathtt{A}) = \{p, k\}$. *In this case,* $\mathsf{SW}^\sharp(\mathtt{A}, \mathtt{A}')$ *returns* $\{\mathtt{true}, \mathtt{false}\}$ *since* $\mathtt{A}'$ *is not a single-path automaton. Indeed, we can easily check that even if the string panda* $\in \mathscr{L}(\mathtt{A})$ *starts with* $p \in \mathscr{L}(\mathtt{A}')$, *the string koala* $\in \mathscr{L}(\mathtt{A})$ *does not.*

Theorem 6. $\mathsf{SW}^\sharp$ *is sound but not complete. Formally,*

$$\forall \mathtt{A}, \mathtt{A}' \in \text{DFA}_{/\equiv}.\, \mathsf{SW}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) \subsetneq \mathsf{SW}^\sharp(\mathtt{A}, \mathtt{A}')$$

As a counterexample to completeness, consider the automata $\mathtt{A}$ s.t. $\mathscr{L}(\mathtt{A}) = \mathtt{A} = \{\ a^n \mid n > 1\ \}$ and $\mathtt{A}'$ s.t. $\mathscr{L}(\mathtt{A}') = \{a\}$. The completeness condition is not met, indeed,

$$\mathsf{SW}^\sharp(\mathtt{A},\mathtt{A}') = \{\mathtt{true},\mathtt{false}\} \not\subseteq \textsc{Sw}(\mathscr{L}(\mathtt{A}),\mathscr{L}(\mathtt{A}')) = \{\mathtt{true}\}$$

5.6. Abstract Semantics of `ToLowerCase`

The abstract semantics of `toLowerCase` is defined by the function $\mathsf{LC}^\sharp : \textsc{Dfa}_{/\equiv} \to \textsc{Dfa}_{/\equiv}$ which returns as result an automaton that recognizes the same strings of the input automaton, where any upper-case symbol is replaced with the corresponding lower-case symbol. $\mathsf{LC}^\sharp$ is computed by Algorithm 3.

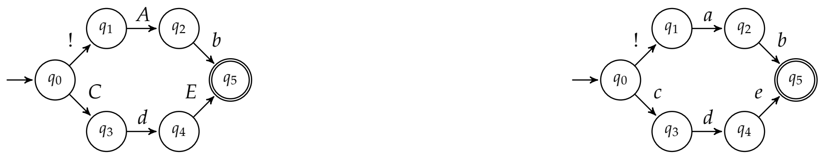
Algorithm 3: $\mathsf{LC}^\sharp : \textsc{Dfa}_{/\equiv} \to \textsc{Dfa}_{/\equiv}$ algorithm

```
Input  : $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$
Output: $\mathsf{LC}^\sharp(\mathtt{A})$
$\delta' \leftarrow \varnothing$;
foreach $(q, c, q') \in \delta$ do
    $\delta' \leftarrow \delta' \cup (q, \mathsf{toLowerCase}(c), q')$
end
$\mathtt{A}' \leftarrow (Q, \Sigma, \delta', q_0, F)$;
return $\mathtt{A}'$
```

Starting from an input automaton $\mathtt{A}$, the idea is to return as result the automaton $\mathtt{A}'$, that is a copy of $\mathtt{A}$ with the exception that any upper-case symbol read by a transition is replaced by its corresponding lower-case symbol. Transitions that already read lower-case or special symbols are unaltered. An example is reported in Figure 10.

Theorem 7. $\mathsf{LC}^\sharp$ *is both sound and complete. Formally,*

$$\forall \mathtt{A} \in \textsc{Dfa}_{/\equiv}.\ \textsc{Lc}(\mathscr{L}(\mathtt{A})) = \mathscr{L}(\mathsf{LC}^\sharp(\mathtt{A}))$$



(a) **(b)**

Figure 10. **(a)** $\mathtt{A}$, $\mathscr{L}(\mathtt{A}) = \{!Ab, CdE\}$, **(b)** $\mathsf{LC}^\sharp(\mathtt{A})$.

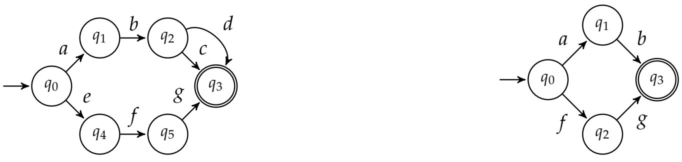
5.7. Abstract Semantics of `Includes`

The abstract semantics of `includes` is defined by the function $\mathsf{IN}^\sharp : \textsc{Dfa}_{/\equiv} \times \textsc{Dfa}_{/\equiv} \to \mathbb{B}^\sharp$. It takes as input two automata $\mathtt{A}$ and $\mathtt{A}'$ and checks whether a string recognized by $\mathtt{A}'$ is a substring of a string recognized by $\mathtt{A}$. The function $\mathsf{IN}^\sharp$ is computed by Algorithm 4, where, given a path $\mathtt{p}$ of an automaton $\mathtt{A}$, we abuse notation denoting by $\mathsf{Min}(\mathtt{p})$ the automaton that recognizes the string encoded by the path $\mathtt{p}$ (lines 11–12). The algorithm first checks some corner cases: if $\mathtt{A}'$ only recognizes the empty string, $\{\mathtt{true}\}$ is returned, since the empty string is always a substring of a non-empty automaton (lines 2–4), if none of the substring of $\mathtt{A}$ is contained in $\mathtt{A}'$, $\{\mathtt{false}\}$ is returned (lines 5–7) and if one of the input automata is cyclic, it returns $\top_{\mathsf{Bool}}$ (lines 8–10). When these corner cases are excluded, we check each string recognized by $\mathtt{A}$. If the algorithm finds at least one string σ' in $\mathscr{L}(\mathtt{A}')$ that is not a substring of a string σ of $\mathtt{A}$, $\top_{\mathsf{Bool}}$ is returned otherwise $\{\mathtt{true}\}$. This is done in lines 10–14 where, for each path $\mathtt{p}$ of $\mathtt{A}$ we create $\mathsf{Min}(\mathtt{p})$ and check if its factorization with $\mathtt{A}'$ equals $\mathtt{A}'$, i.e., we check if it contains any string of $\mathtt{A}'$.

Algorithm 4: $\mathsf{IN}^\sharp : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \to \mathbb{B}^\sharp$ algorithm

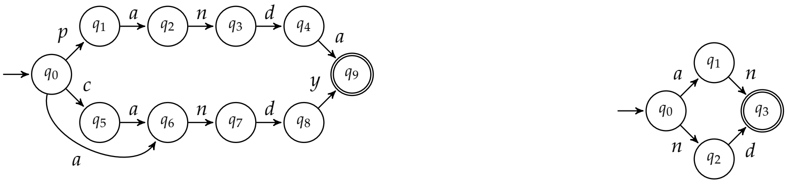
```
Input  : $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$, $\mathtt{A}' = (Q', \Sigma, \delta', q'_0, F')$
Output : $\mathsf{IN}^\sharp(\mathtt{A}, \mathtt{A}')$
if $\mathtt{A}' == \mathsf{Min}(\{\varepsilon\})$ then
    return $\{\mathtt{true}\}$;
end
if $\mathsf{FA}(\mathtt{A}) \sqcap_{\text{DFA}} \mathtt{A}' == \mathsf{Min}(\varnothing)$ then
    return $\{\mathtt{false}\}$;
end
if $\mathsf{hasCycle}(\mathtt{A}) \lor \mathsf{hasCycle}(\mathtt{A}')$ then
    return $\top_{\mathsf{Bool}}$;
end
foreach $\mathsf{p} \in \mathsf{Paths}(\mathtt{A})$ do
    if $\mathtt{A}' \sqcap_{\text{DFA}} \mathsf{FA}(\mathsf{Min}(\mathsf{p})) \neq \mathtt{A}'$ then
        return $\top_{\mathsf{Bool}}$;
    end
end
return $\{\mathtt{true}\}$;
```

For example, consider the automata $\mathtt{A}$ and $\mathtt{A}'$ reported in Figure 11. The algorithm returns $\top_{\mathsf{Bool}}$ since the string $fg \in \mathscr{L}(\mathtt{A}')$ is not a substring of $abc \in \mathtt{A}$. Another example is reported in Figure 12. The result of $\mathsf{IN}^\sharp(\mathtt{A}, \mathtt{A}')$ returns $\{\mathtt{true}\}$ since $\forall \sigma \in \mathscr{L}(\mathtt{A}), \forall \sigma' \in \mathscr{L}(\mathtt{A}').\, \sigma'$ is a substring of σ.



(**a**) (**b**)

Figure 11. (**a**) $\mathtt{A}, \mathscr{L}(\mathtt{A}) = \{abc, abd, efg\}$ (**b**) $\mathtt{A}', \mathscr{L}(\mathtt{A}') = \{ab, fg\}$.



(**a**) (**b**)

Figure 12. (**a**) $\mathtt{A}, \mathscr{L}(\mathtt{A}) = \{panda, candy, andy\}$ (**b**) $\mathtt{A}', \mathscr{L}(\mathtt{A}') = \{an, nd\}$.

Theorem 8. $\mathsf{IN}^\sharp$ *is sound but not complete. Formally,*

$$\forall \mathtt{A}, \mathtt{A}' \in \text{DFA}_{/\equiv}.\, \text{IN}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) \subsetneq \mathsf{IN}^\sharp(\mathtt{A}, \mathtt{A}').$$

As a counterexample to completeness, consider the automaton $\mathtt{A}$ s.t. $\mathscr{L}(\mathtt{A}) = \{\, a^n \mid n > 1 \,\}$ and the automaton $\mathtt{A}'$ s.t. $\mathscr{L}(\mathtt{A}') = \{a\}$. The completeness condition is not met, indeed

$$\text{IN}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \{\mathtt{true}\} \neq \mathsf{IN}^\sharp(\mathtt{A}, \mathtt{A}') = \{\mathtt{true}, \mathtt{false}\}$$

since when one of the input automata is cyclic, Algorithm 4 returns $\top_{\mathsf{Bool}}$.

5.8. Abstract Semantics of `Repeat`

The abstract semantics of `repeat` is defined by the function $\mathsf{RT}^\sharp : \textsc{Dfa}_{/\equiv} \times \mathsf{Const} \rightarrow \textsc{Dfa}_{/\equiv}$ that, given as input an automaton $\mathtt{A}$ and a constant integer value i, returns an automaton that recognizes any string of $\mathscr{L}(\mathtt{A})$ repeated i times. $\mathsf{RT}^\sharp$ is computed by Algorithm 5 and we suppose that the abstract integer value i is positive or zero. Any non-positive value is treated as zero. The algorithm first checks some corner cases. If $i = 0$ or the input automaton only recognizes the empty string, then $\mathsf{Min}(\epsilon)$ is returned (lines 1–3). If the automaton has a cycle or $i = \top_{\mathsf{Const}}$, it returns the Kleene-closure of the input automaton (lines 4–6). If none of these corner cases is detected then, for each string in $\mathscr{L}(\mathtt{A})$, we concatenate it with itself $(i-1)$-times using the already defined $\mathsf{CC}^\sharp$. The result is the union of all the concatenated automata.

Algorithm 5: $\mathsf{RT}^\sharp : \textsc{Dfa}_{/\equiv} \times \mathsf{Const} \rightarrow \textsc{Dfa}_{/\equiv}$ algorithm

```
   Input  : $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$, $i \in \mathsf{Const}$
   Output : $\mathsf{RT}^\sharp(\mathtt{A}, i)$
 1 if $i == 0 \vee \mathtt{A} == \mathsf{Min}(\{\epsilon\})$ then
 2     return $\mathsf{Min}(\{\varepsilon\})$;
 3 end
 4 if $\mathsf{hasCycle}(\mathtt{A}) \vee i == \top_{\mathsf{Const}}$ then
 5     return $\mathsf{Kleene}(\mathtt{A})$;
 6 end
 7 $\mathtt{R} \leftarrow \varnothing$;
 8 foreach $\mathsf{p} \in \mathsf{Paths}(\mathtt{A})$ do
 9     $\mathtt{A}' \leftarrow \mathsf{Min}(\mathsf{p})$;
10     $\mathtt{B} \leftarrow \mathtt{A}'$;
11     foreach $k \in [1, i-1]$ do
12         $\mathtt{A}' \leftarrow \mathsf{CC}^\sharp(\mathtt{A}', \mathtt{B})$;
13     end
14     $\mathtt{R} \leftarrow \mathtt{R} \cup \mathtt{A}'$;
15 end
16 return $\mathtt{R}$;
```

Let us consider the automaton $\mathtt{A}$ reported in Figure 13a and suppose to call $\mathsf{RT}^\sharp(\mathtt{A}, 2)$. The resulting automaton, applying Algorithm 5, is reported in Figure 13b. Let us suppose to call $\mathsf{RT}^\sharp(\mathtt{A}, \top_{\mathsf{Const}})$. In this case, since the input integer value is not determinate, Algorithm 5 returns the Kleene-star automaton of $\mathtt{A}$ and the result is reported in Figure 13c.

Theorem 9. $\mathsf{RT}^\sharp$ *is sound but not complete. Formally,*

$$\forall \mathtt{A} \in \textsc{Dfa}_{/\equiv}, \forall i \in \mathsf{Const}.\ \textsc{Rt}(\mathscr{L}(\mathtt{A}), \gamma(i)) \subsetneq \mathscr{L}(\mathsf{RT}^\sharp(\mathtt{A}, i))$$

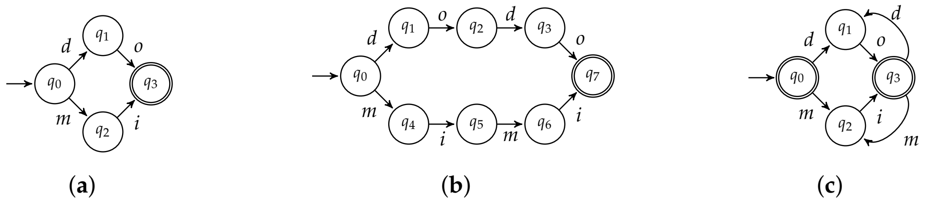


Figure 13. (**a**) $\mathtt{A}, \mathscr{L}(\mathtt{A}) = \{do, mi\}$ (**b**) $\mathsf{RT}^\sharp(\mathtt{A}, 2)$ (**c**) $\mathsf{RT}^\sharp(\mathtt{A}, \top_{\mathsf{Const}})$.

As a counterexample to completeness, consider the automaton $\mathtt{A}$ s.t. $\mathscr{L}(\mathtt{A}) = \{\ ab^n \mid n \in \mathbb{N}\ \}$. The completeness condition is not met, indeed

$$\textsc{Rt}(\mathscr{L}(\mathtt{A}), 2) = \{\ ab^n ab^n \mid n \in \mathbb{N}\ \} \neq \mathsf{RP}^\sharp(\mathtt{A}, 2) = \{\ (ab^n)^m \mid n, m \in \mathbb{N}\ \}$$

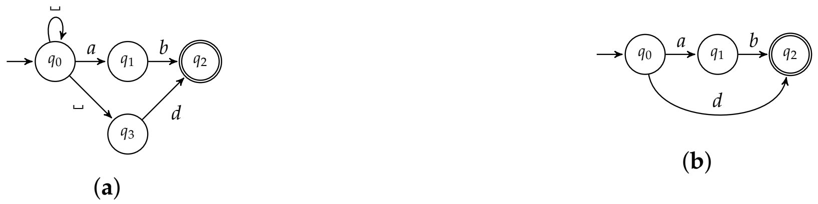
since when the input automaton is cyclic, Algorithm 5 returns the Kleene closure of the input automaton.

5.9. *Abstract Semantics of* `TrimLeft`*,* `TrimRight` *and* `Trim`

In this section, we will show the abstract semantics of `trimLeft`, `trimRight` and `trim` operations. The abstract semantics of `trimLeft` is defined by the function $\mathsf{TL}^\sharp : \textsc{Dfa}_{/\equiv} \rightarrow \textsc{Dfa}_{/\equiv}$. In particular, it takes as input an automaton $\mathtt{A}$ and returns an automaton accepting the same strings of $\mathtt{A}$ removing, at the beginning of each string, consecutive white spaces, if present. In the following, we denote a white-space as ␣. The function is computed by Algorithm 6. The idea of algorithm is to iteratively replace white-space transitions from the initial state with ϵ-transition (lines 5–7), while leaving the other transitions unaltered (lines 7–9). At each iteration, the resulting automaton is minimized, and hence determinized (line 11). This operation is repeated until the initial state has no white-space transitions, checking the condition that white-space is not a prefix of the automaton (line 3). In Figure 14 is depicted an example of application of our algorithm.

Algorithm 6: $\mathsf{TL}^\sharp : \textsc{Dfa}_{/\equiv} \rightarrow \textsc{Dfa}_{/\equiv}$ algorithm

```
   Input  : $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$
   Output : $\mathsf{TL}^\sharp(\mathtt{A})$
1  $\delta' \leftarrow \delta$;
2  $\mathtt{R} \leftarrow \mathtt{A}$;
3  while $\mathsf{PR}(\mathtt{R}) \sqcap_{\textsc{Dfa}} \mathsf{Min}(\{␣\})$ do
4      foreach $(q_0, c, q) \in \delta$ do
5          if $c ==$ ␣ then
6              $\delta' \leftarrow \delta \cup (q_0, \epsilon, q)$
7          else
8              $\delta' \leftarrow \delta \cup (q_0, c, q)$
9          end
10         $\mathtt{R} \leftarrow (Q, \Sigma, \delta', q_0, F)$;
11         $\mathsf{Min}(\mathtt{R})$;
12     end
13 end
14 return $\mathtt{R}$;
```



(**a**) (**b**)

Figure 14. (**a**) $\mathtt{A}$, $\mathscr{L}(\mathtt{A}) = \{(␣)^* ab, ␣d\}$, (**b**) $\mathsf{TL}^\sharp(\mathtt{A})$.

Theorem 10. $\mathsf{TL}^\sharp$ *is sound and complete. Formally,* $\forall \mathtt{A} \in \textsc{Dfa}_{/\equiv}$*,*

$$\textsc{Tl}(\mathscr{L}(\mathtt{A})) = \mathscr{L}(\mathsf{TL}^\sharp(\mathtt{A}))$$

The abstract semantics of `trimRight` can be defined in function of the already defined function $\mathsf{TL}^\sharp$. Indeed, the abstract semantics $\mathsf{TR}^\sharp : \text{DFA}_{/\equiv} \to \text{DFA}_{/\equiv}$ reserves the input automaton, applies $\mathsf{TL}^\sharp$ and finally reverses again the so obtained automaton. Formally,

$$\mathsf{TR}^\sharp \overset{\text{def}}{=} \mathsf{reverse}(\mathsf{TL}^\sharp(\mathsf{reverse}(\mathtt{A}))$$

Similarly, the abstract semantics of `trim` applies both the abstract semantics of `trimLeft` and `trimRight`. Thus, the abstract semantics of `trim` is captured by the function $\mathsf{TM}^\sharp : \text{DFA}_{/\equiv} \to \text{DFA}_{/\equiv}$ and it is defined as

$$\mathsf{TM}^\sharp(\mathtt{A}) \overset{\text{def}}{=} \mathsf{TR}^\sharp(\mathsf{TL}^\sharp(\mathtt{A}))$$

Theorem 11. $\mathsf{TR}^\sharp$ *and* $\mathsf{TM}^\sharp$ *are sound and complete. Formally,* $\forall \mathtt{A} \in \text{DFA}_{/\equiv}$

$$\textsc{Tr}(\mathscr{L}(\mathtt{A})) = \mathscr{L}(\mathsf{TR}^\sharp(\mathtt{A})) \qquad \textsc{Tm}(\mathscr{L}(\mathtt{A})) = \mathscr{L}(\mathsf{TM}^\sharp(\mathtt{A}))$$

Proof. The proof of $\mathsf{TR}^\sharp$ follows from the completeness of $\mathsf{TL}^\sharp$ and reverse operations, while the proof of $\mathsf{TM}^\sharp$ follows from the completeness of $\mathsf{TL}^\sharp$ and $\mathsf{TR}^\sharp$. □

5.10. Concerning Abstract Implicit Type Conversion

In this section, we discuss the abstraction of implicit type conversion functions. Here we will focus only on the conversion of automata into other values, since conversions concerning booleans, not-a-number and integers are standard. Let $\mathtt{toBool}^\sharp : \mathbb{V}^\sharp \to \mathbb{B}^\sharp$ be applied to $\mathtt{A} \in \text{DFA}_{/\equiv}$: If $\mathtt{A} \sqcap \mathsf{Min}(\{\epsilon\}) = \mathsf{Min}(\varnothing)$, it returns $\{\mathtt{true}\}$, when $\mathtt{A} = \mathsf{Min}(\{\epsilon\})$ the function returns $\{\mathtt{false}\}$, otherwise the function return $\top_{\mathsf{Bool}}$. Implicit type conversion to $\text{DFA}_{/\equiv}$ is handled by the function $\mathtt{toStr}^\sharp : \mathbb{V}^\sharp \to \text{DFA}_{/\equiv}$. As far as non numeric strings are concerned, $\mathtt{toStr}^\sharp$ returns $\mathsf{Min}(\{\mathtt{NaN}\})$. If the input is the boolean value `true` [`false`] it returns $\mathsf{Min}(\{\mathtt{true}\})$ [$\mathsf{Min}(\{\mathtt{false}\})$], otherwise it returns $\mathsf{Min}(\{\mathtt{true}\}) \sqcup \mathsf{Min}(\{\mathtt{false}\})$. Regarding abstract integers, if $i \in \mathbb{Z}$, then the automaton recognizing the string $\mathcal{S}(i)$ is returned (We recall that the function $\mathcal{S}(i)$ maps an integer i to its numeric string representation.), otherwise, hence when $i = \top_{\mathsf{Const}}$, the automaton recognizing any possible integer is returned and reported in Figure 15. Finally, $\mathtt{toInt}^\sharp : \mathbb{V}^\sharp \to \mathsf{Const} \cup \{\mathtt{NaN}\}$ handles conversion to constant integers. Given an automaton $\mathtt{A}$, if $\mathtt{A} \sqcap \mathsf{Min}(\Sigma_{\mathbb{Z}}) = \mathsf{Min}(\varnothing)$, the automaton is precisely converted to `NaN`, since `A` does not recognize any numerical string. Otherwise, if $\mathtt{A} \sqsubseteq_{\text{DFA}} \mathsf{Min}(\Sigma_{\mathbb{Z}})$ it means that $\mathscr{L}(\mathtt{A})$ contains only numeric strings. In particular, if `A` recognizes only one numerical string, the corresponding integer is returned, otherwise $\top_{\mathsf{Const}}$ is returned.

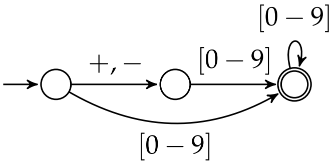


Figure 15. $\mathsf{toStr}^\sharp(\top_{\mathsf{Const}})$.

6. μFASA Implementation

In this section we present μJS Finite-state Automata String Analyzer (μFASA), the string static analyzer integrating the finite state automata abstract domain, and the corresponding abstract semantics, presented in the previous sections.

6.1. Theoretical Concerns

It is worth noting that, as reported in Theorem 1, $\wp(\Sigma^*)$ (string concrete domain) and $\text{DFA}_{/\equiv}$ (abstract string domain) do not form a Galois connection, however 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.

When implementing, an important issue is computational complexity. The abstract semantics reported in this paper often relies on minimization of finite state automata in order to keep the automata, which arise during abstract computations, determinized and minimized. In the worst case, minimization has exponential complexity but this is not a problem. Even if our library relies on the Brzozowski's algorithm, which theoretically has exponential complexity in worst-case scenario, in practice it is extremely fast on average and consistently outperforms other minimization algorithms (e.g., Hopcroft's algorithm, having average-case complexity $O(n \log n)$, where n is the number of states), as reported in [36]. Moreover, the minimization is only applied when the input automaton is not-deterministic.

6.2. Implementation

μFASA is a string static analyzer for extended μJS inter-procedural programs and it is built upon the finite state automata abstract domain described above. (Available at www.github.com/SPY-Lab/mufasa) It analyzes string variables and is also able to express associative arrays. The finite state automata abstract domain has been implemented as an external library (Available at www.github.com/SPY-Lab/java-fsm-library), offering a suitable and easy way to plug the domain into existing static analyzers, such as [3–5,37]. The library includes the implementation of all the algorithms concerning the finite state automata domain and provides well-known operations on automata such as suffix, right quotient, abstract domain-related operations, such as $\sqcup_{\text{DFA}}$, $\sqcap_{\text{DFA}}$, and the parametric widening for tuning precision and forcing convergence.

In addition to the string operations (and the corresponding automata-based abstract semantics) introduced in this paper, μFASA also analyzes functions that can be defined as composition of the ones presented here (e.g., `endsWith` w.r.t. `startsWith`, `slice` w.r.t. `substring`). The full list of implemented string operations is reported in Table 2, also summarizing for which operations holds soundness and completeness and the average complexity of their algorithms (w.r.t. the constant integer abstract domain).

Table 2. μJS Finite-state Automata String Analyzer (μFASA) string operations.

String Operation	Soundness	Completeness	Average Complexity
`substring`	✓	✓	$O(n \log n)$
`charAt`	✓	✓	$O(n \log n)$
`length`	✓	✓	$O(n + m)$
`concat`	✓	✓	$O(n \log n + n + m)$
`startsWith` `endsWith`	✓	✗	$O(n \log n + n + m)$
`toLowerCase` `toUpperCase`	✓	✓	$O(m)$
`includes`	✓	✗	$O(nlogn + n + m)$
`repeat`	✓	✗	$O(n \log n + n + m)$
`replace`	✓	✗	$O((n + m)nlogn)$
`indexOf`	✓	✗	$O(n(n \log n)(n^2 m))$
`slice`	✓	✗	$O((n + m)(n \log n))$

6.3. Extension to Interval Abstract Domain

For the presentation of this paper, in Section 4.1, we have chosen to abstract integer values to the constant integer abstract domain. Of course this affects the abstract semantics of those string operations that involve them, namely `substring`, `charAt`, `repeat` and `length`, while the other methods only use strings or booleans. Nevertheless, μFASA abstracts integer values to the more precise interval abstract domain [9], i.e., to the set Intervals.

$$\mathsf{Intervals} \overset{\text{def}}{=} \{\ [a, b] \mid a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}, a\ \leq b\ \} \cup \{\bot\}$$

The choice of presenting the automata-based abstract semantics with constant integers, rather than intervals, was driven by the will to not burden the definition of the abstract semantics of the string operations involving integers. Let us consider the interval-based `substring` abstract semantics. Since intervals can be unbounded (e.g., $[5, +\infty]$), more than 20 different cases have been identified in its abstract semantics [8]. Given $\mathtt{substr}(\mathtt{A}, [a, b], [c, d])$, for some $\mathtt{A} \in \mathrm{DFA}_{/\equiv}$, many of these cases include $b = +\infty$ and d definite value, b definite and $d = +\infty$, $b, d = +\infty$ and a, c definite values and $a \leq c$, only to cite few. Moreover, the interval-based abstract semantics does not add any further important technical detail to our contribution since the cases cited above, met with an interval-based analysis, were handled in an ad hoc manner and would have made this paper harder to follow. In particular, being the constant integer abstract domain strictly contained into the intervals one, restricting the presentation to constant integers permitted us to report only the meaningful cases (from a technical point of view), avoiding the others (related to intervals) handled in specific ways (and relevant for the implementation).

Nevertheless, μFASA implements intervals (which include constant integers) and, accordingly, the abstract semantics based on them of `substring`, `charAt`, `length` and `repeat`, as reported in [8]. The abstract semantics of the other string operations remain unaffected by the change. Just as an example, in the following we report the abstract semantics of `length` on intervals.

`length` Abstract Semantics with Intervals

The constant integer domain leads to a big loss in precision in the abstract semantics of `length`, reported in Section 5.3. The idea behind the algorithm capturing its abstract semantics is to check if any string recognized by the input automata have the same length $l \in \mathbb{N}$. If so, l is returned as result, otherwise $\top_{\mathsf{Const}}$ is returned. Clearly, this is a forced choice given by the fact that the constant integer abstract domain is only able to track a single integer value. In this sense, the abstract semantics of `length` can be improved, from a precision point of view, when we deal with intervals rather than constant integers. Algorithm 7 reports the abstract semantics of `length` using the former abstract domain. We compute the minimum and the maximum path reaching each final state in the automaton and then we abstract the set of lengths obtained so far into intervals. Problems arise when the automaton contains cycles. In that case, we return the undefined interval starting from the minimum path, to a final state, to $+\infty$.

Clearly, using the interval abstract domain produces more precise results for certain operations, but it complicates the abstract semantics of others.

6.4. Qualitative Evaluation of μFASA

In this section, we evaluate the precision of μFASA and, in turn, of the finite state automata abstract domain. In particular, we comment and discuss two string manipulation programs. The first is the one already introduced in Section 1, namely an obfuscated malware manipulating strings and transforming them into code by using `eval`, while the second is a benevolent function taken from a real-world string manipulation program. In both cases, we will show that important string information can be obtained by μFASA.

Algorithm 7: $\mathsf{LE}^\sharp : \textsc{Dfa}_{/\equiv} \to$ Intervals Algorithm

```
Input: $\mathtt{A} = (Q, \Sigma, \delta, q_0, F)$
Output: $\mathsf{LE}^\sharp(\mathtt{A}) \in$ Intervals
P_len ← 0; p_len ← ∞
if hasCycle(A) then
    foreach $q_f \in F$ do
        p ← minPath(A, $q_0$, $q_f$);
        if len(p) < p_len then p_len ← len(p) ;
    end
    return [p_len, +∞];
else
    foreach $q_f \in F$ do
        p ← minPath(A, $q_0$, $q_f$);
        P ← maxPath(A, $q_0$, $q_f$);
        if len(p) < p_len then p_len ← len(p) ;
        if len(P) > P_len then P_len ← len(P) ;
    end
    return [p_len, P_len];
end
```

6.4.1. Obfuscated Malware

Consider the fragment reported in Figure 1 in the introduction. By analyzing it with μFasa, we obtain that the abstract value of `d`, at the `eval` call, is the automaton $\mathtt{A}_d$ in Figure 16. The cycles are caused by the widening application in `while` computations.

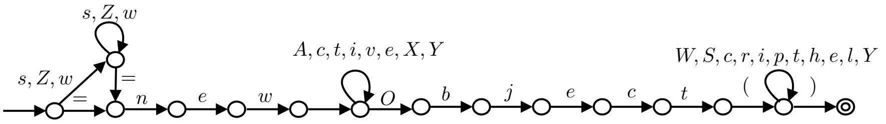


Figure 16. $\mathtt{A}_d$ abstract value of `d` before `eval` call of the program in Figure 1.

From this automaton we are able to retrieve some important and non-trivial information. For example, we are able to answer the following question: May $\mathtt{A}_d$ contain a string corresponding to an assignment to an ActiveXObject? We can answer by checking the predicate $\mathtt{A}_d \sqcap \mathsf{Min}(\mathbf{Id} \cdot \{\textit{new ActiveXObject}(\} \cdot \Sigma^* \cdot \{)\}) \neq \varnothing$, controlling 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 also answer that by checking if $\mathtt{A}_d \sqcap \mathsf{Min}(\{\textit{eval}\}) \neq \mathsf{Min}(\varnothing)$. In this case it returns false and enforces the idea that any explicit call to `eval` cannot occur.

This analysis may lose precision during fix-point computations, causing the cycles in the automaton in Figure 16, 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, narrowing or widening with thresholds, that can surely be implemented in the future development of μFasa.

6.4.2. String Manipulation Program

In order to evaluate the precision of μFasa, we decided to create a benchmark of tests taken from real-world programs. We therefore selected string manipulating functions from popular modern frameworks (such as Mozilla useful methods, RXJava, Mockito) whose code can be easily found on GitHub. (The selected string manipulation functions are available at www.github.com/SPY-Lab/mufasa/src/test/resources and it is possible here to go back to where they were selected.) Among this set of methods, we will focus our attention to the precision of the function `fixStations` reported in Figure 17, taken from [38]. The function takes as input an object `stations` containing information about train stations (each item contains the three-letter station code, followed by some machine-readable data, followed by a semicolon, followed by the human-readable station name) and extracts the station code (in capital letters) and the station name. For instance, given the input `stations ={st1:"MANay781;Manchester", st2:"gNfbx420;Greenfield"}`, the function returns the object`{st1: "MAN: Manchester", st2: "GNF: Greenfield"}`.

Thus, given an object containing strings following the station information pattern previously described, the function `fixStations` returns another object containing strings following the pattern of three capital letters concatenated with a colon concatenated with a string. The goal of our analyzer is to exactly preserve this information on the variable `result`. Let us consider a statically unknown value of `stations`, namely where `stations` $= \{\texttt{st1:}\sigma_1, \ldots, \texttt{stn:}\sigma_n\}$, $n \in \mathbb{N}$ and σ_i follows the station information pattern, for each $i \in [1, n]$. While other static analyzers, such as TAJS, which has a finite height string abstract domain, lose any information about the returned string, μFASA is able to infer, for the variable `result`, the object $\{\texttt{st1:p}_1, \ldots, \texttt{st}_n\texttt{:p}_n\}$, where each $\texttt{p}_i$ is a string abstract value, namely a finite state automaton, following the desired pattern

$$\sigma_1 \cdot \sigma_2 \cdot \sigma_3 \cdot : \cdot \ \sigma \quad \text{where } \sigma_i \text{ is capital}, i \in [1,3], \sigma \in \Sigma^*.$$

We are therefore able to preserve the string pattern that the function returns. As we have already highlighted, the result is obtained without implementing ad-hoc improvements regarding loop computations and we believe that even more precise results can be obtained integrating such techniques. We believe the integration of these analyses will drastically decrease false positives of the proposed string analysis (will address this topic in future works section).

```
function fixStations(stations) {
 result = {};
 for (st in stations) {
  input = stations[st];
  code = input.substring(0,3).toUpperCase();
  semiC = input.indexOf(";");
  name = input.substring(semiC + 1, input.length);
  result[st] = code + ": " + name;
 }

 return result;
}
```

Figure 17. Useful string manipulation method taken from [38].

7. Discussion and Related Work

In this paper we have proposed an abstract semantics for a toy imperative language μJS, 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 the 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.

7.1. Analysis vs. Verification

Even if several solutions, also involving finite state machines, have been proposed for string solving and verification [21,39,40], it should be noted that our approach is placed instead in the context of string static analysis. Over 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 [41], confirming this belief. Therefore, 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 as those cited above.

7.2. Main Related Works

The issue of analyzing strings is a widely studied problem and it has been tackled in literature from different points of view. Before discussing the most related works, we can observe what makes our approach original w.r.t. existing literature: (1) We provide a modular parametric abstract domain on the abstractions of the different primitive types, this allows us to obtain both a tunable semantics precision and to handle dynamic typing for operations having both integer and string parameters, such as `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 the 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 [21], 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 [20], the authors propose an abstract interpretation-based string analyzer approximating strings into a subset of regular languages, called regular strings and define the abstract semantics of four string operations of interest equipped 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. In [19], the authors propose a scalable static analysis for jQuery that relies on a novel abstract domain of regular expressions. The abstract domain in [19] contains the finite state automata one but pursues a different task and does not provide semantics for string operations. Surely it may be interesting to integrate our library for string manipulation operators into SAFE. Finally, [42] 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 also found a critical application in model checking both for enforcing string constraints and for modeling infinite transition systems [43]. For example, the authors of [44] define a sound decision procedure for a regular language-based logic for verification of string properties. The authors of [45] 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 [46–48]. 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 security issues related to the language [3–5,37]. TAJS [3] 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 [49], decreasing the number of false positives when objects are accessed. Upon TAJS, a sound way to statically analyze a large range of non-trivial `eval` patterns has been defined in [50]. In [37], it is defined 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 [51]. The authors have implemented LSA into SAFE [5], a JavaScript web applications static analyzer. As future work, it may be intriguing to combine LSA with our abstract semantics for decreasing the occurrences of false positives introduced by the widening operator during fix-point computations.

7.3. Future Ideas

In this paper we have proposed static string program analysis for a set of relevant JavaScript string manipulation operations, whose semantics is inspired by the official ECMAScript specification [10]. The first goal is to involve 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 can 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, which may lead to a big loss in precision. A narrowing will be studied and integrated in our static analyzer in order to retrieve some of the precision lost when the widening is applied.

We conclude by observing, as already highlighted in [7], the important application of finite state automata for string-to-code primitives analysis. Consider, for instance, in JavaScript programs, the `eval` function, transforming strings into code. Our semantics is sound and precise enough to answer some non-trivial properties of interest. Indeed, in [7], the finite state automata domain and the corresponding abstract semantics for strings turned out to be the basis for a sound and precise enough analysis of `eval`.

Author Contributions: V.A.: writing, conceptualization, formal analysis, original draft preparation, visualization, implementation; I.M.: writing, conceptualization, formal analysis, original draft preparation, visualization, supervision, methodology; S.X.: visualization, formal analysis, implementation. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the University of Verona under the 2017 grant project "Analyzing secuRity in the modErn Software" (ARES).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Selected Proofs

In this appendix we report all the long proofs of results presented in the paper. The proofs are listed in order of appearance.

Proof of Theorem 2. The collecting semantics of `substring` is defined lifting the concrete semantics defined in Section 3 as follows, where $S \in \wp(\Sigma^*)$ and $I, J \in \wp(\mathbb{Z})$.

$$\textsc{Ss}(S, I, J) = \{\ \textsc{Ss}(\sigma, i, j) \mid \sigma \in S, i \in I, j \in J\ \}$$

In order to prove soundness and completeness of $\textsf{SS}^\sharp$, we need to prove that $\forall \mathtt{A} \in \textsc{Dfa}_{/\equiv}, \forall i, j \in \textsf{Const}$

$$\textsc{Ss}(\mathscr{L}(\mathtt{A}), \gamma(i), \gamma(j)) = \mathscr{L}(\textsf{SS}^\sharp(\mathtt{A}, i, j))$$

We split the proof in the following cases. Since in `substring` semantics any negative value is treated as zero, in the proof, we suppose w.l.o.g. that when a negative value arises it is treated as zero.

- $\gamma(i) = \{l\}, \gamma(j) = \{k\}, l$ and $k \in \mathbb{Z}$: let us suppose, w.l.o.g., that $l < k$ (otherwise the indexes are swapped).

$$\begin{aligned}
&\textsc{Ss}(\mathscr{L}(\mathtt{A}), \{l\}, \{k\}) = \\
&= \{\ \sigma_l \ldots \sigma_k \mid \sigma \in \mathscr{L}(\mathtt{A}), k < |\sigma|\ \} \cup \{\ \sigma_i \ldots \sigma_n \mid \sigma \in \mathscr{L}(\mathtt{A}), k \geq n = |\sigma|\ \} \\
&= \{\ y \mid \exists z \in \Sigma^*.\, yz \in \textsc{Su}(\mathscr{L}(\mathtt{A}), l), z \in \textsc{Su}(\mathscr{L}(\mathtt{A}), k), |y| = k - l, k < |\sigma|\ \} \\
&\cup \{\ y \mid y \in \textsc{Su}(\mathscr{L}(\mathtt{A}), l), y \in \Sigma^{\leq k-l}\ \} \\
&= (\textsc{Rq}(\textsc{Su}(\mathscr{L}(\mathtt{A}), l), \textsc{Su}(\mathscr{L}(\mathtt{A}), k)) \cap \Sigma^{k-l}) \cup \textsc{Su}(\mathscr{L}(\mathtt{A}), l) \cap \Sigma^{\leq k-l} \\
&= \mathscr{L}((\mathsf{RQ}(\mathsf{SU}(\mathtt{A}, i), \mathsf{SU}(\mathtt{A}, j)) \sqcap \mathsf{Min}(\Sigma^{j-i})) \sqcup (\mathsf{SU}(\mathtt{A}, i) \sqcap \mathsf{Min}(\Sigma^{<j-i}))) \\
&= \mathscr{L}(\mathsf{SS}(\mathtt{A}, i, j))
\end{aligned}$$

- $\gamma(i) = \mathbb{Z}, \gamma(j) = \{k\}$, with $k \in \mathbb{Z}$

$$\begin{aligned}
&\textsc{Ss}(\mathscr{L}(\mathtt{A}), \mathbb{Z}, \{k\}) = \{\ \textsc{Ss}(\sigma, l, k) \mid \sigma \in \mathscr{L}(\mathtt{A}), l \in \mathbb{Z}\ \} \\
&= \{\ \textsc{Ss}(\sigma, l, k) \mid \sigma \in \mathscr{L}(\mathtt{A}), 0 \leq l < k\ \} \cup \{\ \textsc{Ss}(\sigma, k, l) \mid \sigma \in \mathscr{L}(\mathtt{A}), l \geq k \wedge l < |\sigma|\ \} \\
&= \bigcup_{a \in [0,k]} \textsc{Ss}(\mathscr{L}(\mathtt{A}), a, k) \cup \textsc{Fa}(\textsc{Su}(\mathscr{L}(\mathtt{A}), l)) \\
&= \mathscr{L}(\bigsqcup_{a \in [0,k]} \mathsf{SS}^\sharp(\mathtt{A}, a, k) \sqcup_{\text{DFA}} \mathsf{FA}(\mathsf{SU}(\mathtt{A}, l))) \\
&= \mathscr{L}(\bigsqcup_{a \in [0,k]} \mathsf{SS}(\mathtt{A}, a, k) \sqcup_{\text{DFA}} \mathsf{SS}^{\leftrightarrow}(\mathtt{A}, l)) \\
&= \mathscr{L}(\mathsf{SS}^\sharp(\mathtt{A}, i, j))
\end{aligned}$$

- $\gamma(i) = l \in \mathbb{Z}, \gamma(j) = \mathbb{Z}$:

$$\begin{aligned}
&\textsc{Ss}(\mathscr{L}(\mathtt{A}), l, \mathbb{Z}) = \{\ \textsc{Ss}(\sigma, l, k) \mid \sigma \in \mathscr{L}(\mathtt{A}), k \in \mathbb{Z}\ \} \\
&= \{\ \textsc{Ss}(\sigma, l, k) \mid \sigma \in \mathscr{L}(\mathtt{A}), k \geq l \wedge k \leq |\sigma|\ \} \cup \{\ \textsc{Ss}(\sigma, k, l) \mid \sigma \in \mathscr{L}(\mathtt{A}), 0 \leq k < l\ \} \\
&= \bigcup_{a \in [0,l]} \textsc{Ss}(\mathscr{L}(\mathtt{A}), a, l) \cup \textsc{Fa}(\textsc{Su}(\mathscr{L}(\mathtt{A}), l)) \\
&= \mathscr{L}(\bigsqcup_{a \in [0,l]} \mathsf{SS}^\sharp(\mathtt{A}, a, l) \sqcup_{\text{DFA}} \mathsf{FA}(\mathsf{SU}(\mathtt{A}, l))) \\
&= \mathscr{L}(\bigsqcup_{a \in [0,l]} \mathsf{SS}(\mathtt{A}, a, k) \sqcup_{\text{DFA}} \mathsf{SS}^{\leftrightarrow}(\mathtt{A}, l)) \\
&= \mathscr{L}(\mathsf{SS}^\sharp(\mathtt{A}, i, j))
\end{aligned}$$

- $\gamma(i) = \gamma(j) = \mathbb{Z}$:

$$\begin{aligned}
&\textsc{Ss}(\mathscr{L}(\mathtt{A}), \mathbb{Z}, \mathbb{Z}) = \{\ \textsc{Ss}(\sigma, l, k) \mid \sigma \in \mathscr{L}(\mathtt{A}), l, k \in \mathbb{Z}\ \} \\
&= \{\ \textsc{Ss}(\sigma, l, k) \mid \sigma \in \mathscr{L}(\mathtt{A}), l, k \geq 0, l, k < |\sigma|\ \} \\
&= \textsc{Fa}(\mathscr{L}(\mathtt{A})) = \mathsf{FA}(\mathtt{A}) \\
&= \mathscr{L}(\mathsf{SS}^\sharp(\mathtt{A}, i, j))
\end{aligned}$$

□

Proof of Theorem 3. The collecting semantics of `charAt` is defined lifting the concrete semantics defined in Section 3 as follows, where $S \in \wp(\Sigma^*)$ and $I, \in \wp(\mathbb{Z})$.

$$\textsc{Ca}(S, I) = \{\ \textsc{Ca}(\sigma, i) \mid \sigma \in S, i \in I\ \}$$

In order to prove soundness and completeness of $\mathsf{CC}^\sharp$, we need to prove that $\forall \mathtt{A} \in \mathrm{DFA}_{/\equiv}, \forall i \in \mathsf{Const}$

$$\mathrm{CA}(\mathscr{L}(\mathtt{A}), \gamma(i)) = \mathscr{L}(\mathsf{CA}^\sharp(\mathtt{A}, i))$$

We split the proof in the following two cases.

- Let us suppose that $i \neq \top_{\mathsf{Const}}$, hence $\gamma(i) = \{n\}$, where $n \in \mathbb{Z}$.

$$\begin{aligned}
\mathrm{CA}(\mathscr{L}(\mathtt{A}), \{n\}) &= \{\ \mathrm{CA}(\sigma, n) \mid \sigma \in \mathscr{L}(\mathtt{A})\ \}\\
&= \{\ \sigma_n \mid \sigma \in \mathscr{L}(\mathtt{A}), 0 \leq n < |\sigma|\ \}\\
&\cup \{\ \epsilon \mid \exists \sigma \in \mathscr{L}(\mathtt{A}).\, n \geq |\sigma| \vee n < 0\ \}\\
&= \{\ \mathsf{Ss}(\sigma, n, n+1) \mid \sigma \in \mathscr{L}(\mathtt{A}), 0 \leq n < |\sigma|\ \}\\
&\cup \{\ \mathsf{Ss}(\sigma, n, n+1) \mid \exists \sigma \in \mathscr{L}(\mathtt{A}).\, n \geq |\sigma| \vee n < 0\ \}\\
&= \{\ \mathsf{Ss}(\sigma, \{n\}, \{n+1\}) \mid \sigma \in \mathscr{L}(\mathtt{A})\ \} = \mathsf{Ss}(\mathscr{L}(\mathtt{A}), n, n+1)\\
&= \mathscr{L}(\mathsf{SS}^\sharp(\mathtt{A}, i, i+1))\\
&= \mathscr{L}(\mathsf{CA}^\sharp(\mathtt{A}, i))
\end{aligned}$$

- Let us suppose that $i = \top_{\mathsf{Const}}$, hence $\gamma(i) = \mathbb{Z}$. It is worth noting that the function `chars` we used in the abstract semantics of `charAt` is complete. Let $\mathrm{CHARS} : \wp(\Sigma^*) \to \wp(\Sigma)$ be the function that given a set of strings returns the set of characters inside any string of the input string set. It holds that $\mathrm{CHARS}(\mathscr{L}(\mathtt{A})) = \mathtt{chars}(\mathtt{A})$.

$$\begin{aligned}
\mathrm{CA}(\mathscr{L}(\mathtt{A}), \gamma(i)) &= \{\ \mathrm{CA}(\sigma, n) \mid \sigma \in \mathscr{L}(\mathtt{A}), n \in [0, |\sigma| - 1]\ \} \cup \{\epsilon\} =\\
&= \{\ \sigma_n \mid \sigma \in \mathscr{L}(\mathtt{A}), n \in [0, |\sigma| - 1]\ \} \cup \{\epsilon\} =\\
&= \mathrm{CHARS}(\mathscr{L}(\mathtt{A})) \cup \{\epsilon\}\\
&= \mathscr{L}(\mathsf{Min}(\mathtt{chars}(\mathtt{A})) \sqcup \mathsf{Min}(\{\epsilon\}))\\
&= \mathscr{L}(\mathsf{CA}^\sharp(\mathtt{A}, i))
\end{aligned}$$

□

Proof Of Theorem 4. The collecting semantics of `length` is defined lifting the concrete semantics defined in Section 3 as follows, where $S \in \wp(\Sigma^*)$.

$$\mathrm{LE}(S) = \{\ |\sigma| \mid \sigma \in S\ \}$$

In order to prove soundness of $\mathsf{LE}^\sharp$, we need to prove that $\forall \mathtt{A} \in \mathrm{DFA}_{/\equiv}$

$$\mathrm{LE}(\mathscr{L}(\mathtt{A})) \subseteq \gamma(\mathsf{LE}^\sharp(\mathtt{A}))$$

We split the proof in the following cases.

- $\mathrm{LE}(\mathscr{L}(\mathtt{A}))) = I \in \wp(\mathbb{Z})$, s.t. $|I| = 1$:

$$\begin{aligned}
|\mathrm{LE}(\mathscr{L}(\mathtt{A}))| = 1 &\Leftrightarrow \mathrm{LE}(\mathscr{L}(\mathtt{A})) = \{n\} \text{ for some } n \in \mathbb{N}\\
&\Leftrightarrow \forall \sigma \in \mathscr{L}(\mathtt{A}).\, |\sigma| = n\\
&\Leftrightarrow \forall \mathsf{p} \in \mathsf{Paths}(\mathtt{A}), |p| = n
\end{aligned}$$

  This condition checks whether the size of any path of A is n. This check is performed by Algorithm 1 at lines 5–8.

- $\textsc{Le}(\mathscr{L}(\mathtt{A})) = I \in \wp(\mathbb{Z})$, s.t. $|I| > 1$: this means that

  $$|\textsc{Le}(\mathscr{L}(\mathtt{A}))| > 1 \Leftrightarrow \exists \sigma, \sigma' \in \mathscr{L}(\mathtt{A}).\ |\sigma| \neq |\sigma'|$$

  If $\mathtt{A}$ is cyclic, then the condition at line 1 is successful and $\top_{\mathsf{Const}}$ is returned. Let us suppose that $\mathtt{A}$ is not cyclic.

  $$\begin{aligned}|\textsc{Le}(\mathscr{L}(\mathtt{A}))| > 1 &\Leftrightarrow \exists \sigma, \sigma' \in \mathscr{L}(\mathtt{A}).\ |\sigma| \neq |\sigma'| \\ &\Leftrightarrow \exists \mathsf{p}, \mathsf{p}' \in \mathsf{Paths}(\mathtt{A}).\ |\mathsf{p}| \neq |\mathsf{p}'|\end{aligned}$$

  This condition is checked by lines 5–8 of Algorithm 1.

□

Proof of Theorem 6. The collecting semantics of `startsWith` is defined lifting the concrete semantics defined in Section 3 as follows, where $S, S' \in \wp(\Sigma^*)$.

$$\textsc{Sw}(S, S') = \{\ \textsc{Sw}(\sigma, \sigma') \mid \sigma, \sigma' \in S\ \}$$

In order to prove soundness of $\mathsf{SW}^\sharp$, we need to prove that $\forall \mathtt{A}, \mathtt{A}' \in \textsc{Dfa}_{/\equiv}$

$$\textsc{Sw}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) \subseteq \mathsf{SW}^\sharp(\mathtt{A}, \mathtt{A}')$$

We split the proof in the following cases.

- Let us suppose that $\textsc{Sw}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \{\mathtt{false}\}$.

  $$\begin{aligned}\textsc{Sw}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \{\mathtt{false}\} &\Leftrightarrow \forall \sigma \in \mathscr{L}(\mathtt{A}).\forall \sigma' \in \mathscr{L}(\mathtt{A}').\nexists \phi \in \Sigma^*.\sigma' \cdot \phi = \sigma \\ &\Leftrightarrow \textsc{Pr}(\mathscr{L}(\mathtt{A})) \cap \mathscr{L}(\mathtt{A}') = \varnothing \\ &\Leftrightarrow \mathsf{PR}(\mathtt{A}) \sqcap_{\textsc{dfa}} \mathtt{A}' = \mathsf{Min}(\varnothing) \text{ (lines 4-6 of Algorithm 2)}\end{aligned}$$

- Let us suppose that $\textsc{Sw}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \{\mathtt{true}\}$. We split the proof in the following cases:

  - if $\mathtt{A}' = \mathsf{Min}(\{\epsilon\})$: Algorithm 2 verifies the condition ($\mathtt{A}' == \mathsf{Min}(\{\epsilon\})$) at lines 1–3 and returns $\{\mathtt{true}\}$.
  - if $\mathtt{A}$ or $\mathtt{A}'$ are cyclic: Algorithm 2 verifies the condition ($\mathsf{hasCycle}(\mathtt{A}) \vee \mathsf{hasCycle}(\mathtt{A}')$) at lines 7–9 and returns $\{\mathtt{true}, \mathtt{false}\}$.
  - if $\mathtt{A}'$ is not a single-path automaton: in this case, we check if $\mathtt{A}'$ is not a single path automaton at line 10 of Algorithm 2 and, if so, $\{\mathtt{true}, \mathtt{false}\}$ is returned at line 17.
  - if $\mathtt{A}'$ is a single path automaton: let us denote by $\textsc{maxString}(\mathscr{L}(\mathtt{A}'))$ the longest string recognized by $\mathscr{L}(\mathtt{A}')$. As we already highlighted, if $\mathtt{A}'$ is single path, the longest string is unique. Clearly, we have that $\textsc{maxString}(\mathscr{L}(\mathtt{A}')) = \mathsf{maxString}(\mathtt{A}')$. Let us denote $\textsc{maxString}(\mathscr{L}(\mathtt{A}'))$ by σ^m.

    $$\begin{aligned}\textsc{Sw}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) &= \{\mathtt{true}\} \\ &\Leftrightarrow \forall \sigma \in \mathscr{L}(\mathtt{A}), \forall \sigma' \in \mathscr{L}(\mathtt{A}').\exists \phi \in \Sigma^*.\ \sigma' \cdot \phi = \sigma \\ &\Rightarrow \forall \sigma \in \mathscr{L}(\mathtt{A})\ \exists \phi \in \Sigma^*.\ \sigma = \sigma^m \cdot \phi \\ &\Rightarrow \textsc{Ss}(\mathscr{L}(\mathtt{A}), 0, |\sigma^m|) == \sigma^m \\ &\Leftrightarrow \mathsf{SS}^\sharp(\mathtt{A}, 0, \mathsf{LE}^\sharp(\mathsf{maxString}(\mathtt{A}'))) == \mathsf{maxString}(\mathtt{A}') \\ &\text{(lines 10-15 of Algorithm 2)}\end{aligned}$$

- Let us suppose that $\textsc{Sw}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \{\mathtt{true}, \mathtt{false}\}$. We split the proof in the following cases:

- $\mathtt{A}$ or $\mathtt{A}'$ are cyclic: Algorithm 2 verifies the condition $(\mathsf{hasCycle}(\mathtt{A}) \vee \mathsf{hasCycle}(\mathtt{A}'))$ at lines 7–9 and returns $\{\mathtt{true}, \mathtt{false}\}$.
- if $\mathtt{A}'$ is not single path automaton: the check at line 10 of Algorithm 2 fails and $\{\mathtt{true}, \mathtt{false}\}$ is returned at line 17.
- $\mathtt{A}'$ is single-path automaton: as before, if $\mathtt{A}'$ is single path, the longest string is unique. Let us denote $\textsc{maxString}(\mathscr{L}(\mathtt{A}'))$ by σ^m.

$$\begin{aligned} \textsc{Sw}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \{\mathtt{true}\,\mathtt{false}\} & \\ & \Rightarrow \exists \sigma \in \mathscr{L}(\mathtt{A}), \exists \sigma' \in \mathscr{L}(\mathtt{A}') \forall \phi \in \Sigma^*.\, \sigma' \cdot \phi \neq \sigma \\ & \Rightarrow \forall \sigma \in \mathscr{L}(\mathtt{A}) \exists \phi \in \Sigma^*.\, \sigma^m \cdot \phi \neq \sigma \\ & \Rightarrow \textsc{Ss}(\mathscr{L}(\mathtt{A}), 0, |\sigma^m|) \neq \sigma^m \\ & \Leftrightarrow \mathsf{SS}^\sharp(\mathtt{A}, 0, \mathsf{LE}^\sharp(\mathsf{maxString}(\mathtt{A}'))) \neq \mathsf{maxString}(\mathtt{A}') \\ & \text{(lines 10-15 of Algorithm 2)} \end{aligned}$$

The condition is verified at lines 13 of Algorithm 2, it fails, hence $\{\mathtt{true}, \mathtt{false}\}$ at line 17 is returned.

□

Proof of Theorem 7. The soundness and completeness of $\mathsf{LC}^\sharp$ follows from the fact that any upper-case transition found in $\mathtt{A}$ is replaced with the same transition that reads the corresponding lower-case symbol, without changing neither the orientation of the transitions or the automaton states. □

Proof of Theorem 8. The collecting semantics of `includes` is defined lifting the concrete semantics defined in Section 3 as follows, where $S, S' \in \wp(\Sigma^*)$.

$$\textsc{In}(S, S') = \{\ \textsc{In}(\sigma, \sigma') \mid \sigma, \sigma' \in S\ \}$$

In order to prove soundness of $\mathsf{IN}^\sharp$, we need to prove that $\forall \mathtt{A}, \mathtt{A}' \in \textsc{Dfa}_{/\equiv}$

$$\textsc{In}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) \subseteq \mathsf{IN}^\sharp(\mathtt{A}, \mathtt{A}')$$

We split the proof of the following cases.

- Let us suppose that $\textsc{In}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \{\mathtt{false}\}$.

$$\begin{aligned} \textsc{In}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \{\mathtt{false}\} & \Leftrightarrow \forall \sigma \in \mathscr{L}(\mathtt{A}').\, \sigma \notin \textsc{Fa}(\mathtt{A}) \\ & \Leftrightarrow \mathscr{L}(\mathtt{A}') \cap \textsc{Fa}(\mathscr{L}(\mathtt{A})) = \varnothing \\ & \Leftrightarrow \mathtt{A}' \sqcap_{\textsc{Dfa}} \mathsf{FA}(\mathtt{A}) = \mathsf{Min}(\varnothing) \\ & \text{(lines 4–6 of Algorithm 4)} \end{aligned}$$

- Let us suppose that $\textsc{In}(\mathscr{L}(\mathtt{A}), \mathscr{L}(\mathtt{A}')) = \{\mathtt{true}\}$. Thus, consider the following cases:
  - $\mathtt{A}' = \mathsf{Min}(\{\epsilon\})$: Algorithm 4 verifies the condition $(\mathtt{A}' == \mathsf{Min}(\{\epsilon\}))$ at lines 1–3 and returns $\{\mathtt{true}\}$.
  - $\mathtt{A}$ or $\mathtt{A}'$ are cyclic: Algorithm 2 verifies the condition $(\mathsf{hasCycle}(\mathtt{A}) \vee \mathsf{hasCycle}(\mathtt{A}'))$ at lines 7–9 and returns $\{\mathtt{true}, \mathtt{false}\}$.
  - $\mathtt{A}' \neq \mathsf{Min}(\{\epsilon\})$ and $\mathtt{A}, \mathtt{A}'$ are not cyclic:

$$\begin{aligned}\textsc{In}(\mathscr{L}(\mathtt{A}),\mathscr{L}(\mathtt{A}')) = \{\mathtt{true}\} \Leftrightarrow{} & \forall\sigma' \in \mathscr{L}(\mathtt{A}').\forall\sigma \in \mathscr{L}(\mathtt{A})\\ & \exists\phi,\psi \in \Sigma^*.\, \phi\cdot\sigma'\cdot\psi = \sigma\\ & \Rightarrow \forall\sigma \in \mathscr{L}(\mathtt{A}).\, \textsc{Fa}(\{\sigma\}) \cap \mathscr{L}(\mathtt{A}') = \mathscr{L}(\mathtt{A}')\\ & \Rightarrow \forall \mathsf{p} \in \mathsf{Paths}(\mathtt{A}).\, \mathsf{FA}(\mathsf{Min}(\mathsf{p})) \sqcap_{\textsc{Dfa}} \mathtt{A}' = \mathtt{A}'\end{aligned}$$

This condition is verified in lines 11–15 of Algorithm 4 and in this case the algorithm returns $\{\mathtt{true}\}$.

- Let us suppose that $\textsc{In}(\mathscr{L}(\mathtt{A}),\mathscr{L}(\mathtt{A}')) = \{\mathtt{true},\mathtt{false}\}$. Thus, consider the following cases:
  - $\mathtt{A}$ or $\mathtt{A}'$ are cyclic: Algorithm 4 verifies the condition ($\mathsf{hasCycle}(\mathtt{A}) \vee \mathsf{hasCycle}(\mathtt{A}')$) at lines 7–9 and returns $\{\mathtt{true},\mathtt{false}\}$.
  - $\mathtt{A}' \neq \mathsf{Min}(\{\epsilon\})$ and $\mathtt{A},\mathtt{A}'$ are not cyclic:

$$\begin{aligned}\textsc{In}(\mathscr{L}(\mathtt{A}),\mathscr{L}(\mathtt{A}')) = \{\mathtt{true},\mathtt{false}\} \Rightarrow{} & \exists\sigma' \in \mathscr{L}(\mathtt{A}')\\ & \exists\sigma \in \mathscr{L}(\mathtt{A}).\nexists\phi,\psi \in \Sigma^*.\, \phi\cdot\sigma'\cdot\psi = \sigma\\ & \Rightarrow \exists\sigma \in \mathscr{L}(\mathtt{A}).\, \textsc{Fa}(\{\sigma\}) \cap \mathscr{L}(\mathtt{A}') \neq \mathscr{L}(\mathtt{A}')\\ & \Rightarrow \exists \mathsf{p} \in \mathsf{Paths}(\mathtt{A}).\, \mathsf{FA}(\mathsf{Min}(\mathsf{p})) \sqcap_{\textsc{Dfa}} \mathtt{A}' \neq \mathtt{A}'\end{aligned}$$

This condition is verified in lines 11–15 of Algorithm 4 and in this case the algorithm returns $\{\mathtt{true},\mathtt{false}\}$.

□

Proof of Theorem 9. The collecting semantics of `repeat` is defined lifting the concrete semantics defined in Section 3 as follows, where $S \in \wp(Sigma^*)$ and $I \in \wp(\mathbb{Z})$.

$$\textsc{Rt}(S,I) = \{\ \sigma^n \mid \sigma \in S, n \in I\ \}$$

In order to prove soundness of $\mathsf{RT}^\sharp$, we need to prove that $\forall\mathtt{A} \in \textsc{Dfa}_{/\equiv}, \forall i \in \mathsf{Const}$

$$\textsc{Rt}(\mathscr{L}(\mathtt{A}),\gamma(i)) \subseteq \mathscr{L}(\mathsf{RT}^\sharp(\mathtt{A},i))$$

We split the proof in the following two cases.

- Let us suppose that $i \neq \top_{\mathsf{Const}}$, hence $\gamma(i) = n$, where $n \in \mathbb{Z}$. We split the proof in the following cases:
  - $i = 0$: $\textsc{Rt}(\mathscr{L}(\mathtt{A}),0) = \{\epsilon\}$ and Algorithm 5 checks this condition and returns $\mathsf{Min}(\{\epsilon\})$ at lines 1–3.
  - $i \neq 0$:
    - ∗ if $\mathtt{A}$ is s.t. $\mathscr{L}(\mathtt{A}) = \{\epsilon\}$: since $\textsc{Rt}(\{\epsilon\},i) = \{\epsilon\}$, Algorithm 5 checks this condition and returns $\mathsf{Min}(\{\epsilon\})$ at lines 1–3.
    - ∗ if $\mathtt{A}$ is cyclic: $\textsc{Rt}(\mathscr{L}(\mathtt{A}),i) \subseteq \mathscr{L}(\mathsf{Kleene}(\mathtt{A}))$ and Algorithm5 checks this condition and returns $\mathsf{Kleene}(\mathtt{A})$ at lines 4–6.

- $\texttt{A}$ is not cyclic:

$$\begin{aligned}\textsc{Rt}(\mathscr{L}(\texttt{A}), i) &= \{\ \sigma^i \mid \sigma \in \mathscr{L}(\texttt{A})\ \} \\ &= \{\ \overbrace{\sigma \cdot \sigma \cdot \dots \cdot \sigma}^{i-times} \mid \sigma \in \mathscr{L}(\texttt{A})\ \} \\ &= \mathscr{L}(\{\ \overbrace{\mathsf{Min}(\mathsf{p}) \cdot \mathsf{Min}(\mathsf{p}) \cdot \dots \cdot \mathsf{Min}(\mathsf{p})}^{i-times} \mid \mathsf{p} \in \mathsf{Paths}(\texttt{A})\ \})\end{aligned}$$

In this case, Algorithm5 returns the above automaton at lines 8–15.

- Let us suppose that $i = \top_{\mathsf{Const}}$, hence we have that $\gamma(i) = \mathbb{Z}$.

$$\textsc{Rt}(\mathscr{L}(\texttt{A}), \gamma(i)) = \textsc{Rt}(\mathscr{L}(\texttt{A}), \mathbb{Z}) = \{\ \sigma^n \mid \sigma \in \mathscr{L}(\texttt{A}), n > 0\ \} \subseteq \mathscr{L}(\mathsf{Kleene}(\texttt{A}))$$

In this case, Algorithm 5 returns $\mathsf{Kleene}(\texttt{A})$, guaranteeing the soundness of $\text{RT}^\sharp$.

□

Proof of Theorem 10. At each iteration of Algorithm 6, we remove white-space transitions from the initial state q_0. The invariant of Algorithm 6 after line 9 is that the initial state has no white-space transitions. Before checking the while-loop condition (line 3), the automaton is minimized and determinized (lines 3) with the new set of transitions δ'. Hence, if the initial state q_0 had only white-state transitions, after the minimization at line 11, q_0 is not the initial state of $\texttt{R}$ anymore, and a new initial state will be computed at line 11. Hence, when the loop is repeated, the algorithm will search the other white-space transition from the new initial state. In this way, Algorithm 6 is able to remove consecutive white-space transitions from the original initial state q_0. □

References

1. Pradel, M.; Sen, K. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In Proceedings of the 29th European Conference on Object-Oriented Programming, ECOOP 2015, Prague, Czech Republic, 5–10 July 2015; Boyland, J.T., Ed.; LIPIcs; Schloss Dagstuhl- Leibniz-Zentrum für Informatik: Wadern, Germany, 2015; Volume 37, pp. 519–541. [CrossRef]
2. Xu, W.; Zhang, F.; Zhu, S. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In Proceedings of the 7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, 16–18 October 2012; IEEE Computer Society: Washington, DC, USA, 2012; pp. 9–16. [CrossRef]
3. Jensen, S.H.; Møller, A.; Thiemann, P. Type Analysis for JavaScript. In Proceedings of the 16th International Symposium on Static Analysis, SAS 2009, Los Angeles, CA, USA, 9–11 August 2009; Palsberg, J., Su, Z., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2009; Volume 5673, pp. 238–255._17. [CrossRef]
4. Kashyap, V.; Dewey, K.; Kuefner, E.A.; Wagner, J.; Gibbons, K.; Sarracino, J.; Wiedermann, B.; Hardekopf, B. JSAI: A static analysis platform for JavaScript. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, 16–22 November 2014; Cheung, S., Orso, A., Storey, M.D., Eds.; ACM: New York, NY, USA, 2014; pp. 121–132. [CrossRef]
5. Lee, H.; Won, S.; Jin, J.; Cho, J.; Ryu, S. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In Proceedings of the 19th International Workshop on Foundations of Object-Oriented Languages (FOOL'12), Tucson, AZ, USA, 19–26 October 2012.
6. Hauzar, D.; Kofron, J. Framework for Static Analysis of PHP Applications. In Proceedings of the 29th European Conference on Object-Oriented Programming, ECOOP 2015, Prague, Czech Republic, 5–10 July 2015; Boyland, J.T., Ed.; LIPIcs; Schloss Dagstuhl-Leibniz-Zentrum für Informatik: Wadern, Germany, 2015; Volume 37, pp. 689–711. [CrossRef]

7. Arceri, V.; Mastroeni, I. A sound abstract interpreter for dynamic code. In Proceedings of the SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, Brno, Czech Republic, 30 March–3 April 2020; Hung, C., Cerný, T., Shin, D., Bechini, A., Eds.; ACM: New York, NY, USA, 2020; pp. 1979–1988. [CrossRef]
8. Arceri, V.; Mastroeni, I. Static Program Analysis for String Manipulation Languages. In Proceedings Seventh International Workshop on Verification and Program Transformation, VPT@Programming 2019, Genova, Italy, 2 April 2019; Volume 299, pp. 19–33. [CrossRef]
9. Cousot, P.; Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Proceedings of the Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, CA, USA, 17–19 January 1977; Graham, R.M., Harrison, M.A., Sethi, R., Eds.; ACM: New York, NY, USA, 1977; pp. 238–252. [CrossRef]
10. ECMA. Standard ECMA-262 Language Specification, 9th ed. Available online: https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (accessed on 6 December 2018).
11. Hopcroft, J.E.; Ullman, J.D. *Introduction to Automata Theory, Languages and Computation*; Addison-Wesley: Reading, MA, USA, 1979.
12. Davis, M.D.; Sigal, R.; Weyuker, E.J. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*; Academic Press Professional, Inc.: Cambridge, MA, USA, 1994.
13. Cousot, P.; Cousot, R. Systematic Design of Program Analysis Frameworks. In Proceedings of the Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, TX, USA, 29–31 January 1979; Aho, A.V., Zilles, S.N., Rosen, B.K., Eds.; ACM Press: New York, NY, USA, 1979; pp. 269–282. [CrossRef]
14. Cousot, P.; Cousot, R. Abstract Interpretation Frameworks. *J. Log. Comput.* **1992**, *2*, 511–547. [CrossRef]
15. Giacobazzi, R.; Quintarelli, E. Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking. In Proceedings of the Static Analysis, 8th International Symposium, SAS 2001, Paris, France, 16–18 July 2001; Cousot, P., Ed.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2001; Volume 2126, pp. 356–373. [CrossRef]
16. Giacobazzi, R.; Mastroeni, I. Making abstract models complete. *Math. Struct. Comput. Sci.* **2016**, *26*, 658–701. [CrossRef]
17. Giacobazzi, R.; Mastroeni, I. Transforming Abstract Interpretations by Abstract Interpretation. In Proceedings of the Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, 16–18 July 2008; Alpuente, M., Vidal, G., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2008; Volume 5079, pp. 1–17. [CrossRef]
18. Arceri, V.; Maffeis, S. Abstract Domains for Type Juggling. *Electron. Notes Theor. Comput. Sci.* **2017**, *331*, 41–55. [CrossRef]
19. Park, C.; Im, H.; Ryu, S. Precise and scalable static analysis of jQuery using a regular expression domain. In Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, 1 November 2016; Ierusalimschy, R., Ed.; ACM: New York, NY, USA, 2016; pp. 25–36. [CrossRef]
20. Choi, T.; Lee, O.; Kim, H.; Doh, K. A Practical String Analyzer by the Widening Approach. In Proceedings of the 4th Asian Symposium on Programming Languages and Systems, APLAS 2006, Sydney, Australia, 8–10 November 2006; Kobayashi, N., Ed.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2006; Volume 4279, pp. 374–388. [CrossRef]
21. Yu, F.; Bultan, T.; Cova, M.; Ibarra, O.H. Symbolic String Verification: An Automata-Based Approach. In Proceedings of the 15th International SPIN Workshop on Model Checking Software, Los Angeles, CA, USA, 10–12 August 2008; Havelund, K., Majumdar, R., Palsberg, J., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2008; Volume 5156, pp. 306–324. [CrossRef]
22. Câmpeanu, C.; Paun, A.; Yu, S. An Efficient Algorithm for Constructing Minimal Cover Automata for Finite Languages. *Int. J. Found. Comput. Sci.* **2002**, *13*, 83–97. [CrossRef]
23. Domaratzki, M.; Shallit, J.O.; Yu, S. Minimal Covers of Formal Languages. In Proceedings of the 5th International Conference Developments in Language Theory, DLT 2001, Vienna, Austria, 16–21 July 2001; Revised Papers; Kuich, W., Rozenberg, G., Salomaa, A., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2001; Volume 2295, pp. 319–329. [CrossRef]
24. Mohri, M.; Nederhof, M. Regular Approximation of Context-Free Grammars through Transformation. In *Robustness in Language and Speech Technology*; Springer: Dordrecht, The Netherlands, 2001; pp. 153–163.

25. Cousot, P.; Halbwachs, N. Automatic Discovery of Linear Restraints Among Variables of a Program. In Proceedings of the Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, AZ, USA, 23–25 January 1978; Aho, A.V., Zilles, S.N., Szymanski, T.G., Eds.; ACM Press: New York, NY, USA, 1978; pp. 84–96. [CrossRef]
26. Costantini, G.; Ferrara, P.; Cortesi, A. A suite of abstract domains for static analysis of string values. *Softw. Pract. Exp.* **2015**, *45*, 245–287. [CrossRef]
27. Cousot, P.; Cousot, R. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming, PLILP'92, Leuven, Belgium, 26–28 August 1992; Bruynooghe, M., Wirsing, M., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 1992; Volume 631, pp. 269–295. [CrossRef]
28. D'Silva, V. Widening for Automata. Ph.D. Thesis, Institut Fur Informatick, UZH, Zurich, Switzerland, 2006.
29. Bartzis, C.; Bultan, T. Widening Arithmetic Automata. In Proceedings of the 16th International Conference on Computer Aided Verification, CAV 2004, Boston, MA, USA, 13–17 July 2004; Alur, R., Peled, D.A., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2004; Volume 3114, pp. 321–333. [CrossRef]
30. Cousot, P. Types as Abstract Interpretations. In Proceedings of the Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, 15–17 January 1997; Lee, P., Henglein, F., Jones, N.D., Eds.; ACM Press: New York, NY, USA, 1997; pp. 316–331. [CrossRef]
31. Reynolds, J.C. *Theories of Programming Languages*; Cambridge University Press: Cambridge, UK, 1998.
32. Giacobazzi, R.; Ranzato, F.; Scozzari, F. Making abstract interpretations complete. *J. ACM* **2000**, *47*, 361–416. [CrossRef]
33. Fromherz, A.; Ouadjaout, A.; Miné, A. Static Value Analysis of Python Programs by Abstract Interpretation. In Proceedings of the 10th International Symposium on NASA Formal Methods, NFM 2018, Newport News, VA, USA, 17–19 April 2018; Dutle, A., Muñoz, C.A., Narkawicz, A., Eds.; Lecture Notes in Computer Science; Springer: Berin, Germany, 2018; Volume 10811, pp. 185–202. [CrossRef]
34. Bordihn, H.; Holzer, M.; Kutrib, M. Determination of finite automata accepting subregular languages. *Theor. Comput. Sci.* **2009**, *410*, 3209–3222. [CrossRef]
35. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; MIT Press: Cambridge, MA, USA, 2009.
36. Holzer, M.; Jakobi, S. Brzozowski's Minimization Algorithm - More Robust than Expected-(Extended Abstract). In Proceedings of the 18th International Conference on Implementation and Application of Automata, CIAA 2013, Halifax, NS, Canada, 16–19 July 2013; Konstantinidis, S., Ed.; Springer: Berlin, Germany, 2013; Lecture Notes in Computer Science; Volume 7982, pp. 181–192. [CrossRef]
37. Park, C.; Ryu, S. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In Proceedings of the 29th European Conference on Object-Oriented Programming, ECOOP 2015, Prague, Czech Republic, 5–10 July 2015; LIPIcs; Boyland, J.T., Ed.; Schloss Dagstuhl-Leibniz-Zentrum für Informatik: Wadern, Germany, 2015; Volume 37, pp. 735–756. [CrossRef]
38. Mozilla. MDN Web Docs-Useful String Methods. Available online: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Useful_string_methods (accessed on 20 April 2020).
39. Abdulla, P.A.; Atig, M.F.; Chen, Y.; Holík, L.; Rezine, A.; Rümmer, P.; Stenman, J. Norn: An SMT Solver for String Constraints. In Proceedings of the Computer Aided Verification-27th International Conference, CAV 2015, San Francisco, CA, USA, 18–24 July 2015; Part I; Kroening, D., Pasareanu, C.S., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2015; Volume 9206, pp. 462–469. [CrossRef]
40. Liang, T.; Reynolds, A.; Tsiskaridze, N.; Tinelli, C.; Barrett, C.W.; Deters, M. An efficient SMT solver for string constraints. *Form. Methods Syst. Des.* **2016**, *48*, 206–234. [CrossRef]
41. Cousot, P.; Giacobazzi, R.; Ranzato, F. Program Analysis Is Harder Than Verification: A Computability Perspective. In Proceedings of the Computer Aided Verification-30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, 14–17 July 2018; Part II; Chockler, H., Weissenbacher, G., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2018; Volume 10982, pp. 75–95. [CrossRef]

42. Midtgaard, J.; Nielson, F.; Nielson, H.R. A Parametric Abstract Domain for Lattice-Valued Regular Expressions. In Proceedings of the Static Analysis-23rd International Symposium, SAS 2016, Edinburgh, UK, 8–10 September 2016; Rival, X., Ed.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2016; Volume 9837, pp. 338–360. [CrossRef]
43. Lin, A.W.; Barceló, P. String solving with word equations and transducers: Towards a logic for analysing mutation XSS. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016; Bodík, R., Majumdar, R., Eds.; ACM: New York, NY, USA, 2016; pp. 123–136. [CrossRef]
44. Abdulla, P.A.; Atig, M.F.; Chen, Y.; Holík, L.; Rezine, A.; Rümmer, P.; Stenman, J. String Constraints for Verification. In Proceedings of the Computer Aided Verification-26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, 18–22 July 2014; Biere, A., Bloem, R., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2014; Volume 8559, pp. 150–166. [CrossRef]
45. Bouajjani, A.; Habermehl, P.; Vojnar, T. Abstract Regular Model Checking. In Proceedings of the 16th International Conference on Computer Aided Verification, CAV 2004, Boston, MA, USA, 13–17 July 2004; Alur, R., Peled, D.A., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2004; Volume 3114, pp. 372–386. [CrossRef]
46. Bouajjani, A.; Habermehl, P.; Holík, L.; Touili, T.; Vojnar, T. Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In Proceedings of the 13th International Conference on Implementation and Applications of Automata, CIAA 2008, San Francisco, CA, USA, 21–24 July 2008; Ibarra, O.H., Ravikumar, B., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2008; Volume 5148, pp. 57–67. [CrossRef]
47. Alur, R.; Madhusudan, P. Visibly pushdown languages. In Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, 13–16 June 2004; Babai, L., Ed.; ACM: New York, NY, USA, 2004; pp. 202–211. [CrossRef]
48. Holík, L.; Janku, P.; Lin, A.W.; Rümmer, P.; Vojnar, T. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.* **2018**, *2*, 4. [CrossRef]
49. Balakrishnan, G.; Reps, T.W. Recency-Abstraction for Heap-Allocated Storage. In Proceedings of the 13th International Symposium on Static Analysis, SAS 2006, Seoul, Korea, 29–31 August 2006; Yi, K., Ed.; Lecture Notes in Computer Science; Springer: Berlin, Germany, 2006; Volume 4134, pp. 221–239. [CrossRef]
50. Jensen, S.H.; Jonsson, P.A.; Møller, A. Remedying the eval that men do. In Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, 15–20 July 2012; Heimdahl, M.P.E., Su, Z., Eds.; ACM: New York, NY, USA, 2012; pp. 34–44. [CrossRef]
51. Sharir, M.; Pnueli, A. *Two Approaches to Interprocedural Data Flow Analysis*; NYU CS: New York, NY, USA, 1978.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).