# A Sound Abstract Interpreter for Dynamic Code

Vincenzo Arceri
University of Verona - Dept. of Computer Science
Verona, Italy
vincenzo.arceri@univr.it

Isabella Mastroeni
University of Verona - Dept. of Computer Science
Verona, Italy
isabella.mastroeni@univr.it

## ABSTRACT

Dynamic languages, such as JavaScript, employ string-to-code primitives to turn dynamically generated text into executable code at run-time. These features make standard static analysis extremely hard if not impossible because its essential data structures, i.e., the control-flow graph and the system of recursive equations associated with the program to analyze, are themselves dynamically mutating objects. Hence, the need to handle string-to-code statements approximating what they can execute, and therefore allowing the analysis to continue (even in presence of string-to-code statements) with an acceptable degree of precision. In order to reach this goal, we propose a static analysis allowing us to collect string values and allowing us to soundly over-approximate and analyze the code potentially executed by a string-to-code statement.

## KEYWORDS

Abstract interpretation, Static analysis, Dynamic languages

**ACM Reference Format:**
Vincenzo Arceri and Isabella Mastroeni. 2020. A Sound Abstract Interpreter for Dynamic Code. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3341105.3373964

## 1 INTRODUCTION

The possibility of dynamically building code instructions as the result of text manipulation is a key aspect in dynamic languages. With reflection, programs can turn text, which can be built at run-time, into executable code [30]. These features are often used in code protection and tamper resistant applications, employing camouflage for escaping attack or detection [26], in malware, in mobile code, in web servers, in code compression, and in Just-in-Time (JIT) compilers employing optimized run-time code generation.

While the use of dynamic code generation may simplify considerably the *art and performance of programming*, this practice is also highly dangerous, making the code prone to unexpected behaviors and malicious exploits of its dynamic vulnerabilities, such as code/object-injection attacks for privilege escalation, database corruption, and malware propagation. Clearly more advanced and secure functionalities based on reflection could be permitted if we

```
vd, ac, la = "";                    ac += tt.substring(tt.
v = "wZsZ"; m = "                      indexOf("O"), 3);
   AYcYtYiYvYeYXY";                 ac += tt.substring(tt.
tt = "AObyaSZjectB";                   indexOf("j"), 11);
l = "WYSYcYrYiYpYtY.                while (k+=2 < l.length)
   YSYhYeYlYlY";                     la = la + l.charAt(k);
while (i+=2 < v.length)
 vd = vd + v.charAt(i);             d = vd + "=new " + ac +
while (j+=2 < m.length)                "(" + la + ")";
 ac = ac + m.charAt(j);             eval(d);
```

**Figure 1: A potentially malicious JavaScript program.**

better master how to safely generate, analyze, debug, and deploy programs that dynamically generate and manipulate code.

There are lots of good reasons to analyze programs building strings that can be executed as code. An interesting example is code obfuscation. Recently, several techniques have been proposed for JavaScript code obfuscation[1], meaning that also client-side code protection starts to be a critical aspect to tackle. For example, consider the JavaScript fragment in Fig. 1 where strings are manipulated, de-obfuscated [17], combined together into the variable `d` and finally transformed into code by **`eval`**, the statement `d = new ActiveXObject(WScript.Shell)`. This command, in Internet Explorer, opens a shell which may execute malicious commands. The command is not hard-coded in the fragment but it is built at run-time and the initial values of `i`,`j` and `k` are unknown, such as the number of iterations of the loops in the fragment.

Hence, it is not always possible to simply ignore **`eval`** without losing the possibility of analyzing the rest of the program.

*The problem.* A major problem in presence of dynamic code generation is that static analysis becomes extremely hard if not even impossible. This happens because program's essential data structures, such as the control-flow graph and the system of recursive equations associated with the program to analyze, are themselves dynamically mutating objects: *"You can't check code you don't see"* [5]. Indeed, the only *sound* way analyses have to overcome the execution of code they "don't see" is to suppose that a string-to-code statement can do *anything*, i.e., it can generate *any* possible memory.

```
1     x = 1; a = 1;
2     y = "a++;";
3     while (x<10)
4      y = y + y;
5      eval(y);
6      x++;
```

Hence, when reaching such a statement, an analysis may continue but by accepting to lose any previously gathered information. Let us show this situation on a simple but enough expressive example. Consider the code on the left, where the variable `x` is independent from what is dynamically executed in

[1]https://www.daftlogic.com/projects-online-javascript-obfuscator.htm, http://www.danstools.com/javascript-obfuscate/, http://javascript2img.com/

*SAC '20, March 30-April 3, 2020, Brno, Czech Republic*
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373964

y. Suppose we are interested in analyzing the interval of x at line 6. We can observe that the interval of x at line 6 is precisely $[1, 9]$, and this would be the result of any interval analysis on the code without line 5. Unfortunately, the presence of **eval** makes impossible for the analysis to know whether there is any "hidden" (dynamically generated) modification of x, and therefore it cannot soundly compute the interval of x. This is a simple use of **eval**, but anyway it is not even suitable to code rewriting techniques removing **eval** by replacing it with equivalent code (without **eval**) [22], since the **eval** parameter is not *hard-coded* but dynamically generated.

Clearly, the only way to make the analysis aware of the fact that the execution of **eval** does not modify x is to compute, or at least to over-approximate, what is executed in **eval**. Hence, we first need an abstract domain for *collecting* the strings of variables, such as y in the example. Unfortunately, this is not sufficient, since once we have an over-approximation of the values of y we need to "execute" it for analyzing the potential effects on x. Hence, we also need to extract, from any abstract value, (an over-approximation of) the code that could be executed by **eval**. The idea, at this point, is that of (recursively) calling the abstract interpreter, for the performed analysis, on this *approximated* code. In the example, we could over-approximate the language of y as an arbitrarily long concatenation of strings `"a++;"`.[2] Anyway, what is clear is that any code we can synthesize from this language cannot add any statement modifying x. In particular, the call of the analysis on the synthesized code will surely return a memory where a is changed, but such that the analysis of x can continue unaffected.

In this paper, we tackle the problem of analyzing dynamic code by *treating code as any other dynamic structure that can be statically analyzed by abstract interpretation* [13], and to treat the abstract interpreter as any other program function that can be recursively called on a piece of code. In order to obtain this it is necessary to tackle two main issues:

- Since we have to collect the strings that may be argument of an **eval**, we need an abstract domain for strings collecting, as much faithfully as possible, the set of possible values that a string variable may receive before **eval** executes it.
- Since we have to analyze the code potentially executed by **eval**, we need to extract from its abstract argument an abstraction of the code that **eval** may execute. Clearly, this abstraction must be in a form that the analyzer can interpret.

As far as the first issue is concerned, we choose regular languages as abstraction [4], since they are both enough precise for analyzing string properties in general, and suitable (by considering their finite representation as finite state automata) for building algorithms able to extract/approximate the executable sub-language of the string when an **eval** occurs. As far as the second issue is concerned, we choose control-flow graphs (CFG for short) as code abstraction, where the abstraction relation is the semantic inclusion relation, i.e., a CFG $G_1$ is more abstract than the CFG $G_2$ if the set of executions of $G_1$ contains the set of executions of $G_2$. In this way, we guarantee a sound by construction (due to the abstract interpretation framework) abstraction of the code executed by **eval**. It is clear that we have to transform the automaton $A$, generated by the string analysis, in a CFG which over-approximates the executable strings recognized by $A$. The result is a first step towards a static analyzer for dynamic code containing non removable **eval** statements, that still have some limitations (as we will explain in Sect. 5.1) but which provides the necessary ground for studying more general solutions for the problem.

[2] It is an *over*-approximation since the program executes at most 9 concatenations.

$$
\begin{aligned}
\mathrm{Exp} \ni \mathsf{e} &::= \mathsf{a} \mid \mathsf{b} \mid \mathsf{s} \\
\mathrm{AExp} \ni \mathsf{a} &::= x \mid n \mid \mathbf{len}(\mathrm{s}) \mid \mathbf{num}(\mathrm{s}) \mid \mathsf{a} + \mathsf{a} \mid \mathsf{a} - \mathsf{a} \mid \mathsf{a} * \mathsf{a} \\
\mathrm{BExp} \ni \mathrm{b} &::= x \mid \mathbf{true} \mid \mathbf{false} \mid \mathrm{e} = \mathrm{e} \mid \mathrm{e} > \mathrm{e} \mid \mathrm{e} < \mathrm{e} \mid \mathrm{b} \wedge \mathrm{b} \mid \neg \mathrm{b} \\
\mathrm{SExp} \ni \mathrm{s} &::= x \mid \text{" "} \mid \text{"}\sigma\text{"} \mid \mathbf{concat}(\mathrm{s}, \mathrm{s}) \mid \mathbf{substr}(\mathrm{s}, \mathsf{a}, \mathsf{a}) \\
\mathrm{Comm} \ni \mathrm{c} &::= {}^{\ell_1}\mathbf{skip}^{\ell_2} \mid {}^{\ell_1}x := \mathrm{e}^{\ell_2} \mid {}^{\ell_1}\mathrm{c}; {}^{\ell_2}\mathrm{c}^{\ell_3} \mid {}^{\ell_1}\mathbf{eval}(\mathrm{s})^{\ell_2} \\
&\mid {}^{\ell_1}\mathbf{if}\ (\mathrm{b})\ \{{}^{\ell_2}\mathrm{c}^{\ell_3}\}\ \mathbf{else}\ \{{}^{\ell_4}\mathrm{c}^{\ell_5}\}^{\ell_6} \mid {}^{\ell_1}\mathbf{while}\ (\mathrm{b})\ \{{}^{\ell_2}\mathrm{c}^{\ell_3}\}^{\ell_4} \\
\mu\mathrm{JS} \ni \mathsf{P} &::= {}^{\ell_\iota}\mathrm{c}; {}^{\ell_2} \qquad \text{where } n \in \mathbb{Z}, \sigma \in \Sigma^*, x \in \mathsf{Id}
\end{aligned}
$$

**Figure 2: Syntax of $\mu$JS.**

This provides us both, with a proof of concept that a sound approximation of semantics of dynamically generated programs is possible in abstract interpretation, and with a static analyzer for a core dynamic language. The choice of considering a restricted language is just for focusing the attention on the approach, namely on the analysis architecture and on the algorithms proposed.

## 2 THE ANALYSIS INGREDIENTS

In this section, we focus on the problem of defining an abstract (collecting) semantics for dynamic programs, namely programs containing string-to-code statements such as **eval**. This means, that, as observed in the introduction, we need a semantics able to collect strings, sufficiently precise for inferring an approximation of what could be executed by the string to code statement, but also not too complex, in order to guarantee the effectiveness of the analysis.

### 2.1 The language: $\mu$JS

In this work we propose a language-independent framework for analyzing dynamic code, for this reason we consider an imperative language plus **eval**, $\mu$JS in Fig. 2. Each $\mu$JS statement is annotated with a label $\ell \in \mathit{Lab}$ corresponding to its program point in P. Let $\ell_\iota$ and $\ell_f$ be two special labels identifying the initial and the final program points, respectively. We denote by $\mathit{Lab}_\mathsf{P}$ the labels of P.

### 2.2 Analyzing $\mu$JS programs

In this section, we recall the static analysis process and the necessary semantic transformers corresponding to statements of $\mu$JS. In order to analyze a program $\mathsf{P} \in \mu$JS, we analyze its CFG, which embeds the control structure of the P. We follow [31] for the construction of the CFG , where each node is a program point, and each edge is labeled with a statement or a guard that is the *effect/action* from its input node to its output node. Given $\mathsf{P} \in \mu$JS, we suppose to generate the corresponding CFG following [31]. In Fig. 3 we report an example of CFG. It is worth noting that the language of CFG is slightly different from $\mu$JS [31], and in particular it is generated by the grammar: $\mu\mathrm{JS}^{\mathrm{CFG}} \ni \mathtt{l} ::= x := \mathrm{e} \mid \mathrm{b} \mid \mathbf{eval}(\mathrm{s})$. Given a program $\mathsf{P} \in \mu$JS, the corresponding CFG $\mathsf{G}_\mathsf{P} \triangleq \textsc{Cfg}(\mathsf{P}) = \langle \mathrm{N}_\mathsf{P}, \mathrm{E}_\mathsf{P}, \mathrm{I}_\mathsf{P}, \mathrm{O}_\mathsf{P} \rangle$ where nodes are $\mathrm{N}(\mathsf{G}_\mathsf{P}) \triangleq \mathit{Lab}_\mathsf{P}$, the input node (without incoming edges) $\mathrm{I}(\mathsf{G}_\mathsf{P}) \triangleq \ell_\iota$, the output node (without outgoing edges)

```
1x := "";
2while (B)
 {3 if (B')
  {4x:=concat(x,"a:=1;")
     5};
  else
  {6x:=concat(x,"b:=1;")
     7}8
};9
```

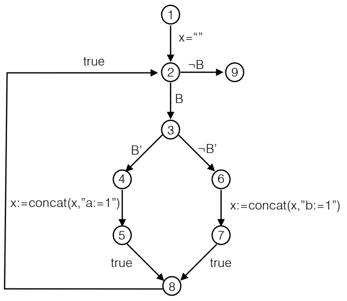


**Figure 3: Example of** CFG **.**

$\mathrm{O}(\mathsf{G_P}) \triangleq \ell_\mathsf{f}$, and $\mathrm{E}(\mathsf{G_P}) \subseteq \mathrm{N}(\mathsf{G_P}) \times \mu\mathsf{JS}^{\mathsf{CFG}} \times \mathrm{N}(\mathsf{G_P})$ is the set of the CFG edges. Let define the set of computations of a CFG as follows.

$$Paths(\mathsf{G}) \triangleq \left\{ \mathtt{l}_0\mathtt{l}_1\dots\mathtt{l}_k \;\middle|\; \begin{array}{l} \forall i \leq k. \langle \ell_i, \mathtt{l}_i, \ell_{i+1}\rangle \in \mathrm{E}(\mathsf{G}) \\ \ell_0 = \mathrm{I}(\mathsf{G}), \ell_{k+1} = \mathrm{O}(\mathsf{G}) \end{array} \right\}$$

Our aim is to analyze $\mu$JS programs by analyzing their CFG . In particular, we need to define a semantics for $\mu\mathsf{JS}^{\mathsf{CFG}}$. First of all, we define the set of (collecting) memories as $\mathbb{M} \triangleq \mathsf{Id} \longrightarrow \wp(\mathbb{Z}) \cup \wp(\{\mathbf{false}, \mathbf{true}\}) \cup \wp(\Sigma^*)$, ranged over $\mathbb{m}$, which is a set of maps associating with each variable a collection of possible values. We denote by $\mathbb{m}_\varnothing$ the memory associating $\varnothing$ with any variable, and $\mathbb{m}_\top$ associating the set of all possible values with each variable. The update of memory $\mathbb{m}$ for $x \in \mathsf{Id}$ with set of values $v$ is denoted by $\mathbb{m}[x/v]$, while lub and glb of memories are computed point-wise, i.e., $\mathbb{m}_1 \sqcup \mathbb{m}_2(x) = \mathbb{m}_1(x) \cup \mathbb{m}_2(x)$ and $\mathbb{m}_1 \sqcap \mathbb{m}_2(x) = \mathbb{m}_1(x) \cap \mathbb{m}_2(x)$. We can now define how each CFG edge transforms its current state. The semantics of statements $\mathsf{c} \in \mu\mathsf{JS}^{\mathsf{CFG}}$ on $\mathbb{m}$ is defined as the function $[\![\mathsf{c}]\!] : \mathbb{M} \longrightarrow \mathbb{M}$, where $(\!|\cdot|\!)$ denotes the collecting semantics of expressions, defined as additive lift of the standard expression semantics.

$$\begin{array}{rcl} [\![x := \mathsf{e}]\!]\,\mathbb{m} & = & \mathbb{m}[x/(\!|\mathsf{e}|\!)\,\mathbb{m}] \\ [\![\mathsf{b}]\!]\,\mathbb{m} & = & \mathbb{m} \sqcap \bigsqcup \left\{ \mathbb{m}_t \;\middle|\; (\!|\mathsf{b}|\!)\,\mathbb{m}_t = \mathbf{true} \right\} \\ [\![\mathbf{eval}(\mathsf{s})]\!]\,\mathbb{m} & = & \bigsqcup_{\mathsf{c} \in \mathsf{C}} [\![\mathsf{c}]\!]\,\mathbb{m} \qquad \text{where} \quad \mathsf{C} \triangleq (\!|\mathsf{s}|\!)\,\mathbb{m} \Cap \mu\mathsf{JS} \end{array}$$

where $\Cap$ is the intersection in the set of $\mu$JS programs, formally let $\mathcal{S}$ a function mapping any sequence of strings of $(\Sigma^*)^*$ on its string counterpart on $\Sigma^*$ (and, abusing notation, also its additive lift to sets of sequences), and let $tocode(\sigma)$ the function interpreting a string $\sigma$ as code, if possible (and as **skip** otherwise), then for any $L \subseteq \Sigma^*$ we define $L \Cap \mu\mathsf{JS} \triangleq \left\{ tocode(\sigma) \;\middle|\; \sigma \in L \cap \left\{ \mathcal{S}(\delta) \;\middle|\; \delta \in \mu\mathsf{JS} \right\} \right\}$.

This semantics is standard for assignments and guards, while when an **eval** is met, we have to extract from the collection of strings for its argument only those strings corresponding to executable programs of $\mu$JS, we execute all these programs and we join all the resulting memories. Clearly, if the semantics of the **eval** input is an infinite set then the semantics of **eval** is undecidable. We can extend this definition of semantics to paths in a CFG $\mathsf{G_P}$: Let $\pi \in Paths(\mathsf{G_P})$, $\pi = \mathtt{l}_0\mathtt{l}_1\dots\mathtt{l}_k$, and $\mathbb{m} \in \mathbb{M}$ then $[\![\pi]\!]\,\mathbb{m} \triangleq [\![\mathtt{l}_k]\!] \circ \dots \circ [\![\mathtt{l}_1]\!] \circ [\![\mathtt{l}_0]\!]\,\mathbb{m}$ [31]. Note that, given a program $\mathsf{P}$ and $\mathsf{G_P} = \textsc{Cfg}(\mathsf{P})$, it is well known [31] that

$$\forall \mathbb{m} \in \mathbb{M}.\, \exists \Pi \in Paths(\mathsf{G_P}).\; [\![\mathsf{P}]\!]\,\mathbb{m} = \bigcup_{\pi \in \Pi} [\![\pi]\!]\,\mathbb{m} \tag{1}$$

where the $[\![\mathsf{P}]\!]\,\mathbb{m}$ is the collection of the executions of $\mathsf{P}$ on the concrete memories collected in $\mathbb{m}$.

**Algorithm 1** Static analysis on CFG of $\mathsf{P}$.

```
Require: CFG $\mathsf{G_P} = \langle \mathrm{N_P}, \mathrm{E_P}, \mathrm{I_P}, \mathrm{O_P} \rangle$, initial store $\mathbb{s}_0$
Ensure: $\mathbb{s}$ fix-point of the collecting memories for each $\ell \in Lab_\mathsf{P}$
1: procedure ANALYZE($\mathsf{G_P}$, $\mathbb{s}_0$)
2:     $\mathbb{s} \longleftarrow \mathbb{s}_0$; $\mathbb{s}' \longleftarrow \varnothing$
3:     while $\mathbb{s} \neq \mathbb{s}'$ do
4:         $\mathbb{s}' \longleftarrow \mathbb{s}$
5:         for $\langle \ell_1, \mathsf{c}, \ell_2 \rangle \in \mathrm{E_P}$ do
6:             $\mathbb{s} \longleftarrow \mathbb{s}[\mathbb{s}_{\ell_2} / [\![\mathsf{c}]\!]\,\mathbb{s}_{\ell_1} \sqcup \mathbb{s}_{\ell_2}]$
       return $\mathbb{s}$
```

At this point, we use this semantic transformer for analyzing $\mu$JS programs by computing the fix-point of the semantics for each program point. We recast the standard static analysis fix-point algorithm [28] in our notation. Firstly, we introduce *flow-sensitive* stores $\mathbb{S} \triangleq Lab_\mathsf{P} \longrightarrow \mathbb{M}$, ranged over $\mathbb{s}$, which is a sequences of memories for each program point, associating with each program point a memory. We denote by $\mathbb{s}_\ell$ the memory at line $\ell$, i.e., $\mathbb{s}(\ell)$. Given a store $\mathbb{s}$, the update of memory $\mathbb{s}_\ell$ with a memory $\mathbb{m}$ is denoted $\mathbb{s}[\mathbb{s}_\ell/\mathbb{m}]$ and provides a new store $\mathbb{s}'$ s.t. $\mathbb{s}'_\ell = \mathbb{m}$ while $\forall \ell' \neq \ell$ we have $\mathbb{s}'_{\ell'} = \mathbb{s}_{\ell'}$. Finally, let $\mathbb{s}_\varnothing$ be the store where all the memories associate with all the variables the empty set, i.e., $\forall \ell \in Lab_\mathsf{P}.\; \mathbb{s}_\varnothing(\ell) = \mathbb{m}_\varnothing$. Then the analysis algorithm is Alg. 1, whose result is a store $\mathbb{s}$ s.t. for each $\ell \in Lab_\mathsf{P}$, $\mathbb{s}_\ell$ is the fix-point collecting memory for $\ell$, namely the set of all the possible values associated with each variable at the program point $\ell$.

## 3 DYNAMIC LANGUAGE ANALYSIS

It is well known that Alg. 1 may diverge on concrete memories. Hence, in order to ensure convergence, we need abstraction, as it is usual in static analysis. Unfortunately, this is not sufficient to avoid divergence when the code is dynamic. We have already observed that the collection of potential executable strings reaching an **eval** argument may be infinite, implying that, as it happens for data values, we need to abstract also code in order to enforce convergence. Moreover, there is another potential subtle source of divergence due to the unpredictability of the code to execute in dynamic languages. Let us consider the following fragment:

```
ℓ1x := "eval(x)";ℓ2eval(x);ℓ3
```

The **eval** call actives an infinite nested call chain to **eval**. This divergence comes directly from the meaning of dynamically generated code from strings and cannot be controlled by the semantics once we execute **eval**. In the following of the section we tackle these three problems separately, by suitably abstracting data, and in particular strings preparing the field for analyzing **eval** (Sect. 3.1); by approximating code executed by **eval** in order to recursively call the analysis algorithm on the abstracted code (Sect. 3.2 and 4); and by controlling the **eval** nested calls depth (Sect. 3.3).

### 3.1 Abstracting data

In order to solve the first source of divergence, we have to consider a suitable abstraction of data, combining an abstraction of numerical values, of boolean and of strings. In particular, we abstract integers to the well known interval domain Int for integers [15] (with widening for avoiding computation divergence) and we use the identity on booleans: $\mathrm{Bool} \triangleq \wp(\{\mathbf{true}, \mathbf{false}\})$.

As far as strings is concerned, we need to collect strings during computation in order to be able to extract and approximate what is executable when an **eval** statement is met. Therefore, the resulting domain should have to approximate the set of possible strings and it has to keep enough information for allowing us to extract code from it, but it has also to keep enough information for analyzing properties of strings that are never executed by an **eval** during computation. For instance, if we consider abstract parsing (w.r.t. the programming language grammar) for abstracting strings [16], we could probably keep enough information allowing us to over-approximate the executed code, when the analyzed string is executed, but we lose any other kind of information on strings that could be useful for analyzing strings that are never executed. We believe that a good choice, meeting all these requirements, are finite state automata (regular languages), since regular languages are enough precise for analyzing string properties in general, and since their finite representation (by means of finite state automata) is suitable for building algorithms able to extract the executable sub-language of strings in presence of string-to-code statements.

The finite state automata (FA) domain has been introduced in [4] for analyzing string manipulation programs, that over-approximates strings as regular languages, represented by the minimum deterministic finite state automata recognizing them. A FA $A$ is a tuple $(Q, \delta, q_0, F, \Sigma)$, where $Q$ is the set of states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\Sigma$ is the finite alphabet of symbols. The domain is $\langle \text{DFA}_{/\equiv}, \sqsubseteq_{\text{DFA}}, \sqcup_{\text{DFA}}, \sqcap_{\text{DFA}}, \mathsf{Min}(\varnothing), \mathsf{Min}(\Sigma^*)\rangle$, where $\text{DFA}_{/\equiv}$ is the quotient set of DFA w.r.t. the equivalence relation induced by language equality, $\sqsubseteq_{\text{DFA}}$ is the partial order induced by language inclusion, $\sqcup_{\text{DFA}}$ and $\sqcap_{\text{DFA}}$ are the lub and glb operators, corresponding to automata union and intersection, respectively. The minimum is $\mathsf{Min}(\varnothing)$, corresponding to the FA recognizing the empty language and the maximum is $\mathsf{Min}(\Sigma^*)$, corresponding to the FA recognizing any string of $\Sigma^*$. We abuse notation by representing equivalence classes in the domain $\text{DFA}_{/\equiv}$ by one of its FA (usually the minimum), i.e., when we write $A$ we mean $[A]_\equiv$. Since the domain $\text{DFA}_{/\equiv}$ is infinite, and it is not ACC, i.e., it contains infinite ascending chains, it is equipped with the parametric widening $\nabla^n_{\text{DFA}}$. The latter is defined in terms of a state equivalence relation merging states that recognize the same language, up to a fixed length $n \in \mathbb{N}$, a parameter used for tuning the widening precision [18]. By changing $n$, we obtain different widening operators [18]. In particular, the parameter $n$ tunes the length of the strings determining the equivalence of states and therefore used for merging them in the widening: the smaller is $n$, the more information will be lost by widening automata. For instance, let us consider the automata $A, A' \in \text{DFA}_{/\equiv}$ s.t., $\mathscr{L}(A) = \{\epsilon, a\}$ and $\mathscr{L}(A') = \{\epsilon, a, aa\}$. The result of the application of the widening $\nabla^1_{\text{DFA}}$ is $A \, \nabla^3_{\text{DFA}} \, A' = A''$ s.t. $\mathscr{L}(A'') = \{\, a^n \mid n \in \mathbb{N} \,\}$.

At this point, we can combine all the three domains, $\mathsf{Int}$, $\mathsf{Bool}$ and $\text{DFA}_{/\equiv}$. The way we combine these domains is not relevant for **eval** analysis, but it may be relevant for other language aspects (e.g., type juggling [3]). The simplest combination is the coalesced sum [12]. In the following, we denote by $\mathfrak{m}^\# \in \mathbb{M}^\#$ the abstract memories, associating, with each variable, values in the abstract domain just described. We denote by $(\!|\cdot|\!)^\# \mathfrak{m}^\#$ the abstract expression semantics, and by $[\![\cdot]\!]^\# \mathfrak{m}^\#$ the abstract semantics, computed on this domain.

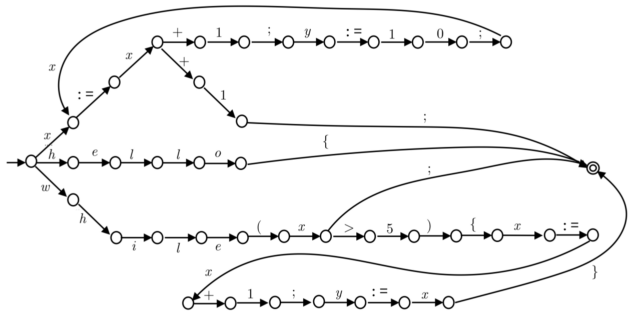


**Figure 4: FA $A_{\mathsf{ds}}$ abstract value of ds at line 14 of Ex. 3.1.**

*The* **eval** *abstract semantics.* At this point, we are able to compute a static analysis of strings, where strings are abstracted in FA. In the concrete, **eval** turns strings into executable code, hence, in the abstract, we need to approximate the sub-language of only executable strings. The abstract semantics of **eval** is

$$[\![\mathbf{eval}(\mathsf{s})]\!]^\# \mathfrak{m}^\# = \bigsqcup_{\mathsf{c} \in \mathsf{C}} [\![\mathsf{c}]\!]^\# \mathfrak{m}^\# \qquad \text{where } \mathsf{C} \triangleq \mathscr{L}((\!|\mathsf{s}|\!)^\# \mathfrak{m}^\#) \Cap \mu\mathsf{JS}$$

*Example 3.1.* Consider the following $\mu$JS program P. For the sake of readability, we omit the **else** empty branches.

```
1while (x++ < 3)
 {2os := os + "xA:=Bx+1B;y:=1A0;x:=Bx+1A;"3};
4if (x > 10)
 {5os := "whiAleB(x>5A)A{x:A=x+1;y:=x};B"6};
7if (x = 5)
 {8os := "hello{"9};
10if (x = 8)
 {11os := "while(x;"12};
13ds := deobf(os);14eval(ds);15
```

where ds = deobf(os) is a syntactic sugar for the string transformer that removes the chars "A" and "B" from the string. In Fig. 4 we depict the abstract value of ds after the program line 13, computed analyzing strings on the $\text{DFA}_{/\equiv}$ domain, w.r.t. the widening $\nabla^3_{\text{DFA}}$. At this point, the idea is to remove from the FA all the non-executable strings. This corresponds to perform the intersection between the (regular) language computed as the abstract value of ds (denoted by $\mathscr{L}((\!|\mathsf{ds}|\!)^\# \mathfrak{m}^\#)$ for a given memory $\mathfrak{m}^\#$) and the (context-free) $\mu$JS language (also denoted by $\mu$JS): $\mathscr{L}((\!|\mathsf{ds}|\!)^\# \mathfrak{m}^\#) \Cap \mu\mathsf{JS}$.

Note that, the intersection between a context-free language and a regular language (which is our case) is always a context-free language. This means that we could remove the non-executable FA paths by performing the intersection above, but unfortunately the computation of this intersection could be costly in practice due to the size of a real language grammar.

The semantics of the other labels in $\mu\mathsf{JS}^{\mathsf{CFG}}$ can be also abstracted on our abstract domain:

$$\begin{aligned} [\![x := \mathsf{e}]\!]^\# \mathfrak{m}^\# &= \mathfrak{m}^\#[x / (\!|\mathsf{e}|\!)^\# \mathfrak{m}^\#] \\ [\![\mathsf{b}]\!] \mathfrak{m}^\# &= \mathfrak{m}^\# \sqcap^\# \bigsqcup\nolimits^\# \left\{ \mathfrak{m}^\#_t \;\middle|\; (\!|\mathsf{b}|\!)^\# \mathfrak{m}^\#_t = \mathbf{true} \right\} \end{aligned}$$

where $\sqcap^\#$ and $\sqcup^\#$ are, respectively, the glb and the lub between abstract values associated to variables. Finally, let us observe how

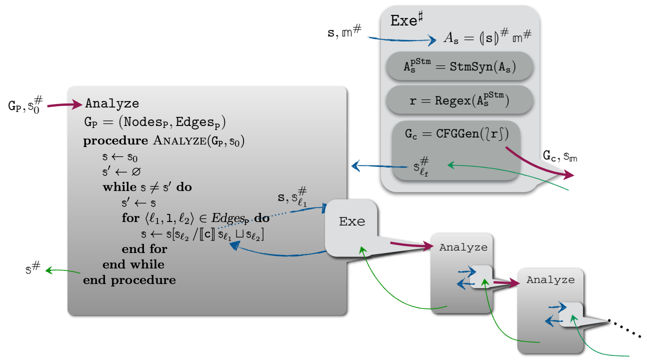


**Figure 5: Analyzer architecture and call execution structure.**

Eq. 1 is rewritten on the abstract semantics

$$\forall\, m^\# \in \mathbb{M}^\#.\ \exists \Pi \subseteq Paths(G_P).\ [\![P]\!]^\# m^\# \subseteq \bigcup_{\pi \in \Pi} [\![\pi]\!]^\# m^\# \tag{2}$$

## 3.2 The analyzer architecture

In this section, we have to characterize the sub-language of executable strings of a FA in a constructive way. Moreover, **eval** turns strings into executable code, hence, once we have the FA of the sub-language of executable strings in the abstract domain, we need to turn FA into executable code. Namely, we have to *synthesize* from the FA an approximation of a $\mu$JS program that is a sound approximation of the code that may be executed in the concrete execution. Hence, we provide an algorithmic approach for approximating in a decidable way the test $\mathscr{L}((|s|)^\# m^\#) \Cap \mu$JS, by building a CFG that soundly approximates the executable $\mu$JS programs in $\mathscr{L}((|s|)^\# m^\#)$, i.e., whose semantics soundly approximates the semantics of the code that may be executed by **eval**. This allows us to recursively call the abstract interpreter on the synthesized CFG . This original approach works by steps: Let $[\![\mathbf{eval}(s)]\!]^\# m^\#$ be the semantics the analyzer has to compute

(1) First, we have to clean up the language $\mathscr{L}((|s|)^\# m^\#)$ from all the strings that are surely not executable. This is obtained by visiting the FA $A_s = (|s|)^\# m^\#$ and by keeping only those paths that can be executable. It should be clear that a FA cannot recognize precisely a context free language, hence we still keep in the resulting FA not executable strings, in particular those that do not respect the balanced bracketing. Let us denote the resulting FA $A_s^{pStm} \triangleq \mathtt{StmSyn}(A_s)$;
(2) From the so far obtained automaton $A_s^{pStm}$, the aim is to build a CFG , over-approximating the executable strings in the FA, i.e., $G_s \triangleq \mathtt{CFGGen}(A_s^{pStm})$. Then on this CFG the analyzer can be recursively called.

The whole architecture is given in Fig. 5, where the procedure $\mathtt{Exe}^\#$ encapsulates (1) and (2) and their details are in the next section.

## 3.3 Abstracting sequences of eval nested calls

We have previously described the architecture of the analysis, which recursively calls the analysis on the synthesized CFG when an **eval** occurs. Due to unpredictability of the code that can be generated, it is impossible to foresee from the program code whether the recursive sequence of calls will terminate. At the beginning of Sect. 3, we have seen a quite simple example with a divergent recursion, but in general this kind of situations may be hard to detect and is clearly out of the scope of the abstraction made on data (and of its widening). If the program using **eval** terminates, then there must be a maximal depth of nested calls to **eval**, and therefore we can ensure enough precision until a maximal degree of nested calls to **eval**. However, to extract this maximal depth is in general undecidable.

In order to approximate this maximal depth of nested **eval** call, we can introduce a *nested call widening*, which consists of fixing a threshold of allowed height of towers. Once we reach the threshold, the only way to keep soundness consists of approximating the collection of values for any variable to the top, when the threshold is overcome, meaning that after the threshold anything can be computed. In this way, we guarantee soundness by fixing a degree of precision in observing the nesting of **eval** statements.

## 4 APPROXIMATING EXECUTABLE CODE

In this section, we go into the details of how the synthesis of the CFG executed by an **eval** works, i.e., how $\mathtt{Exe}^\#$ works. The abstract interpreter reported in Fig. 5, when an **eval** is met, calls $\mathtt{Exe}^\#$ on the FA approximating the **eval** input. For example, at line 13 of Ex. 3 we need to execute $\mathtt{Exe}^\#$ on $A_{\mathtt{ds}}$. In particular, $\mathtt{Exe}^\#$ goes through two steps: (1) extract from a FA the sub-language of executable strings (StmSyn); (2) generate from the FA of the sub-language of executable strings a CFG (CFGGen). In the following, we describe these two sub-modules of $\mathtt{Exe}^\#$.

## 4.1 StmSyn: Extracting the executable language

The first step consists of reducing the number of states of the FA, by over-approximating every string recognized as a statement, or partial statement, in $\mu$JS. The idea is to derive, starting from the original FA $A_s$ (generated by the string analysis), whose alphabet is the set of characters $\Sigma$, a new FA whose alphabet is a set of strings. These strings are obtained by collapsing consecutive edges, in $A_s$, up to any punctuation symbol in $\mathtt{Punct} \triangleq \{;, \{, \}, (, )\}$. In particular, any executable statement ends with a semicolon by language definition, while the braces allow us to split strings when the body of a **while** or of an **if** either begins or ends, finally the parentheses recognize the begin and the end of a parenthesized expression (the guard of an **if** or a **while**). In particular, we define a set of *partial statements*, that is a regular over-approximation of the $\mu$JS grammar, which will be the alphabet of the resulting FA. The *partial statements* $\Sigma_{\mathtt{pStm}} \subseteq \Sigma^*$ are defined as follows

$$\Sigma_{\mathtt{pStm}} \triangleq \mathtt{Punct} \cup \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} \text{x is a maximal substring of a } \mu\text{JS state-} \\ \text{ment between two punctaction symbols} \\ \text{(first punctaction symbol excluded)} \end{array} \right\}$$

At this point, the idea is that of transforming the FA $A_s$ on the alphabet $\Sigma$ in the FA $A_s^{pStm}$ on the alphabet $\Sigma_{\mathtt{pStm}}$, removing any string recognized by $A_s$ which will be surely not executable. The soundness constraints obviously consists in guaranteeing that any executable string is not lost by this transformation.

In order to derive the FA $A_s^{pStm}$, we design the procedure StmSyn (Alg. 2) taking as input a FA on $\Sigma$ (i.e., $A_s$ for $\mathbf{eval}(s)$ and returning the FA on a finite subset of $\Sigma_{\mathtt{pStm}}$. In particular, the idea of Alg. 2 is to perform, starting from $q_0$, a visit of the states recursively

**Algorithm 2** Building the FA.

```
Require: An FA $A = (Q, \delta, q_0, F, \Sigma)$
Ensure: An FA $A' = (Q', \delta', q_0, F', \Sigma_{\mathsf{pStm}})$
 1: procedure StmSyn($A$)
 2:     $Q' \leftarrow \{q_0\}$; $F' \leftarrow F \cap \{q_0\}$; $\delta' \leftarrow \emptyset$, Visited $\leftarrow \{q_0\}$;
 3:     stmsyntr($q_0$);
 4: procedure stmsyntr(q)
 5:     $B \leftarrow$ Build$(A, q)$;
 6:     Visited $\leftarrow$ Visited $\cup \{q\}$; $Q' \leftarrow Q' \cup \{ p \mid (\mathsf{a}, p) \in B \}$;
 7:     $F' \leftarrow Q' \cap F$; $\delta' \leftarrow \delta' \cup \{ (q, \mathsf{a}, p) \mid (\mathsf{a}, p) \in B \}$;
 8:     $W \leftarrow \{ p \mid (\mathsf{a}, p) \in B \} \smallsetminus$ Visited;
 9:     while $W \neq \emptyset$ do
10:         select $p$ in $W$ ($W \leftarrow W \smallsetminus \{p\}$);
11:         stmsyntr($p$);
```

**Algorithm 3** Statements recognized from a state $q$.

```
Require: An FA $A = (Q, \delta, q_0, F, \Sigma)$
Ensure: $I_q$ set of all pairs (partial statement,reached state)
 1: procedure Build($A, q$)
 2:     $I_q \leftarrow \emptyset$;    buildtr(q,$\varepsilon$,$\emptyset$)
 3: procedure buildtr(q,word,Mark)
 4:     $\Delta_q \leftarrow \{ (\sigma, p) \mid \delta(q, \sigma) = p \}$
 5:     while $\Delta_q \neq \emptyset$ do
 6:         select $(\sigma, p)$ in $\Delta_q$ ($\Delta_q \leftarrow \Delta_q \smallsetminus \{(\sigma, p)\}$)
 7:         if $(q, p) \notin$ Mark then
 8:             if $\sigma \notin$ Punct $\wedge$ $p \notin F$ then
 9:                 buildtr(p,word.$\sigma$,Mark$\cup\{$(q,p)$\}$)
10:             if $\sigma \in$ Punct $\wedge$ word.$\sigma \in \Sigma_{\mathsf{Syn}}$ then
11:                 $I_q \leftarrow I_q \cup \{(\text{word}.\sigma, p)\}$
12:             if $p \in F \wedge$ word.$\sigma \in \Sigma_{\mathsf{Syn}}$ then
13:                 $I_q \leftarrow I_q \cup \{(\text{word}.\sigma, p)\}$
```

identified by Alg. 3, that is the states reached by $q_0$ reading partial statements, and to recursively replace the sequences of edges that recognize a symbol in $\Sigma_{\mathsf{pStm}}$ with a single edge labeled by the corresponding string. Alg. 3 scans the edges of the original FA $A_{\mathsf{s}}$ and, when a punctuation symbol occurs or a final state is reached, it verifies whether the string read so far is in $\Sigma_{\mathsf{pStm}}$, otherwise it is discarded: This *executability* check is performed at lines 8 and 10 and ensures, for any state $q$ of the FA $A_{\mathsf{s}}$, that $I_q$ contains only (partial) statements of $\mu$JS. In particular, from $q_0$ we reach the states computed by Build($q_0$), and the corresponding read words. Recursively, we apply Build to these states, following only those edges that we have not already visited. For instance, in Fig. 6 we have the FA $A^{\mathsf{pStm}}_{\mathsf{ds}}$ = StmSyn($A_{\mathsf{ds}}$). Note that the string `hello{` is not in $A_{\mathsf{ds}}$ since it is discarded by Alg. 2, because it does not belong to $\Sigma_{\mathsf{pStm}}$. Instead, the string **while**(x; is still recognized by the resulting FA even if it is not executable (this is due to the fact that FA cannot recognize the balanced parenthesisation). Next theorem tells us that any executable string collected during computation is kept in the transformed FA, guaranteeing the algorithm soundness.

Theorem 4.1. *Let* s $\in$ SExp*, let* $A_{\mathsf{s}}$ *be the FA recognizing the strings associated with* s*, and* $A^{\mathsf{pStm}}_{\mathsf{s}} \triangleq$ StmSyn($A_{\mathsf{s}}$)*, then* $\forall \sigma \in \mathscr{L}(A_{\mathsf{s}}) \Cap \mu JS.\ \exists \delta \in \mathscr{L}(A^{\mathsf{pStm}}_{\mathsf{s}})$ *s.t.* $tocode(\mathcal{S}(\delta)) = \sigma$.

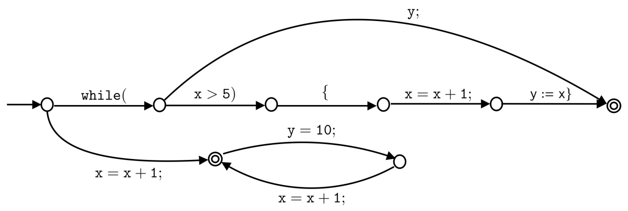


**Figure 6: FA** $A^{\mathsf{pStm}}_{\mathsf{ds}}$ = **StmSyn**($A_{\mathsf{ds}}$).

We can observe that the procedure Build($A, q$) executes a number of recursive-call sequences equal to the number of maximal acyclic paths starting from $q$ on $A$. The number of these paths can be computed as $\sum_{q \in Q}(out(q) - 1) + 1$, where $out(q)$ is the number of outgoing edges from $q$. The worst case depth of a recursive-call sequence is $|Q|$. Thus, the worst case complexity of Build (when $out(q) = |Q| \times |\Sigma|$ for all $q \in Q$) is $O(|Q|^3)$. Concerning StmSyn, we can observe that in the worst case we keep in StmSyn($A$) all the $|Q|$ states of $A$, hence in this case we launch $|Q|$ times the procedure Build. Hence, the worst case complexity of StmSyn is $O(|Q|^4)$.

## 4.2 CFGGen: Control-flow graph generation

At this point, the idea is to use the so far obtained FA over $\Sigma_{\mathsf{pStm}}$ to generate a CFG approximating the program executed in **eval**(s). This phase is handled by the procedure CFGGen and works by several steps. It is well known that a FA $A$ can be equivalently rewritten as a regular expression (regex for short) r, s.t. $\mathscr{L}(A) = \mathscr{L}(\mathsf{r})$ [8]. Let *RE* be the domain of regexes over $\Sigma_{\mathsf{pStm}}$, and Regex : FA $\longrightarrow$ *RE* be such an extractor. In the running example, $\mathsf{r}_{\mathsf{ds}}$ = Regex($A^{\mathsf{pStm}}_{\mathsf{ds}}$) is the following regex:

$$\mathsf{r}_{\mathsf{ds}} = \texttt{x:=x+1;} \parallel \textbf{while}\texttt{(x;} \parallel \textbf{while}\texttt{(x>5)\{x:=x+1;y:=x\};} \parallel \texttt{x:=x+1;(y:=10;x:=x+1;)}^*$$

where $\parallel$ and $^*$ respectively correspond to the disjunction and the Kleene-star between regexes. The analyzer implements the Brzozowski algebraic method [8] to convert a FA to an equivalent regex.[3] At this point, we pass through an augmented version of $\mu$JS before generating a CFG . We add to the $\mu$JS boolean expressions a *statically unknown guard* $\circledast$, namely BExp ::= $\cdots \mid \circledast$, that must be intended as a boolean expression that both evaluates to **true** and **false** (i.e., it is statically unknown). We denote by $\mu\text{JS}^{\circledast}$ the $\mu$JS language plus $\circledast$. Let us show how we want to use $\circledast$ in the CFG generation by means of the examples shown in Fig. 7a and Fig. 7b, corresponding to the CFG of **if**($\circledast$){a:=a+1}**else**{b:=b+1}; and **while**($\circledast$){a:=a+1};, respectively. When $\circledast$ occurs in a program, the CFG generator labels both the edges exiting from its program point with **true**. Let us focus on the **if** case. The static analysis algorithm on CFG (namely Alg. 1) must take into account both **if**-branches, emulating an abstract execution where the boolean guard is statically unknown. Similarly, in the **while** case, both the **true** and **false** branches of the **while** loop are labeled with **true**. In this way, we emulate a **while** loop where the boolean guard is statically unknown, i.e., the body must be executed an unbounded number of times.

[3]Since concatenation is distributive w.r.t. $\parallel$, the conversion algorithm always distributes, in this case. For instance, x=(1; $\parallel$ 2;) is converted to (x=1;) $\parallel$ (x=2;).

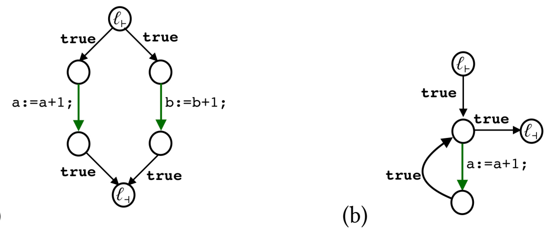


**Figure 7: (a) Examples of** CFG **generation with** $\circledast$**.**

We abuse notation denoting by Cfg the control-flow graph generator for $\mu$JS$^\circledast$ programs. At this point, we have all the ingredients to generated a $\mu$JS$^\circledast$ program from a regular expression. In particular, we inductively define on the structure of regexes the function $\wr \cdot \int : RE \longrightarrow \mu\text{JS}^\circledast$ that, given $\mathsf{r} \in RE$, translates r to a $\mu$JS$^\circledast$.[4]

$$\wr \mathsf{d} \int = \begin{cases} tocode(\mathcal{S}(\mathsf{d})) & \text{if } \mathsf{d} \in \Sigma_{\mathtt{pStm}} \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$$\wr \mathsf{r}_1\mathsf{r}_2 \int = \wr \mathsf{r}_1 \int \ \wr \mathsf{r}_2 \int$$

$$\wr \mathsf{r}_1 || \mathsf{r}_2 \int = \mathbf{if}\ (\circledast)\ \{\wr \mathsf{r}_1 \int \in \mu\text{JS}^\circledast\ ?\ \wr\ \mathsf{r}_1\ \int\ : \mathbf{skip}\}$$
$$\mathbf{else}\ \{\wr \mathsf{r}_2 \int \in \mu\text{JS}^\circledast\ ?\ \wr\ \mathsf{r}_2\ \int\ : \mathbf{skip}\}$$

$$\wr (\mathsf{r})^* \int = \mathbf{while}\ (\circledast)\ \{\wr \mathsf{r} \int \in \mu\text{JS}^\circledast\ ?\ \wr\ \mathsf{r}\ \int\ : \mathbf{skip}\}$$

In the base case (first line), we check if d is a partial statement, namely if $\mathsf{d} \in \Sigma_{\mathtt{pStm}}$. If so, it is returned as code (abusing notation of *tocode*), otherwise **skip** statement is returned. In the case of $\wr \mathsf{r}_1\mathsf{r}_2 \int$, the function concatenates the two programs inductively generated. In the case of $\wr \mathsf{r}_1 || \mathsf{r}_2 \int$, we need to emulate the non-deterministic execution of both the operands. Here comes to play $\circledast$, previously introduced. In particular, we return an `if` statement where the `if`-true body is replaced with $\wr \mathsf{r}_1 \int$ if it is executable, **skip** otherwise, and the `if`-false body is replaced with $\wr \mathsf{r}_2 \int$, if it is executable, **skip** otherwise. The boolean guard of the `if` statement is $\circledast$. It is worth noting that we need to check the executability of $\wr \mathsf{r}_1 \int$ and $\wr \mathsf{r}_2 \int$, since the **true** and **false** bodies must be $\mu$JS$^\circledast$ executable. We treat in a similar way the case of $\wr (\mathsf{r})^* \int$: in order to guarantee soundness, the $\mu$JS$^\circledast$ program $\wr \mathsf{r} \int$ must be executed an undefined number of times, hence, we build a **while** loop program, where the guard is $\circledast$.

The code synthesis from the regular expression $\mathsf{r}_{\mathsf{ds}}$ is the $\mu$JS$^\circledast$ program reported in Fig. 8.

The last step consists of generating a CFG on which we can recursively call our abstract interpreter. Hence, we call Cfg on the synthesized code, namely Cfg($\wr \mathsf{r} \int$). In our running example, the CFG corresponding to the program reported in Fig. 8 is $\mathsf{G}_{\mathsf{ds}}$ = Cfg($\wr \mathsf{r}_{\mathsf{ds}} \int$) reported in Fig. 9, where the labels of consecutive **true** edges are omitted. Note that the CFG of **while**`(x;` corresponds to the CFG of **skip** (right-most path in Fig. 9).

Putting all the sub-procedures together, we can define the produce `CFGGen`, that takes as input an automaton $A$ over $\Sigma_{\mathtt{pStm}}$ and generates a CFG , as `CFGGen`$(A) \triangleq$ Cfg($\wr$`Regex`$(A) \int$).

Finally, we have to prove soundness, proving that the output CFG contains the computation of all the executable strings that are

```
if (⊛) {
   if (⊛) {
      if (⊛) {
         x := x + 1
      } else {
         skip
      }
   } else {
      while (x > 5) {
         x := x + 1; y := x
      }
   }
} else {
   x := x + 1;
   while (⊛) {
      y := 10; x := x + 1
   }
}
```

**Figure 8:** $\mu$**JS**$^\circledast$ **program of** $\wr \mathsf{r}_{\mathsf{ds}} \int$**.**

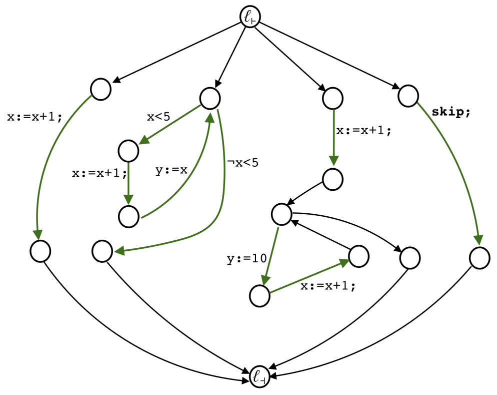


**Figure 9:** CFG $\mathsf{G}_{\mathsf{ds}}$ **generated by** `CFGGen` **module**

in the starting FA. In particular, next lemma shows that the CFG generated by Cfg($\wr \mathsf{r} \int$) contains all the computations of executable strings recognized by r.

Lemma 4.2. *Given* $\mathsf{r} \in RE$*, let* $\mathsf{G}_\mathsf{r} \triangleq$ Cfg($\wr \mathsf{r} \int$)*, then* $\forall \delta \in \mathscr{L}(\mathsf{r})$, $\forall \mathbb{m}^\# \in \mathbb{M}^\#.\ \exists \Pi \subseteq Paths(\mathsf{G}_\mathsf{r})$ *s.t.*

$$[\![ tocode(\mathcal{S}(\delta)) ]\!]^\# \mathbb{m}^\# \subseteq \bigcup_{\pi \in \Pi} [\![ \pi ]\!]^\# \mathbb{m}^\#$$

Finally, next theorem tells us that any executable string collected by the analysis is kept in the final generated CFG.

Theorem 4.3. *Let* $\mathsf{s} \in \mathrm{SExp}$*, let* $A_\mathsf{s}$ *be the FA recognizing the strings associated with* s*, then* $\forall \sigma \in \mathscr{L}(A_\mathsf{s}) \Cap \mu JS, \forall \mathbb{m}^\# \in \mathbb{M}^\#.\ \exists \Pi \subseteq$ *Paths*$(\mathsf{G}_\mathsf{s})$, $\mathsf{G}_\mathsf{s} \triangleq$ `CFGGen(StmSyn(`$A_\mathsf{s}$`))`. $[\![\sigma]\!]^\# \mathbb{m}^\# \subseteq \bigcup_{\pi \in \Pi} [\![\pi]\!]^\# \mathbb{m}^\#$.

## 5 EVALUATING THE ANALYZER

We have implemented the $\mu$JS static analyzer (available at https://github.com/SPY-Lab/mujs-analyzer) described in this paper, testing it on some significant **eval** programs in order to highlight the strengths and the weaknesses of the presented analyzer. In this section, we report the most significant cases in order to evaluate our approach. Moreover, we are currently integrating our approach upon TAJS static analyzer [23]. The proposed prototype shows that it is possible to design and implement an efficient sound-by-construction static analyzer based on abstract interpretation for self modifying code. In order to measure quality and precision of our abstract interpreter we tackle the following questions:

[4]We denote by $b$ ? `tt` : `ff` the inline conditional construct, namely if $b$ is true do `tt`, `ff` otherwise.

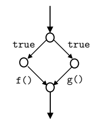
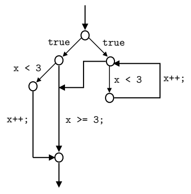
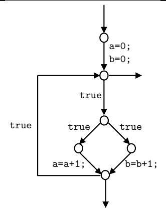
| Code fragment | Exe$^{\#}$ output |
|---|---|
| `str = "x=";`<br>`if (B)`<br>`  str = str + "f";`<br>`else`<br>`  str = str + "g";`<br>`eval(str + "();");` |  |
| `if (B)`<br>`  str = "if";`<br>`else`<br>`  str = "while";`<br>`str = str + "(x<3){x++;}"`<br>`eval(str);` |  |
| `str = "a=0;b=0;";`<br>`while (i++ < 100)`<br>`  if (B)`<br>`    str = str + "a++;";`<br>`  else`<br>`    str = str + "b++;";`<br>`eval(str);` |  a=0;<br>b=0;<br>true<br>true<br>true<br>true<br>a=a+1;<br>b=b+1; |

**Table 1**

**Q1:** Does the analyzer handle efficiently string-to-code statements (**eval**), even in presence of join points?
**Q2:** Does the analyzer handle nested calls to **eval**?

In order to answer to **Q1** and **Q2**, we evaluate the precision of our approach discussing, in the next sections, several **eval** usages inspired by real-world JavaScript applications. Finally, we conclude the evaluation by comparing our analyzer with TAJS [22, 23](version 0.9-8).

**eval** *of dynamic-generated string (Q1).* As observed before, the proposed architecture allows the analyzer to handle non-standard uses of **eval**, where the **eval** input string is dynamically manipulated. In the following, we describe three significant witnesses, allowing us to discuss about the precision of the analyzer.

Consider the first row of Tab. 1, supposing that the boolean guard B is unknown; hence both the branches must be taken into account, implying that the statements executed by **eval** may be either x=f() or x=g(). We approximate the code potentially executed by **eval** with the CFG reported in the first row, second column, in Tab. 1 (Exe$^{\#}$ output). Concerning precision, the synthesized CFG is precise since it precisely contains the two possible executions.

Consider a more challenging example, provided in the second row of Tab. 1. The boolean value of the guard is unknown, hence **eval** may execute either an **if** or a **while** statement. In this case, the code that will be potentially executed is not a simple combination of syntactic language structures. Hence, we believe this is an harder case to tackle for existing analysis tools. The approximation of the potentially executed code is reported in the second row. As before, the generated CFG is precise since it contains the two possible programs to execute.

Finally, in the last row of Tab. 1, the **eval** input string is built after a **while** statement *join point*. In this case, we also need to approximate the **while** loop execution, in order to avoid divergence. The number of loop iterations is unknown due to the unknown

```
str = "x=5";
while (i++ < 3)
 str += "5";
eval(str + ";");
```

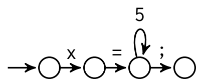


**Figure 10:** $A_{\mathsf{str}}$ **s.t.** $\mathscr{L}(A_{\mathsf{str}}) = \{\mathsf{x} = 5^n; \mid n > 0\}$

value of i before the loop. Hence, we need to apply a widening operator to ensure termination (Sect. 3.1). In the example we use the widening operator $\nabla^5_{\text{DFA}}$, allowing us to over-approximate the value of str by the regex a=0;b=0;(a++;||b++;)$^*$. It is possible to tune string approximation precision, and therefore to obtain different code approximations, by changing the widening operator used in the analysis. The corresponding CFG , over-approximating the code executed by **eval**, is shown in the last row. In this case, the CFG generation process adds further imprecision due to both the widening (generating cyles in the FA) and the way a CFG is generated starting from a star regex.

*Nested eval calls (Q2).* As explain in Sect. 3.3, the soundness and termination of our approach is guaranteed by nested call widening. Note that different results can be obtained from the analysis by tuning this parameter. In order to show how the analysis behaves in these situations, let us consider two significant examples: the first example is a terminating sequence of nested **eval** calls, while the second one is an infinite one. Consider the fragment below.

```
a=0;
str = "a++;if(a < 3){eval(\"a++;\" + str);}";
eval(str);
```

As long as a is less than 3, the program concatenates "a++;" with str, while, when a becomes greater then or equal to 3, the **eval** call returns, closing the sequence of nested calls. Clearly, the analysis result depends on the value of the nested call widening: if it is greater than or equal to 3, no loss of precision occurs during the analysis, handling precisely and efficiently the whole sequence of nested **eval** calls. Otherwise, the analysis gives up, returning the $\top$ abstract state (i.e., all the possible variables evaluated to $\top$) as explained in Sect. 3.3. In this way, while preserving soundness, the analysis may continue on the code after the **eval** call causing the nested call sequence, still able to get significant information about the program. Indeed, the $\top$ abstract string value is modeled by the FA recognizing $\Sigma^*$, making the analyzer able to trace string manipulations also of unknown (set to $\top$) variables. This is an important plus value of our analyzer, since most of the existing static analysis tools simply stuck the execution when a non-handled case occurs, returning no useful analysis information. Next code fragment shows an example of non-terminating sequence of nested **eval** calls. In this case, independently from the choice of the nested call widening, the static analyzer has to give up because the program diverges.

```
a=0; str = "a++;"
str = str + "if(a<3){str = \"a++\" + str;} eval(str);";
eval(str);
```

In order to be sound, a $\top$ abstract state is returned. Some techniques to detect as precisely as possible the presence of infinite nested **eval** call sequences can be studied and involved into the analyzer. This would define a smart widening technique for approximating nested **eval** calls for tuning the precision of the analysis in these situations, and it surely deserves further investigation.

## 5.1 Limitations

As shown in the previous sections, the proposed abstract interpreter is able to precisely answer about several **eval** patterns, even in presence of join points. Anyway, for some cases, even our abstract interpreter is not able to derive a CFG that over-approximates the **eval** input string. Consider the fragment reported in Fig. 10a, assuming that the value of i is unknown. Moreover, consider to apply $\nabla^2_{\text{DFA}}$ in the **while**-loop. In Fig. 10b, we report the FA abstracting the **eval** input, where the cycle in the FA is due to the application of the widening $\nabla^2_{\text{DFA}}$ to ensure termination. In this case, our analyzer can not return a CFG that over-approximates the code that may be concretely executed: the hypothetical CFG could be infinite since it should consider any possible assignment to x of any possible number formed by sequences of 5 (i.e., x=5;, x=55;, x=555;...). In general, our analyzer fails to construct a CFG that approximates the code that may be concretely executed when the cycles in the FA abstracting the input value of **eval** do not repeat valid statements, as in the example. In order to preserve soundness, when an **eval** statement occurs, our analyzer checks whether the input FA contains cycles that do not repeat a valid statement; if so, top abstract state is returned.

## 5.2 Comparison with TAJS

In [22], the authors introduce an automatic code rewriting technique removing **eval** constructs in JavaScript applications, showing that, in some cases, **eval** can be replaced by an equivalent JavaScript code without **eval**. This work has been inspired by [30] showing that **eval** is widely used. In particular, the authors integrate a refactoring of the calls to **eval** into TAJS. It performs inter-procedural data-flow analysis capturing whether **eval** input expression evaluate to constant values. If so, **eval** call can be replaced with an **eval**-free alternative code. It is clear that code refactoring is possible only when the abstract analysis recognizes that the arguments of **eval** are constants. Moreover, they handle the presence of nested **eval** by fixing a maximal degree of nesting, but in practice they set this degree to 1, since, as they claim, it is not often encountered in practice. The solution we propose allows us to go beyond constant values and refactor code also when the arguments of **eval** are not constants. We have identified three particular classes of **eval** programs depending on some features of the analyzed program which allow us to underline the differences between TAJS and our prototype. We report three significant examples in Tab. 2, where we summarize the comparison with TAJS. The first class of tests consists in programs where the string variables collect only one value during execution, i.e., they are constant strings. A witness of this class of programs is provided in the first row of Tab. 2, where the string value contained in y is constant. In this case, both, TAJS and our analyzer, are precise since no loss of information occurs during both the analyses. By using the value of y as input of **eval**, we obtain exactly the statement x=x+1; since Exe$^\#$, in this case, behaves as the identity function. TAJS performs the *uneval* transformation and executes the same statement.

The second class of tests consists in programs where there are no constant strings, namely strings whose value before **eval** is not precisely known and it is approximated by a set of potential string values. An example of this class is reported in the second row of

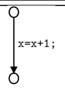
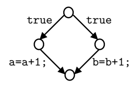
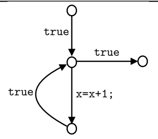
| P | TAJS result | Exe$^\#(A_\text{y})$ result |
|---|---|---|
| `y="x=x+1;";`<br>**`eval`**`(y);` | `x=x+1;` |  |
| **`if`**` (x > 0)`<br>` y="a=a+1;";`<br>**`else`**<br>` y="b=b+1;";`<br>**`eval`**`(y);` | Analysis Limitation Exception |  |
| `y= "";`<br>**`while`**` (x < 3) {`<br>` y = y + "x=x+1;";`<br>` x=x+1;`<br>`}`<br>**`eval`**`(y);` | Analysis Limitation Exception |  |

Table 2: Comparison with TAJS

Tab. 2, which is a simplification of Ex. 3.1. In this case, since we do not have any information about x, we must consider both the branches, meaning that before **eval** we only know that y is one value between "a=a+1" and "b=b+1". If we analyze this program in TAJS, the value of y before the **eval** call is identified as a string, since TAJS does not perform a collecting semantics, and when it loses the constant information it loses the whole value, leading to an exception in TAJS analysis when **eval** is met. On the other hand, our analyzer keeps the least upper bound between the stores computed in each branch, obtaining the abstract value for y modeled by the FA $A_\text{y}$ recognizing the language {a=a+1;, b=b+1;}. Afterwards, our analyzer returns and analyzes the sound approximation of the program passed to **eval** reported in the second row.

In the last class of examples, the string that will be executed is dynamically built at run-time. In the example provided in Tab. 2, the dynamically generated string is x=x+1;(x=x+1;)$^*$. In this case, as it happened before, TAJS loses the value of y and can only identify y as a string. This means that, again, **eval** makes the analysis stuck, throwing an exception. On the other hand, our analyzer performs a sound over-approximation of the set of values computed in y. In particular, the analysis, in order to guarantee termination, and therefore decidability, computes widening instead of least upper bound between FA, inside the loop. This clearly introduces imprecision, since it makes us lose the control on the number of iterations. In particular, applying $\nabla^3_{\text{DFA}}$, we compute a FA $A_\text{y}$ strictly containing the concrete set of possible string values, recognizing the regex x=x+1;(x=x+1;)$^*$. The presence of possible infinite sequences of x=x+1; is due to the over-approximation induced by the use of widening operator on FA. Nevertheless, the widening parameter can be tuned in order to get the desired precision degree of the analysis: The higher is the parameter, the more precise and costly the analysis is. The CFG extracted from $A_\text{y}$ is reported in the third row.

## 6 RELATED WORK AND CONCLUSIONS

The analysis of strings is nowadays a common practice in program analysis due to the widespread use of dynamic languages [10, 16, 25, 27, 32, 36]. The use of symbolic objects in abstract domains is also not new (see [14, 20, 33]) and some works explicitly use transducers for string analysis in script sanitization [21, 36], all recognizing that specifying the analysis in terms of abstract interpretation makes it suitable to potential combinations with other analyses, providing a better potential in tuning accuracy and costs. None of these works use string analysis for analyzing string-to-code statements. We already introduced TAJS [22], but other JavaScript static analyzers have been developed, such as JSAI [24] and SAFE [29]. They look for a flexible, configurable and tunable tool focusing on context-sensitiveness, heap-sensitiveness [24] and loop-sensitiveness [29]. Anyway they do not explicitly mention solutions to analyze string-to-code statements. TamiFlex [7] synthesizes a program at any `eval` call by considering the code that has been executed during some (dynamically) observed execution traces. The static analysis can then proceed with the so obtained code without `eval`. It is sound only w.r.t. the considered execution traces, producing a warning otherwise. Static analysis for a PHP core (ignoring `eval`-like primitives) has been developed in [6]. Static *taint analysis* keeping track of values derived from user inputs has been developed for self-modifying code by partial derivation of the CFG [34]. The approach is limited to taint analysis, e.g., for limiting code-injection attacks. Staged information flow for JavaScript in [11] with *holes* provides a conditional (a la abduction analysis in [19]) static analysis of dynamically evaluated code. Symbolic execution-based static analyses have been developed for scripting languages, including string-to-code statements, paying the introduction of false negatives [35]. We are not aware of effective general purpose sound static analyses handling self-modifying code for dynamic languages. On the contrary, a huge effort was devoted to bring static type inference to object-oriented dynamic languages (e.g., [1] for Ruby) but with a different perspective: *Bring into dynamic languages the benefits of static ones.* Our approach is different: *Bring into static analysis the possibility of handling dynamically mutating code.* A similar approach is in [2]. The idea is to extracting a code representation which is descriptive enough to include most code mutations by a dynamic analysis, and then reform analysis on a linearization of this code. On the semantics side, since the pioneering work on certifying self-modifying code in [9], the approach to self-modifying code consists in treating instructions as regular mutable data structures, and to incorporate a logic dealing with code mutation within a la Hoare logics for program verification.

This paper attacks an extremely hard problem in static program analysis: Analyzing dynamically mutating code in a meaningful and sound way. This provides the very first proof of concept in sound static analysis for self-modifying code based on bounded reflection for a high-level scripting language. Our main contribution is in proposing an innovative approach for designing sound static analyzers for string-to-code statements. The main idea is to analyze strings by approximating them as FA. When `eval` is met, the FA modeling its input is analyzed in order to approximate its *executable* sub-language, namely the sub-language of all the executable statements at that program point. The executable sub-language is then used for building a CFG whose semantics soundly approximate the semantics of what is concretely executed by `eval`. In this way we can recursively call the same abstract interpreter on the synthesized code. Once the recursive call returns we continue the standard analysis. The approach we propose is, in this sense, a truly *dynamic static analyzer*, keeping the analysis going even when code is dynamically built. Finally, we are currently implementing a prototype of this abstract interpreter for JavaScript, built upon TAJS, in order to prove that such a sound abstract interpreter for dynamic languages can be actually built and tested.

## REFERENCES

[1] An, J. D., Chaudhuri, A., Foster, J. S., and Hicks, M. Dynamic inference of static types for Ruby. In *POPL'11* (2011).
[2] Anckaert, B., Madou, M., and Bosschere, K. D. A model for self-modifying code. In *Information Hiding* (2006).
[3] Arceri, V., and Maffeis, S. Abstract domains for type juggling. *Electr. Notes Theor. Comput. Sci. 331* (2017).
[4] Arceri, V., and Mastroeni, I. Static program analysis for string manipulation languages. In *VPT'19* (2019).
[5] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Gros, C., Kamsky, A., McPeak, S., and Engler, D. R. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM 53*, 2.
[6] Biggar, P., and Gregg, D. Static analysis of dynamic scripting languages. Technical report, Department of Computer Science, Trinity College Dublin, 2009.
[7] Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., and Mezini, M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE'11* (2011).
[8] Brzozowski, J. A. Derivatives of regular expressions. *J. ACM 11*, 4 (1964).
[9] Cai, H., Shao, Z., and Vaynberg, A. Certified self-modifying code. In *PLDI* (2007).
[10] Christensen, A. S., Møller, A., and Schwartzbach, M. I. Precise analysis of string expressions. In *SAS'03* (2003).
[11] Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavaScript. In *PLDI* (2009).
[12] Cousot, P. Types as abstract interpretations (invited paper). In *POPL'97* (1997).
[13] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77* (1977).
[14] Cousot, P., and Cousot, R. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA'95* (1995).
[15] Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *POPL'78* (1978).
[16] Doh, K., Kim, H., and Schmidt, D. A. Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *SAS'09* (2009).
[17] Drape, S., Thomborson, C., and Majumdar, A. Specifying imperative data obfuscations. In *ISC'07* (2007).
[18] D'Silva, V. Widening for automata. Diploma Thesis, Universitat Zurich, 2006.
[19] Giacobazzi, R. Abductive analysis of modular logic programs. *J. Log. Comput. 8*, 4 (1998).
[20] Heintze, N., and Jaffar, J. Set constraints and set-based analysis. In *PPCP'94* (1994).
[21] Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., and Veanes, M. Fast and precise sanitizer analysis with BEK. In *USENIX'11* (2011).
[22] Jensen, S. H., Jonsson, P. A., and Møller, A. Remedying the eval that men do. In *ISSTA'12* (2012).
[23] Jensen, S. H., Møller, A., and Thiemann, P. Type Analysis for JavaScript. In *SAS'09* (2009).
[24] Kashyap, V., Dewey, K., Kuefner, E. A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., and Hardekopf, B. JSAI: a static analysis platform for javascript. In *FSE'14* (2014).
[25] Kim, H., Doh, K., and Schmidt, D. A. Static validation of dynamically generated HTML documents based on abstract parsing and semantic processing. In *SAS'13* (2013).
[26] Mavrogiannopoulos, N., Kisserli, N., and Preneel, B. A taxonomy of self-modifying code for obfuscation. *Computers & Security 30*, 8 (2011), 679–691.
[27] Minamide, Y. Static approximation of dynamically generated web pages. In *WWW'05* (2005).
[28] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis.* Springer, 1999.
[29] Park, C., and Ryu, S. Scalable and precise static analysis of javascript applications via loop-sensitivity. In *ECOOP'15* (2015).
[30] Richards, G., Hammer, C., Burg, B., and Vitek, J. The eval that men do - A large-scale study of the use of eval in javascript applications. In *ECOOP'11* (2011).
[31] Seidl, H., Wilhelm, R., and Hack, S. *Compiler Design - Analysis and Transformation.* Springer, 2012.
[32] Thiemann, P. Grammar-based analysis of string expressions. In *TLDI'05* (2005).
[33] Venet, A. Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Program. 35*, 2 (1999), 223–248.
[34] Wang, X., Jhi, Y., Zhu, S., and Liu, P. Still: Exploit code detection via static taint and initialization analyses. In *ACSAC* (2008).
[35] Xie, Y., and Aiken, A. Static detection of security vulnerabilities in scripting languages. In *USENIX '06* (2006).
[36] Yu, F., Alkhalaf, M., and Bultan, T. Patching vulnerabilities with sanitization synthesis. In *ICSE'11* (2011).