Improving Dynamic Code Analysis by Code Abstraction

Isabella Mastroeni
Department of Computer Science, University of Verona (Italy)
isabella.mastroeni@univr.it

Vincenzo Arceri
Department of Environmental Sciences, Informatics and Statistics,
Ca' Foscari University of Venice (Italy)
vincenzo.arceri@unive.it

In this paper, our aim is to propose a model for code abstraction, based on abstract interpretation, allowing us to improve the precision of a recently proposed static analysis by abstract interpretation of dynamic languages. The problem we tackle here is that the analysis may add some spurious code to the string-to-execute abstract value and this code may need some abstract representations in order to make it analyzable. This is precisely what we propose here, where we drive the code abstraction by the analysis we have to perform.

1 Introduction

The possibility of dynamically building code instructions as the result of text manipulation is a key aspect in dynamic programming languages. In this scenario, programs can turn text, which can be built at run-time, into executable code [25]. These features are often used in code protection and tamper-resistant applications, employing camouflage for escaping attack or detection [21], in malware, in mobile code, in web servers, in code compression, and in code optimization, e.g., 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. It is clear that more advanced and secure functionalities based on string-to-code statements could be permitted if we 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 later 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 is becoming an increasingly important problem to be tackled by the research community and by practitioners. Hence, it is not always possible to simply ignore `eval` without accepting to lose the possibility of analyzing the rest of the program [3].

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

Alexei Lisitsa and Andrei Nemytykh (Eds.): 9th International Workshop
on Verification and Program Transformation (VPT 2021)
EPTCS 341, 2021, pp. 17–32, doi:10.4204/EPTCS.341.2

```
str = "x=5";
while (i < 3) {
 str = str + "5";
 i = i + 1;
}
str = str + ";"; eval(str);
```

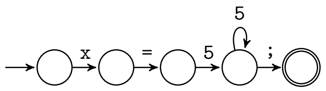


Figure 1: A s.t. $\mathscr{L}(A) = \{\texttt{x} = \texttt{5}^n\texttt{;} \mid n > 0\}$, where $\texttt{5}^n$ means 5 repeated n times.

The Context: Analyzing Dynamic Code. A major problem in presence of dynamic code generation is that static analysis becomes extremely hard if not impossible. This happens because program data structures, such as the control-flow graph and the system of recursive equations associated with the program in question, are themselves dynamically mutating objects. Recently [3], the problem of analyzing dynamic code has been tackled by *treating code as any other dynamic structure that can be statically analyzed by abstract interpretation, and to treat the abstract interpreter as any other program function that can be recursively called.* In particular, in [3], we provide a static analyzer architecture for a core dynamic language, containing non-removable eval statements, that still has some limitation in terms of precision but provides the necessary ground for studying more precise solutions to the problem. In particular,

- We have designed an automata-based string abstract domain [4] for analyzing string values during execution. Automata (FA) provide the perfect choice for abstracting strings that may be executed by eval since they allow us to over-approximate the set of possible values of string variables by keeping enough information for both analyzing properties of string variables that are never executed by an eval during computation and for extracting the potential executable sub-language.
- In order to statically analyze the code potentially executed by an eval, we have designed a systematic process for extracting from the (abstract) argument of eval (i.e., from the FA collection of its potential arguments) an over-approximation of executable code that this collection contains. Clearly, this approximation must keep a form that the analyzer can interpret.
- We designed a static analyzer for dynamic languages performing a recursive call of the interpreter on the (over-approximated) code that eval may execute.

The Problem: Improve Precision Analysis by Abstracting Code. This analysis provides a first step towards the analysis of dynamic languages but still has some important precision loss [3]. In particular, there are particular forms of FA (which occur when the string is dynamically generated by loops) avoiding the possibility of generating a control flow graph (CFG) able to approximate the code executed by an eval. For instance, when the FA accepts a language such as $\left\{ \texttt{x=(5)}^n\texttt{;} \,\middle|\, n > 0 \right\}$, the analysis in [3] cannot extract, from the FA, the CFG approximating the eval argument. In order to better explain the problem, consider the code in Fig. 1, where the value of i is statically unknown. In Fig. 1, we draw the automaton A representing the abstract value of str before the eval execution. The problem is that A has a cycle not involving a whole statement [3]. This situation makes the analyzer unable to build a CFG over-approximating the code potentially executed since, intuitively, such a CFG should be infinite. Indeed, only an infinite CFG could capture all the possible assignments described by the FA, namely all the assignments of any possible number formed only by 5 to the variable x (i.e., x=5;,x=55;,x=555;...). In order to make it possible to overcome this limitation, at least for a set of potential eval patterns, we propose to define a form of *abstract* CFG able to finitely represent a potential infinite set of CFGs, e.g., we look for a CFG representing $\texttt{x=5}^*$.

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

Figure 2: Syntax of Imp

Unfortunately, things are not so easy as it may seem, since this abstract code representation has to be built in such a way that the analyzer may still be able to interpret it.

Contribution. The main contributions for tackling the problem above are:

- We first define the notion of *abstract* CFG, based on the idea of making it possible to still perform a given analysis. The idea is to leave the control structure unchanged while approximating the edge labels (the statements to execute) to sets of labels, i.e., those sharing a fixed abstract property.
- We show how completeness of code abstraction w.r.t. the semantic observation models the possibility, for the static analyzer, of interpreting also the abstract code, and we show how we can make any code abstraction complete.
- We provide a systematic approach, based on the one proposed in [3], allowing us to analyze also the `eval` patterns described above, for which, instead, the analysis in [3] loses precision.

2 The Core Language: Imp

The language is quite standard (see Fig. 2[2]), and each statement is annotated with a label $\ell \in \mathit{Lab}$ (not part of the syntax) corresponding to the statement program point[3].

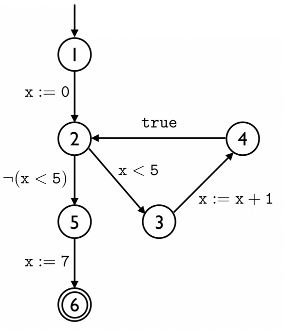
In order to analyze a program $\mathsf{P} \in \mathsf{Imp}$, we need to model it by building a corresponding control flow graph [27] (CFG for short), which embeds the control structure in the graph structure and leaves in the edges (or equivalently on the nodes) only the access to states, i.e., manipulation of the states (assignments) and guards. The approach we use is quite standard, and we follow [27] for the construction of the control flow graph. For technical details see [3], here we show the construction on the example in Fig. 3, where i denotes the node corresponding to the program point ℓ_i. Note that, by construction [3], the language of the CFG edge labels is an intermediate language slightly different from the Imp grammar. Edge labels correspond to a primitive statement (i.e., an assignment or `eval`) or a boolean guard, namely they form the language Ψ generated by the grammar $\mathtt{l} ::= \mathtt{x:=e} \mid \mathtt{b} \mid \mathtt{eval(s)}$.

[2] We use n to denote the semantic value corresponding to the syntactic symbol $\mathtt{n}$.

[3] We suppose that there exists a function that, taken a well-written program, can label it with a fresh label for each program point.

```
ℓ1x := 0;
ℓ2while (x<5)
  {ℓ3x := x + 1ℓ4};
ℓ5x:=7ℓ6
```

(a)



(b)

Figure 3: Example of CFG: (a) Fragment of code and (b) corresponding CFG.

Concrete Semantics. The concrete semantics of our language Imp is intuitive and it is fully reported in [2]. Since our aim is to analyze Imp programs by analyzing their CFGs, we focus here only on the interpretation of CFG's labels [27]. In particular, we have to specify the semantics associated with each possible edge of the CFG. In other words, we have to formalize how each statement transforms a current state, which is represented as a store, namely as an association between identifiers and values. It is well known that static program analysis works by computing (abstract) collecting semantics, namely, for each program point ℓ and for each variable $\mathtt{x}$, it computes the set of values that the variable $\mathtt{x}$ can have in any computation at the program point ℓ. Hence, we define (collecting) memories $\mathbbm{m}$, associating with each variable a *set* of values. The basic values of Imp are integers, booleans and strings, hence we define the set of memories as $\mathbb{M} \stackrel{\text{def}}{=} \mathsf{Var} \rightarrow (\wp(\mathbb{Z}) \cup \mathsf{Bool} \cup \wp(\Sigma^*))$, ranged over the meta-variable $\mathbbm{m}$, where $\mathsf{Bool} = \wp(\{\mathtt{false}, \mathtt{true}\})$. Let us denote by $\mathbb{V}$ this domain of collections of values $\wp(\mathbb{Z}) \cup \mathsf{Bool} \cup \wp(\Sigma^*)$. The update of memory $\mathbbm{m}$ for a variable $\mathtt{x}$ with set of values v is denoted $\mathbbm{m}[\mathtt{x}/v]$. The partial order $\sqsubseteq$ between memories is defined as $\mathbbm{m}_1 \sqsubseteq \mathbbm{m}_2 \Leftrightarrow \forall \mathtt{x} \in \mathsf{Id}.\, \mathbbm{m}_1(\mathtt{x}) \subseteq \mathbbm{m}_2(\mathtt{x})$. Finally, lub and glb of memories are computed point-wise, i.e., $\mathbbm{m}_1 \sqcup \mathbbm{m}_2 \stackrel{\text{def}}{=} \lambda \mathtt{x}.\, \mathbbm{m}_1(\mathtt{x}) \cup \mathbbm{m}_2(\mathtt{x})$ and $\mathbbm{m}_1 \sqcap \mathbbm{m}_2 \stackrel{\text{def}}{=} \lambda \mathtt{x}.\, \mathbbm{m}_1(\mathtt{x}) \cap \mathbbm{m}_2(\mathtt{x})$.
The collecting (input/output) semantics of statements $\mathsf{c} \in \Psi$ is defined as the function $[\![\mathsf{c}]\!] : \mathbb{M} \longrightarrow \mathbb{M}$. We denote by $(\!|\cdot|\!)$ the collecting semantics of expressions, defined as additive lift[4] to sets of memories of the standard expression semantics. We abuse notation by denoting as $[\![\cdot]\!]$ also its additive lift to sets of statements.

$$\begin{array}{rcl} [\![\mathtt{x}\texttt{:=}\mathsf{e}]\!]\mathbbm{m} & = & \mathbbm{m}[\mathtt{x}/(\!|\mathsf{e}|\!)\mathbbm{m}] \qquad [\![\mathsf{b}]\!]\mathbbm{m} = \mathbbm{m} \sqcap \bigsqcup \left\{ \mathbbm{m} \;\middle|\; (\!|\mathsf{b}|\!)\mathbbm{m} = \mathtt{true} \right\} \\ [\![\mathtt{eval}(\mathsf{s})]\!]\mathbbm{m} & = & [\![(\!|\mathsf{s}|\!)\mathbbm{m} \Cap \mathsf{Imp}]\!]\mathbbm{m} \end{array}$$

where $\Cap$ is the intersection in the set of Imp programs. By computing the traces of application of this transfer function, starting from any possible input memory, we precisely compute the maximal trace semantics [22].

Static Analysis on CFG: Semantic Abstraction. It is well known that when we perform static analysis on a CFG, we interpret, on the corresponding abstract domain, all the edges, and more specifically all the labels (in Ψ) [27]. This is also a quite standard approach, but we recall it here for fixing the notation used. We suppose to abstract values on the coalesced sum [2] of the Sign abstract domain for integers,

[4] Let $f : S \rightarrow S$ be a generic function, by *additive lift* we mean its extension to sets of elements, i.e., $\forall X \subseteq S$ we define $f(X) \stackrel{\text{def}}{=} \left\{ f(x) \;\middle|\; x \in S \right\}$. If $f : S \rightarrow \wp(S)$, then its lift to sets of memories is $f(X) \stackrel{\text{def}}{=} \bigcup \left\{ f(x) \;\middle|\; x \in S \right\}$

of the concrete domain for booleans and of the (deterministic) finite state automata abstract domain for strings [2][5]. Let us consider an abstraction $\rho \in uco(\mathbb{V})$[6] of the values manipulated by our language, we denote by $\mathbb{M}^\rho : \mathsf{Var} \longrightarrow \rho(\mathbb{V})$ the set of (collecting) memories, where sets of values are abstracted by ρ, ranged over $\mathbb{m}^\rho$. In the following, we abuse notation by applying ρ to memories in $\mathbb{M}$, simply by defining $\rho(\mathbb{m}) \in \mathbb{M}^\rho$ as $\rho(\mathbb{m}) : \mathtt{x} \in \mathsf{Var} \mapsto \rho(\mathbb{m}(\mathtt{x}))$[7]. In this way, we can see abstract memories as sets of concrete memories, and therefore as particular collecting memories, i.e., $\mathbb{M}^\rho \subseteq \mathbb{M}$. Finally, we can define the abstract edge effect $\llbracket\cdot\rrbracket^\rho$ [27] telling us how to abstractly interpret each edge of the CFG:

$$\begin{array}{rcl}\llbracket \mathtt{x:=e} \rrbracket^\rho \mathbb{m}^\rho & = & \mathbb{m}^\rho[\mathtt{x}/\rho(\llparenthesis \mathsf{e} \rrparenthesis^\rho \mathbb{m}^\rho)] \\ \llbracket \mathtt{eval}(\mathsf{s}) \rrbracket^\rho \mathbb{m}^\rho & = & \llbracket \llparenthesis \mathsf{s} \rrparenthesis^\rho \mathbb{m}^\rho \Cap \mathsf{Imp} \rrbracket^\rho \mathbb{m}^\rho \end{array} \qquad \llbracket \mathsf{b} \rrbracket^\rho \mathbb{m}^\rho = \mathbb{m}^\rho \sqcap \rho(\sqcup \left\{ \, \mathbb{m} \,\middle|\, \mathtt{true} \in \llparenthesis \mathsf{b} \rrparenthesis^\rho \mathbb{m}^\rho \, \right\})$$

where $\llparenthesis\cdot\rrparenthesis^\rho \overset{\text{def}}{=} \rho \circ \llparenthesis\cdot\rrparenthesis \circ \rho$.The semantics of a path in the CFG is the composition of the interpretation of each edge, and the interpretation of an edge is the interpretation, given above, of its label [27].

This is clearly, what happens when the CFG is not abstracted, namely when the edge labels are single statements. Finally, since we deal with potential abstract CFG, we have to say how we execute them, potentially on an abstract semantics. The idea is simple, since we move from executing single statements to executing sets of statements, we simply take as execution of the abstract CFG the additive lift of the single statements executions. Since the semantics is always additive[8], in order to guarantee that everything works, also the semantic abstraction ρ must be additive. Hence, in the following of the paper we always require ρ to be additive.

3 Semantic-driven Code Abstraction

In this section, we study how we can model a syntactic abstraction of the CFG and which is its *relation* with the semantic abstraction, i.e., the code analysis.

Modeling Code Abstraction. Following the standard approach for abstracting objects, we should abstract each CFG in a set of CFGs sharing an invariant property, i.e., an equivalence class of CFGs. In particular, since we aim at abstracting code (CFG) without changing the analysis performed on the code, we choose to abstract CFG by abstracting edge labels, and by leaving unchanged the control structure of the CFG. In other words, an abstract CFG, denoted CFG$^\#$, is a pair $\langle \mathit{Nodes}, \mathit{Edges}^\# \rangle$, where we leave the nodes unchanged, while the edge labels are abstracted to sets of labels. Formally, $\mathit{Edges}^\# \subseteq \mathit{Nodes} \times \wp(\Psi) \times \mathit{Nodes}$, where Ψ is the CFG label language.

Given $\eta \in uco(\wp(\Psi))$, $\mathtt{G}^\eta \overset{\text{def}}{=} \langle \mathit{Nodes}(\mathtt{G}), \mathit{Edges}^\eta(\mathtt{G}) \rangle$ is the CFG$^\#$ built from a CFG $\mathtt{G}$ in terms of η, where $\mathit{Edges}^\eta(\mathtt{G}) \subseteq \mathit{Nodes}(\mathtt{G}) \times \eta(\wp(\Psi)) \times \mathit{Nodes}(\mathtt{G})$.
As an example, consider the CFG in Fig. 3, in Fig. 4 we have the CFG$^\#$ where numerical expressions are abstracted by $\left\{ \, \mathtt{m} \,\middle|\, m \in \mathsf{Sign}(n) \, \right\}$[9] (where Sign is the well-known sign abstraction $\mathsf{Sign} \in uco(\wp(\mathbb{Z}))$ such as $\mathsf{Sign}(\wp(\mathbb{Z})) = \{\top, \mathbb{Z}^+, \mathbb{Z}^-, \{0\}, \varnothing\}$). For instance, $\mathtt{x:=x+1}$ is abstracted in $\mathtt{x:=x+}\mathbb{Z}^+$ where $\mathtt{x+}\mathbb{Z}^+ \overset{\text{def}}{=} \left\{ \, \mathtt{x+n} \,\middle|\, n \in \mathbb{Z}^+ \, \right\}$, being $\mathsf{Sign}(1) = \mathbb{Z}^+$.

[5] A string static analyzer using finite state automata abstract domain has been developed and it is available in [2].

[6] For the sake of simplicity here we abuse notation by considering a unique ρ which is indeed the coalesced sum of three abstractions, one for integers, one for booleans and one for strings.

[7] For the sake of simplicity of presentation and implementation, we have considered here non-relational abstractions of data, anyway we believe that it is possible to easy extend our work to relational abstractions.

[8] A function is said to be *additive* if it commutes with least upper bound.

[9] We use *n* to denote the semantic value corresponding to the syntactic symbol $\mathtt{n}$.

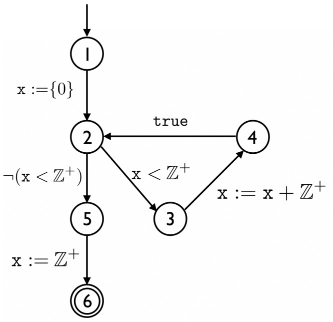


Figure 4: CFG abstracted by signs.

Abstracting Code vs Abstracting Semantics. As previously noted, we aim at characterizing code abstractions, for dynamically generated code, for which the given analysis works precisely. Formally, let us consider the following equation:

$$\forall \mathbb{m}^{\rho} \in \mathbb{M}^{\rho} \subseteq \mathbb{M}. \forall \varphi \in \Psi. [\![\eta(\varphi)]\!]\mathbb{m}^{\rho} = [\![\eta(\varphi)]\!]^{\rho}\mathbb{m}^{\rho} \tag{1}$$

If this equality does not hold it means that the abstract semantic interpretation $[\![\cdot]\!]^{\rho}$ merges predicates distinguished by η. Namely, when the program is observed by means of its (abstract) semantics the actual abstraction of predicates is not precisely η, but it is η affected in some way by $[\![\cdot]\!]^{\rho}$. By changing the point of view, we have that, in this case, the analysis cannot precisely interpret the abstract code, since η abstracts the code by distinguishing information that ρ cannot distinguish.

As an example, consider the sign domain above, when $\eta(\texttt{x:=5}) = \left\{\, \texttt{x:=n} \,\middle|\, 1 \le n \le 5 \,\right\}$ the equation does not hold since the concrete semantics of this set does not take *any* positive value for x. While, if $\eta(\texttt{x:=5}) = \left\{\, \texttt{x:=n} \,\middle|\, n \in \mathbb{Z}^+ \cup \{0\} \,\right\}$, then Eq. 1 holds since its concrete semantics is precisely the set of non-negative values. It is worth noting that Eq. 1 is a forward completeness [15] of the code abstraction w.r.t. the semantic interpretation, meaning that the semantic abstraction does not add imprecision to the code one.

In order to investigate the relation existing between the code abstraction η and the semantic abstraction ρ, we observe that, whenever we have a semantic abstraction ρ, we have a natural code abstraction induced by ρ. Namely, by only observing (abstract) information about the computation, we cannot distinguish statements with the same (abstract) semantics, independently from what any possible code abstraction does. For instance, if we analyze parity of program variables, we are unable to distinguish `x:=2` from `x:=4`, independently from how a potential code abstraction η is defined on `x:=2`. The first step consists in defining a code abstraction for expressions in terms of semantic one. Consider $\rho \in uco(\mathbb{V})$, we define $\widehat{\rho}(\mathsf{e})$ inductively on the expressions structure

$$\widehat{\rho}(\mathsf{a}) \quad : \begin{cases} \widehat{\rho}(\mathsf{a}_1 \mathsf{op}\, \mathsf{a}_2) \stackrel{\text{def}}{=} \left\{\, \mathsf{a}' \mathsf{op}\, \mathsf{a}'' \,\middle|\, \mathsf{a}' \in \widehat{\rho}(\mathsf{a}_1), \mathsf{a}'' \in \widehat{\rho}(\mathsf{a}_2) \,\right\} \stackrel{\text{def}}{=} \widehat{\rho}(\mathsf{a}_1)\mathsf{op}\,\widehat{\rho}(\mathsf{a}_2) \\ \widehat{\rho}(\mathtt{x}) \stackrel{\text{def}}{=} \mathtt{x}, \qquad \widehat{\rho}(\mathtt{n}) \stackrel{\text{def}}{=} \left\{\, \mathtt{m} \,\middle|\, m \in \rho(\{n\}) \,\right\} \end{cases}$$

$$\widehat{\rho}(\mathsf{b}) \quad : \begin{cases} \widehat{\rho}(\mathsf{b}_1 \mathsf{bop}\, \mathsf{b}_2) \stackrel{\text{def}}{=} \widehat{\rho}(\mathsf{b}_1)\mathsf{bop}\,\widehat{\rho}(\mathsf{b}_2), \qquad \widehat{\rho}(\neg \mathsf{b}) \stackrel{\text{def}}{=} \neg\widehat{\rho}(\mathsf{b}) \\ \widehat{\rho}(\mathtt{x}) \stackrel{\text{def}}{=} \mathtt{x}, \qquad \widehat{\rho}(\mathtt{true}) \stackrel{\text{def}}{=} \left\{\, \mathtt{t} \,\middle|\, t \in \rho(\mathtt{true}) \,\right\}, \qquad \widehat{\rho}(\mathtt{false}) \stackrel{\text{def}}{=} \left\{\, \mathtt{t} \,\middle|\, t \in \rho(\mathtt{false}) \,\right\} \end{cases}$$

$$\widehat{\rho}(\mathsf{s}) \quad : \begin{cases} \widehat{\rho}(\mathtt{concat}(\mathsf{s}_1,\mathsf{s}_2)) \stackrel{\text{def}}{=} \mathtt{concat}(\widehat{\rho}(\mathsf{s}_1),\widehat{\rho}(\mathsf{s}_2)), \\ \widehat{\rho}(\mathtt{substr}(\mathsf{s},\mathsf{a}_1,\mathsf{a}_2)) \stackrel{\text{def}}{=} \mathtt{substr}(\widehat{\rho}(\mathsf{s}),\widehat{\rho}(\mathsf{a}_1),\widehat{\rho}(\mathsf{a}_2)) \\ \widehat{\rho}(\mathtt{x}) \stackrel{\text{def}}{=} \mathtt{x}, \qquad \widehat{\rho}(\texttt{"}\sigma\texttt{"}) \stackrel{\text{def}}{=} \left\{\, \texttt{"}\delta\texttt{"} \,\middle|\, \delta \in \rho(\sigma) \,\right\} \end{cases}$$

At this point, we can characterize the CFG labels abstraction $\overline{\Upsilon}[\rho] : \wp(\Psi) \longrightarrow \wp(\Psi)$, as the additive lift of the function

$$\begin{array}{rl} \overline{\Upsilon}[\rho](\mathtt{x}\,\mathtt{:=}\mathsf{e}) & \overset{\text{def}}{=} x := \widehat{\rho}(\mathsf{e}) \overset{\text{def}}{=} \left\{\, x := \mathsf{e}' \,\middle|\, \mathsf{e}' \in \widehat{\rho}(\mathsf{e}) \,\right\} \\ \overline{\Upsilon}[\rho](\mathsf{b}) & \overset{\text{def}}{=} \widehat{\rho}(\mathsf{b}) \qquad\qquad \overline{\Upsilon}[\rho](\mathtt{eval}(\mathsf{s})) \overset{\text{def}}{=} \mathtt{eval}(\widehat{\rho}(\mathsf{s})) \end{array}$$

where $\mathtt{eval}(\widehat{\rho}(\mathsf{s}))$ is treated as the implicit representation of all the statements that it can execute, namely it represents the (potentially infinite) set $\left\{\, \mathsf{c} \,\middle|\, \llbracket \mathsf{c} \rrbracket \mathbbm{m} \sqsubseteq \llbracket \llparenthesis \mathsf{s} \rrparenthesis^{\rho} \Cap \mathsf{Imp} \rrbracket^{\rho} \mathbbm{m} \,\right\}$.
The following result is immediate by construction.

Proposition 3.1 *Given $\rho \in uco(\mathbb{V})$, then $\overline{\Upsilon}[\rho] \in uco(\wp(\Psi))$ and it is additive.*

Finally, in order to show that this code abstraction can be used to force satisfiability of Eq. 1, we have first to characterize the meaning of interpreting an edge label abstracted by $\overline{\Upsilon}[\rho]$:

$$\begin{array}{rcl} \llbracket \mathtt{x}\,\mathtt{:=}\widehat{\rho}(\mathsf{e}) \rrbracket \mathbbm{m} & = & \bigsqcup \left\{\, \llbracket \mathtt{x}\,\mathtt{:=}\mathsf{e}' \rrbracket \mathbbm{m} \,\middle|\, \mathsf{e}' \in \widehat{\rho}(\mathsf{e}) \,\right\} \qquad\qquad \llbracket \widehat{\rho}(\mathsf{b}) \rrbracket \mathbbm{m} = \bigsqcup \left\{\, \llbracket \mathsf{b}' \rrbracket \mathbbm{m} \,\middle|\, \mathsf{b}' \in \widehat{\rho}(\mathsf{b}) \,\right\} \\ \llbracket \mathtt{eval}(\widehat{\rho}(\mathsf{s})) \rrbracket \mathbbm{m} & = & \bigsqcup \left\{\, \llbracket \mathsf{c} \rrbracket \mathbbm{m} \,\middle|\, \llbracket \mathsf{c} \rrbracket \mathbbm{m} \sqsubseteq \llbracket \llparenthesis \mathsf{s} \rrparenthesis^{\rho} \Cap \mathsf{Imp} \rrbracket^{\rho} \mathbbm{m} \,\right\} \end{array}$$

Then we have the following results

Lemma 3.2 *Given $\rho \in uco(\mathbb{V})$ additive, then $\forall \mathsf{e}.\, \forall \mathbbm{m} \in \mathbb{M}^{\rho}.\, \llparenthesis \widehat{\rho}(\mathsf{e}) \rrparenthesis \mathbbm{m} = \llparenthesis \mathsf{e} \rrparenthesis^{\rho} \mathbbm{m}$ (trivially implying $\mathsf{e}' \in \widehat{\rho}(\mathsf{e}) \Leftrightarrow \forall \mathbbm{m} \in \mathbb{M}^{\rho}.\, \llparenthesis \mathsf{e}' \rrparenthesis \mathbbm{m} \subseteq \llparenthesis \mathsf{e} \rrparenthesis^{\rho} \mathbbm{m}$) and $\forall \Phi \in \wp(\Psi).\, \forall \mathbbm{m} \in \mathbb{M}^{\rho}.\, \llbracket \overline{\Upsilon}[\rho](\Phi) \rrbracket \mathbbm{m} = \llbracket \Phi \rrbracket^{\rho} \mathbbm{m}$.*

PROOF. Let us prove first the property for expressions by induction on the syntactic structure of e.

$\mathsf{e} = \mathtt{n}$: $\llparenthesis \widehat{\rho}(\mathsf{e}) \rrparenthesis \mathbbm{m} = \llparenthesis \widehat{\rho}(\mathtt{n}) \rrparenthesis \mathbbm{m} \overset{\text{def}}{=} \rho(n)$, while $\llparenthesis \mathsf{e} \rrparenthesis^{\rho} \mathbbm{m} = \llparenthesis \mathtt{n} \rrparenthesis^{\rho} \mathbbm{m} = \rho(n)$ (where $\llparenthesis \mathtt{n} \rrparenthesis \mathbbm{m} = n$);

$\mathsf{e} = \mathtt{x}$: $\llparenthesis \widehat{\rho}(\mathsf{e}) \rrparenthesis \mathbbm{m} = \llparenthesis \widehat{\rho}(\mathtt{x}) \rrparenthesis \mathbbm{m} \overset{\text{def}}{=} \llparenthesis \mathtt{x} \rrparenthesis \mathbbm{m} = \mathbbm{m}(\mathtt{x})$, while $\llparenthesis \mathsf{e} \rrparenthesis^{\rho} \mathbbm{m} = \llparenthesis \mathtt{x} \rrparenthesis^{\rho} \mathbbm{m} = \rho(\mathbbm{m}(\mathtt{x})) = \mathbbm{m}(\mathtt{x})$ (since $\mathbbm{m} \in \mathbb{M}^{\rho}$);

$\mathsf{e} = \mathsf{e}_1\, \mathsf{op}\, \mathsf{e}_2$: Suppose op any arithmetic or boolean operator.
$\llparenthesis \widehat{\rho}(\mathsf{e}) \rrparenthesis \mathbbm{m} = \llparenthesis \widehat{\rho}(\mathsf{e}_1\, \mathsf{op}\, \mathsf{e}_2) \rrparenthesis \mathbbm{m} \overset{\text{def}}{=} \llparenthesis \widehat{\rho}(\mathsf{e}_1)\, \mathsf{op}\, \widehat{\rho}(\mathsf{e}_2) \rrparenthesis \mathbbm{m} = \llparenthesis \widehat{\rho}(\mathsf{e}_1) \rrparenthesis \mathbbm{m}\, \mathsf{op}\, \llparenthesis \widehat{\rho}(\mathsf{e}_2) \rrparenthesis \mathbbm{m} = \llparenthesis \mathsf{e}_1 \rrparenthesis^{\rho} \mathbbm{m}\, \mathsf{op}\, \llparenthesis \mathsf{e}_2 \rrparenthesis^{\rho} \mathbbm{m}$ by inductive hypothesis. But this is precisely $\llparenthesis \mathsf{e}_1\, \mathsf{op}\, \mathsf{e}_2 \rrparenthesis^{\rho} \mathbbm{m}$ since op is computed on the semantics as additive lift to sets.

Analogously, we can prove all the other cases.

Now, let us prove the fact for CFG single edge labels, again by induction on the syntactic structure. Note that, being ρ additive then also $\llbracket \cdot \rrbracket^{\rho}$ is additive, being also the concrete semantics additive on sets of statements.

$$\begin{array}{rcl} \llbracket \overline{\Upsilon}[\rho](\mathtt{x}\,\mathtt{:=}\mathsf{e}) \rrbracket \mathbbm{m} & = & \llbracket \mathtt{x}\,\mathtt{:=}\widehat{\rho}(\mathsf{e}) \rrbracket \mathbbm{m} \\ & = & \bigsqcup \left\{\, \llbracket \mathtt{x}\,\mathtt{:=}\mathsf{e}' \rrbracket \mathbbm{m} \,\middle|\, \mathsf{e}' \in \widehat{\rho}(\mathsf{e}) \,\right\} \\ & = & \bigsqcup \left\{\, \mathbbm{m}[\mathtt{x}/\llparenthesis \mathsf{e}' \rrparenthesis \mathbbm{m}] \,\middle|\, \mathsf{e}' \in \widehat{\rho}(\mathsf{e}) \,\right\} \\ & = & \mathbbm{m}[\mathtt{x}/\bigcup \left\{\, \llparenthesis \mathsf{e}' \rrparenthesis \mathbbm{m} \,\middle|\, \mathsf{e}' \in \widehat{\rho}(\mathsf{e}) \,\right\}] \\ & = & \mathbbm{m}[\mathtt{x}/\bigcup \left\{\, \llparenthesis \mathsf{e}' \rrparenthesis \mathbbm{m} \,\middle|\, \llparenthesis \mathsf{e}' \rrparenthesis \mathbbm{m} \subseteq \llparenthesis \mathsf{e} \rrparenthesis^{\rho} \mathbbm{m} \,\right\}] \\ & = & \mathbbm{m}[\mathtt{x}/\llparenthesis \mathsf{e}' \rrparenthesis^{\rho} \mathbbm{m}] = \llbracket \mathtt{x}\,\mathtt{:=}\mathsf{e} \rrbracket^{\rho} \mathbbm{m} \end{array}$$

$$\begin{array}{rcl} \llbracket \overline{\Upsilon}[\rho](\mathsf{b}) \rrbracket \mathbbm{m} & = & \llbracket \widehat{\rho}(\mathsf{b}) \rrbracket \mathbbm{m} \\ & = & \bigsqcup \left\{\, \llbracket \mathsf{b}' \rrbracket \mathbbm{m} \,\middle|\, \mathsf{b}' \in \widehat{\rho}(\mathsf{b}) \,\right\} \\ & = & \bigsqcup \left\{\, \mathbbm{m} \sqcap \bigsqcup \left\{\, \mathbbm{m} \,\middle|\, \llparenthesis \mathsf{b}' \rrparenthesis \mathbbm{m} = \mathtt{true} \,\right\} \,\middle|\, \mathsf{b}' \in \widehat{\rho}(\mathsf{b}) \,\right\} \\ & = & \mathbbm{m} \sqcap \bigsqcup \left\{\, \mathbbm{m} \,\middle|\, \llparenthesis \mathsf{b}' \rrparenthesis \mathbbm{m} = \mathtt{true},\, \mathsf{b}' \in \widehat{\rho}(\mathsf{b}) \,\right\} \\ & = & \mathbbm{m} \sqcap \bigsqcup \left\{\, \mathbbm{m} \,\middle|\, \llparenthesis \mathsf{b}' \rrparenthesis \mathbbm{m} = \mathtt{true},\, \llparenthesis \mathsf{b}' \rrparenthesis \mathbbm{m} \subseteq \llparenthesis \mathsf{b} \rrparenthesis^{\rho} \mathbbm{m} \,\right\} \\ & = & \mathbbm{m} \sqcap \bigsqcup \left\{\, \mathbbm{m} \,\middle|\, \mathtt{true} \in \llparenthesis \mathsf{b} \rrparenthesis^{\rho} \mathbbm{m} \,\right\} = \llbracket \mathsf{b} \rrbracket^{\rho} \mathbbm{m} \end{array}$$

$$\begin{array}{rcl}
[\![\overline{\Upsilon}[\rho](\texttt{eval(s)})]\!]\mathbb{m} & = & [\![\texttt{eval}(\widehat{\rho}(\mathsf{s}))]\!]\mathbb{m} \\
 & = & \bigsqcup\left\{\, [\![\mathsf{c}]\!]\mathbb{m} \,\middle|\, [\![\mathsf{c}]\!]\mathbb{m} \sqsubseteq [\![(\!|\mathsf{s}|\!)^{\rho} \Cap \mathsf{Imp}]\!]^{\rho}\mathbb{m} \,\right\} \\
\text{By additivity of } [\![\cdot]\!]^{\rho} & = & [\![(\!|\mathsf{s}|\!)^{\rho} \Cap \mathsf{Imp}]\!]^{\rho}\mathbb{m} = [\![\texttt{eval(s)}]\!]^{\rho}\mathbb{m}
\end{array}$$

Finally, for each set of labels Φ, we have that $[\![\overline{\Upsilon}[\rho](\Phi)]\!]\mathbb{m} = \bigsqcup_{\varphi\in\Phi} [\![\overline{\Upsilon}[\rho](\varphi)]\!]\mathbb{m} = \bigsqcup_{\varphi\in\Phi} [\![\varphi]\!]^{\rho}\mathbb{m} = [\![\Phi]\!]^{\rho}\mathbb{m}$, since all the involved functions are additive by definition or by construction. □

Then we have that:

Theorem 3.3 *Let $\rho \in uco(\mathbb{V})$ additive, and $\eta \in uco(\wp(\Psi))$. Then $\overline{\eta}_{\uparrow} \overset{def}{=} \overline{\Upsilon}[\rho] \circ \eta$ satisfies Eq. 1.*

PROOF. It is worth noting that, we trivially have by abstraction that $\forall\varphi \in \Psi.\ [\![\eta_{\uparrow}(\varphi)]\!] \subseteq [\![\eta_{\uparrow}(\varphi)]\!]^{\rho}$. Let us prove the other implication: $\forall\varphi \in \Psi$

$$\begin{array}{rll}
[\![\eta_{\uparrow}(\varphi)]\!] & = [\![\overline{\Upsilon}[\rho] \circ \eta(\varphi)]\!] & \\
 & = [\![\overline{\Upsilon}[\rho] \circ \overline{\Upsilon}[\rho] \circ \eta(\varphi)]\!] & \text{[By properties of uco]} \\
 & = [\![\overline{\Upsilon}[\rho] \circ \eta(\varphi)]\!]^{\rho} & \text{[By Lemma. 3.2]} \\
 & = [\![\eta_{\uparrow}(\varphi)]\!]^{\rho} &
\end{array}$$

□

This result tells us that by taking a code abstraction more abstract than (or equal to) $\overline{\Upsilon}[\rho]$, we guarantee that the abstract interpretation ρ *can be* performed on the abstracted program (Eq. 1). We have so far proved that it is always possible to force Eq. 1, in order to make it possible to continue the analysis (observing ρ) also on the abstracted code. In the following we show how this framework can be integrated with the existing analysis of dynamic code [3] in order to improve its precision.

4 An Improved Dynamic Code Analysis

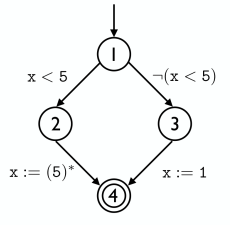
In this section we show how the constructive code abstraction characterization, provided in the previous section, can be used for representing the code approximation which soundly captures the potential code executed by a string-to-code statement. As we will show, without abstracting code, we cannot capture situations where the collecting semantics on strings generates sets of statements that cannot be represented by using the concrete syntax. Nevertheless, we must also observe that the analyzer cannot change dynamically with the generated code, hence the abstraction *must* be driven by the semantic property analyzed. This means that, without using the proposed framework, the analysis would surely be less precise in those situations where code abstraction becomes a necessity.

Let us summarize how we propose to exploit the framework:

- Consider a fixed semantic abstraction $\rho \in uco(\mathbb{V})$ and a corresponding static analyzer, designed in such a way that it can interpret also code abstracted by $\overline{\Upsilon}[\rho]$.
- Analyze the program, and when an `eval` is met, extract the language of its argument. If the language is infinite (under specific conditions that we will discuss) build the abstract CFG approximating it and extract the corresponding code abstraction η. In general, this code abstraction η is not more abstract than $\overline{\Upsilon}[\rho]$ (the code abstraction already embedded in the static analyzer, depending only on ρ);
- Build $\overline{\Upsilon}[\rho] \circ \eta$ in order to make also the generated code (approximated by the generated abstract CFG) analyzable by the static analysis for ρ.

```
ℓ1str := "x:=5"; ℓ2i := 0;
ℓ3while (i < N) {
 ℓ4str := str + "5";
 ℓ5i:=i+1;ℓ6
}
ℓ7str := "if(x<5){"  + str
  + "}else{x:=1};";
ℓ8eval(str)ℓ9
```

(a)



(b)

Figure 5: (a) Dynamically-generating code sample. (b) CFG associated to `str` labeled with abstract expressions.

Analyzing Dynamic Code. Let ρ be a static analysis performing in particular $\rho_S \in uco(\wp(\mathbb{S}))$ on strings, where $\mathbb{S} = \mathscr{K}^*$ denotes strings over a finite alphabet $\mathscr{K}$. Note that, our analyzer has to work on any (abstract) CFG that can be dynamically generated, hence it has to be designed with this purpose in mind. In particular, as we will show, we will generate only abstract CFGs with a code abstraction η complete w.r.t. ρ. This means, by construction, that η must be more abstract than $\overline{\Upsilon}[\rho]$, which means that each set of elements in η corresponds to a subset of the elements (abstract predicates) of $\overline{\Upsilon}[\rho]$. Hence, in order to guarantee to interpret predicates in any η complete, it is sufficient to design the analyzer soundly interpreting any abstract predicate in $\overline{\Upsilon}[\rho]$. For instance, $\overline{\Upsilon}[\mathsf{Sign}]$ is the abstraction containing all the predicates, involving integers, of the form `x:=S`, `x<S`, etc, with `S` $\in$ Sign, e.g., an abstract predicate is `x:=`$\mathbb{Z}^+$, and the analyzer for Sign should be able to interpret also such abstract predicates.
Let `x` be the input string parameter of an `eval` statement, we denote by $\mathscr{S}^{\rho_S}(\mathtt{x})$ the abstract value for `x` computed by the analysis on ρ_S. For example, suppose that the collection of values for the string `x` before the `eval` is $\{$`a:=0`$,$`a:=1`$\}$. By defining ρ_S as the k-bounded string set abstract domain [1], with $k = 2$, $\mathscr{S}^{\rho_S}(\mathtt{x}) = \{\mathtt{a:=0}, \mathtt{a:=1}\}$, while by using the prefix abstract domain $\overline{\mathscr{P}\mathscr{R}}$ [8], $\mathscr{S}^{\mathscr{P}\mathscr{R}}(\mathtt{x}) = \left\{ \mathtt{a:=s} \;\middle|\; s \in \mathbb{S} \right\}$. When the abstracted string and the abstraction is clear from the context, we simply denote this set by $\mathscr{S}$ and we assume (for the sake of simplicity) that any string in $\mathscr{S}$ is an executable language statement[10]. In the following, we abuse notation by denoting $\mathscr{S}$ also the automaton recognizing the language.
Consider for example, the program reported in Fig. 5a, a program building and manipulating the string `str` at run-time, which is, afterwards, interpreted as executable code, being the input parameter of the string-to-code statement `eval`. Since the value of `N` is unknown at compile-time, we cannot predict the precise number of iterations of the `while`-loop. In this case, a suitable string abstract analysis would approximate the value of `str`, before the `eval` execution, to an abstract value corresponding to an over-approximation of the possible values for `str`, which may be also, due to abstraction, an infinite set of strings, and therefore an infinite set of possible programs. For instance, in the example, if we abstract strings into the regular expression abstract domain [7] (or equivalently into the finite state automata abstract domain [2]), the value of `str` after the `while` loop will be the abstract value $\mathtt{x} := 5(5)^*$; corresponding to an infinite set of programs, i.e., `x:=5;`, `x:=55`, `x:=555;`.... In this case, the common practice for analyzing `eval` is simply to give up with the analysis, for example by halting the analysis throwing an exception [16] or forbidding its usage [17].

[10] Note that, this assumption corresponds to a decidable condition, hence it is possible to check it and to implement ad hoc solutions when it does not hold.

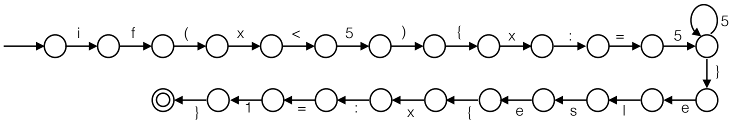


Figure 6: Finite state automaton corresponding to the abstract value of `str`.

Let $\rho_{\mathscr{C}\mathscr{S}}$ be the abstract domain for all the possible values (integers, strings and booleans) [3]. Note that, $\overline{\Upsilon}[\rho_{\mathscr{C}\mathscr{S}}]$ contains, for integers, predicates like the ones in the abstract CFG in Fig. 4.
The analysis $\rho_{\mathscr{C}\mathscr{S}}$ at point ℓ_3, due to widening[11] applied in the analysis of the while loop [2], abstracts the value of `str` in the infinite language $\left\{\ \texttt{x:=s} \,\middle|\, s \in (5)^+\ \right\}$ (namely `x` is assigned to any value represented by a finite sequence of 5). Hence, at point ℓ_8 the analysis abstracts `str` to the strings set $\mathscr{S}_{\texttt{str}} = \left\{\ \texttt{if(x<5)\{x:=s\}else\{x:=1\}} \,\middle|\, s \in (5)^+\ \right\}$ meaning that, the true-branch of the string that may be transformed by `eval` may be either `x:=5`, or `x:=55`, or `x:=555`, ... The automaton corresponding to the abstract value of `str` is reported in Fig. 6, and it denotes an infinite language, i.e., an infinite set of possible statements. Unfortunately, this is a problem for the analysis provided in [3], where the language containing all the possible strings would be returned, losing any precision.

Generating the Code: From Automata to CFG**s.** At this point, we have the (potentially infinite) language of the `eval` argument (and hence an automaton $\mathscr{S}$), and the goal is to generate a CFG modeling an over-approximation of the executable code contained in the language of the automaton $\mathscr{S}$. The idea is to generate a CFG from a language of strings, i.e., from an automaton, by performing a parsing on the paths of the automaton. Indeed, we have defined and implemented an algorithm[12], reported in Alg. 1, performing an abstract parser on automata that, given an automaton $\mathscr{S}$, returns the CFG $\mathscr{P}$ that over-approximates, for each $s \in \mathscr{S}$ (executable), the concrete execution of `eval`.

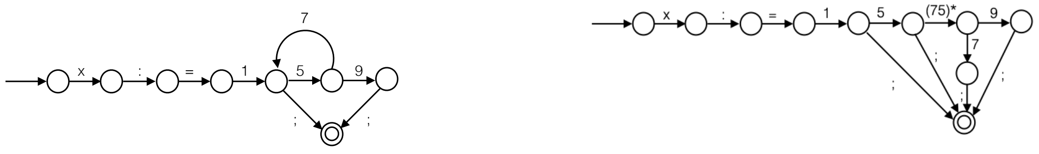
The idea of Alg. 1 is to perform a depth-first search on the automaton and, when a language statement is recognized, to generate an edge in the CFG. This phase is handled by lines 3-13 of Alg. 1, building the set of nodes *Nodes* and the set of edges *Edges* of the resulting CFG $\mathscr{P}$. The set W contains the states of the finite state automaton for which we still have to generate edges in the CFG and it is initialized, at line 2, with the initial state q_0. At this point, Alg. 1 looks for language statements readable from any path of the input automaton starting from a state q, taken from W, by means of the module ReduceStmts (line 5). In particular, ReduceStmts returns a set of triples $(q', \texttt{c}, q'')$, where each returned triple means that from $q' \in Q$ to $q'' \in Q$ a language statement `c` has been recognized. The set returned by ReduceStmts corresponds to the set of statements of $\mathscr{P}$ readable from the state q, hence they are added to *Edges*, substituting the reached states with the corresponding labels by means of the function lab (lines 7-8). At this point, we need to look for the statements that can be read from q'', hence, q'' is added to W in order to be eventually processed at the next iterations of the while loop at lines 3-13. When there are no more states of $\mathscr{S}$ to be processed, namely when W is empty, the CFG $\mathscr{P} = \langle \mathit{Nodes}, \mathit{Edges} \rangle$ is returned (line 14), with entry label $\mathsf{lab}(q_0)$ and exit labels the ones associated with the states in F.

Problems arise when the automaton contains cycles (namely, when the automaton denotes an infinite language). In this case, Alg. 1 first transforms, at line 1, the input automaton, over the alphabet $\mathscr{K}$, in

[11] Widening is a fix-point accelerator used in infinite domains with infinite ascending chains, namely where the semantic fix-point computation may diverge. In this case we use a widening on automata defined in [7]

[12] In the following, we only discuss the main parts of the algorithm for space limitations.

```
Algorithm 1:
  Input: $\mathscr{S} = (Q, \mathscr{K}, \delta, q_0, F)$
  Output: CFG $\mathscr{P}$ over-approximating executable strings of $\mathscr{S}$
$\mathscr{S}$ = ReduceCycles($\mathscr{S}$);
$Nodes \leftarrow \varnothing$; $Edges \leftarrow \varnothing$; $W \leftarrow \{q_0\}$; $visited \leftarrow \varnothing$;
while $W \neq \varnothing$ do
    select and remove $q$ from $W$;
    $stmts \leftarrow$ ReduceStmts($\mathscr{S}, q$);
    foreach $(q', \mathsf{c}, q'') \in stmts$ do
        $Nodes \leftarrow Nodes' \cup \{\mathsf{lab}(q'), \mathsf{lab}(q'')\}$;
        $Edges \leftarrow Edges \cup \{(\mathsf{lab}(q'), \mathsf{c}, \mathsf{lab}(q''))\}$;
        $visited \leftarrow visited \cup \{q'\}$;
        $W \leftarrow W \cup \{q''\}$;
        $W \leftarrow W \smallsetminus visited$;
    end
end
return $\mathscr{P} = \langle Nodes, Edges \rangle$;
```



(a) (b)

Figure 7: (a) Finite state automaton with cycle. (b) Result of ReduceCycles.

an automaton without cycles, over the alphabet $\mathscr{K} \cup \wp(\mathscr{K}^*)$, by means of the module ReduceCycles. Given an input automaton $\mathscr{S}$, we retrieve the cycles of $\mathscr{S}$ using the well-known Tarjan's algorithm [26] for identifying cycles. Then, for each detected cycle of $\mathscr{S}$, we check whether the string read by the cycle is a whole statement r or not. In the first case, we substitute the cycle of the string r in the automaton, i.e., r^*, with the automaton reading the string corresponding to the statement `while(true){ r }` over the alphabet $\mathscr{K}$. Otherwise, if the cycle does not read a whole statement, the idea is to collapse the cycle in a single transition, labeled with the regular expression corresponding to what is read in the cycle, i.e., denoting a set of string on $\mathscr{K}$ ($\wp(\mathscr{K}^*)$). Hence the resulting automaton is on the alphabet $\mathscr{K} \cup \wp(\mathscr{K}^*)$. In Fig. 7 we report an example of application of ReduceCycles algorithm. As example note that, by applying Alg. 1 to the automaton for $\mathscr{S}_{\mathtt{str}}$ in Fig. 6, we generate the CFG $\mathscr{P}_{\mathtt{str}}$, depicted in Fig. 5b. It is worth noting that the CFG obtained so far may contain abstract expressions on edges, hence edges may represent an infinite collection of statements. At this point, we need to approximate these edges for making it possible to analyze the CFG.

Making the Code Analyzable: Abstracting the CFG. Let us recall that we have to perform the analysis ρ also on the resulting code, in order to continue the static analysis. Hence, as observed before, we have to combine the code abstraction corresponding to the generated (abstract) CFG with the code abstraction induced by the semantic abstraction ρ, i.e., $\overline{\Upsilon}[\rho]$, which models, as code abstraction, the analysis.

First of all, we have to formally characterize the abstraction η induced by the construction of the CFG given above, namely we characterize how the construction abstracts together different predicates. Let

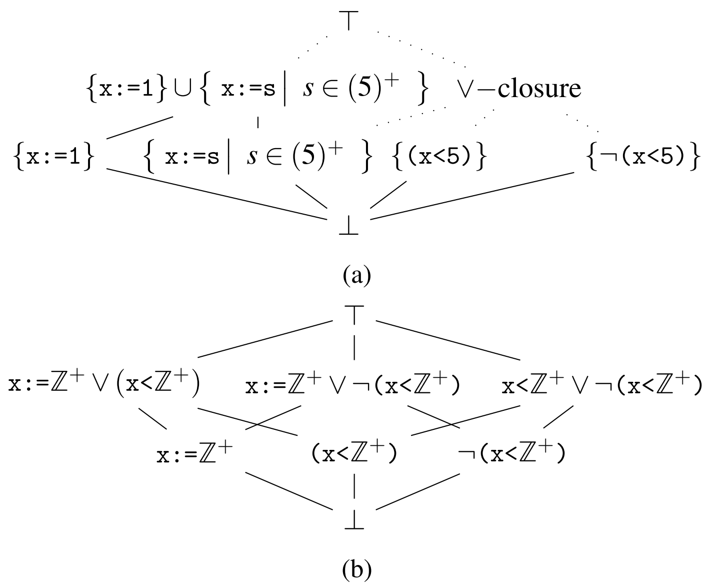


Figure 8: (a) Code abstraction $\eta^{\mathscr{P}_{\texttt{str}}}$ w.r.t. the CFG reported in Fig. 5b, (b) Code abstraction $\overline{\Upsilon}[\rho_{\mathscr{CS}}]^{\mathscr{P}_{\texttt{str}}}$

us build a code abstraction starting from the CFG $\mathscr{P} = \langle Nodes, Edges \rangle$ built in Alg. 1: In particular, let $Merge \overset{\text{def}}{=} \left\{ \left\{ \varphi \in \Psi \;\middle|\; \langle \ell', \varphi, \ell'' \rangle \in Edges \right\} \;\middle|\; \ell', \ell'' \in Nodes \right\} \subseteq \Psi$ be the set of collections of predicates between any pair of states in the CFG, we define

$$\eta^{\mathscr{P}}(\wp(\Psi)) \overset{\text{def}}{=} \wp(\{\, X \in Merge \mid \forall Y \in Merge \smallsetminus \{X\}. X \cap Y = \varnothing \,\}) \in uco(\wp(\Psi)) \tag{2}$$

Note that, this abstraction, being characterized starting from the CFG is defined only in terms of a finite subset of Ψ, namely on the predicates in the given CFG, i.e., $\Psi^{\mathscr{P}} \overset{\text{def}}{=} \Psi \cap \{\, \varphi \mid \langle \ell', \varphi, \ell'' \rangle \in Edges \,\}$.
In the example, $\Psi^{\mathscr{P}_{\texttt{str}}(\wp(\Psi))} = \{\{\, \texttt{x:=s} \mid s \in (5)^{+} \,\}, \{\texttt{x:=1}\}, \{\texttt{(x<5)}\}, \{\neg\texttt{(x<5)}\}\}$, hence we have that $\eta^{\mathscr{P}_{\texttt{str}}} = \wp(\Psi^{\mathscr{P}_{\texttt{str}}})$, being $\Psi^{\mathscr{P}_{\texttt{str}}}$ already a partition. In Fig. 8a this abstraction is partially depicted.

Finally, we need to satisfy Eq. 1 (completeness) between the code abstraction $\eta^{\mathscr{P}}$, built so far, and the static analysis, modeled as a semantic abstraction ρ, performing ρ_S (introduced above) on strings. Clearly we have no guarantee that $\eta^{\mathscr{P}}$ satisfies Eq. 1, hence, we have to (further) abstract the CFG in order to guarantee completeness w.r.t. the performed static analysis, namely in order to make it possible to perform the given static analysis on the code in the generated CFG. As observed in the previous section, in order to force completeness, we have to combine the desired abstraction $\eta^{\mathscr{P}}$ on predicates, with the code abstraction $\overline{\Upsilon}[\rho]$. Formally, in order to allow this operation, since $\eta^{\mathscr{P}}$ is defined on $\Psi^{\mathscr{P}}$, we have to restrict also $\overline{\Upsilon}[\rho]$ on $\Psi^{\mathscr{P}}$ (denoted $\overline{\Upsilon}[\rho]^{\mathscr{P}}$). This abstraction is obtained by intersecting the meaning of each one of its elements (i.e., its concretization) with the set of predicates in the CFG. In the running example, we have to compute $\overline{\Upsilon}[\rho_{\mathscr{CS}}]^{\mathscr{P}_{\texttt{str}}}$, which is the code abstraction induced by the Sign on the predicates in $\mathscr{P}_{\texttt{str}}$. For instance, all the predicates in $\{\, \texttt{x:=s} \mid s \in (5)^{+} \,\}$ and the predicate $\texttt{x:=1}$ cannot be distinguished when integers are abstracted by observing only their signs, hence the resulting abstraction is depicted in Fig. 8b, where the abstract predicate $\texttt{x:=}\mathbb{Z}^{+}$ corresponds, in the concrete, to the set of predicates $\{\, \texttt{x:=s} \mid s \in (5)^{+} \,\} \cup \{\texttt{x:=1}\}$, while $\texttt{x<}\mathbb{Z}^{+}$ and $\neg\texttt{(x<}\mathbb{Z}^{+}\texttt{)}$ correspond, respectively, to $\{\texttt{(x<5)}\}$ and to $\{\neg\texttt{(x<5)}\}$ (all the other elements corresponds to $\bot$).

Finally, we aim at building a code abstraction which can be interpreted by the initial abstract interpreter ρ, namely, that satisfies Eq. 1. By Th. 3.3 such an abstraction is $\overline{\eta}^{\mathscr{P}}_{\uparrow} = \overline{\Upsilon}[\rho]^{\mathscr{P}} \circ \eta^{\mathscr{P}}$.

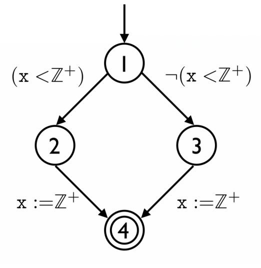


Figure 9: Abstract CFG generated by abstracting $\mathscr{P}_{\mathtt{str}}$ by means of $\overline{\eta}_{\uparrow}^{\mathscr{P}_{\mathtt{str}}}$

Corollary 4.1 *Let $\rho \in uco(\mathbb{V})$ be additive. Then the code abstraction $\overline{\eta}_{\uparrow}^{\mathscr{P}} = \overline{\Upsilon}[\rho]^{\mathscr{P}} \circ \eta^{\mathscr{P}} \in uco(\Psi^{\mathscr{P}})$ is complete w.r.t. the semantic abstraction ρ, i.e., it satisfies Eq. 1.*

Hence, in our example, the code abstraction $\overline{\eta}_{\uparrow}^{\mathscr{P}_{\mathtt{str}}} = \overline{\Upsilon}[\rho_{\mathscr{CS}}]^{\mathscr{P}_{\mathtt{str}}} \circ \eta^{\mathscr{P}_{\mathtt{str}}}$ satisfies Eq. 1. In particular, we can observe that $\overline{\eta}_{\uparrow}^{\mathscr{P}_{\mathtt{str}}} = \overline{\Upsilon}[\rho_{\mathscr{CS}}]^{\mathscr{P}_{\mathtt{str}}}$. Finally, we have to abstract the CFG $\mathscr{P}$, previously generated, by applying $\overline{\eta}_{\uparrow}^{\mathscr{P}}$ to each edge of the CFG. In our example, the so far resulting abstract CFG is reported in Fig. 9, where the abstract CFG generated by abstracting $\mathscr{P}_{\mathtt{str}}$ by means of $\overline{\eta}_{\uparrow}^{\mathscr{P}_{\mathtt{str}}}$ is depicted.

A Taste of Implementation. A static analyzer based on finite state automata is available at [2]. Moreover, we have implemented Alg. 1 in order to validate our approach[13]. The implementation of a static analysis of abstract CFGs is in an early stage development and it is left as future work. Nevertheless, it is able to parse executable automata and to abstract them into abstract CFGs, as we have previously described. In order to make these abstract CFGs effectively analyzable, we are currently extending the static analyzer, and the underlying abstract interpreter, to parse, and thus analyze, also abstract predicates.

5 Conclusion

We conclude by highlighting the value, in the context of static analysis, of the framework presented in this paper. What we propose here is a precision improvement of [3], an analysis that attacks an extremely hard problem in static program analysis by abstract interpretation, since the standard static analysis assumption (i.e., the program code we want to analyze must be static) is broken when we have to deal with string-to-code statements. In [3], we have shown that even without this assumption, it is still possible for static analysis to semantically analyze dynamically mutating code in a meaningful and sound way. It has been the very first proof of concept for a sound static analysis for self-modifying code based on bounded reflection for a high-level script-like programming language. In this paper, we improve this approach by characterizing code transformations that do not lose precision w.r.t. a fixed abstract semantics/analysis of the code. The idea we develop consists of embedding the property to analyze in the code transformation in order to make the property analysis work also on the transformed code (as it happens in dynamic code analysis). Hence, the main contribution is to make even more precise the first truly *dynamic static analyzer*, which has the feature to keep the analysis going on, even when code is dynamically built.
Clearly, the framework improved here is still at an early stage and surely there is much work to do, not only for the presented algorithm and the implementation, which has clearly to be further developed but

[13] Available at
`https://github.com/SPY-Lab/java-fsm-library/tree/abstract-parser`

also for making the approach more precise and general. As far as the algorithm is concerned we have not explicitly provided soundness and completeness proofs or discussions. In particular, completeness holds under decidable hypotheses (the input automaton has to recognize only executable strings), here only briefly treated, and therefore these aspects need further formal development.
On the other hand, a direction for improving precision can be that of integrating the proposed static analysis in a hybrid solution, by using, for instance, taint analysis (or other dynamic analyses) for driving when to apply static analysis, or considering more advanced forms of automata-based domains for abstracting strings, such as the one reported in [23]. Finally, we have considered only `eval` as a string-to-code statement, while there are other ways, for dynamically executing code built out of strings, that should be investigated. However, we strongly believe that the same approach used for `eval`, could be easily applied to any other string-to-code statement. Moreover, we believe that this framework could be instantiated in order to deal with other forms of code transformations, maybe by considering more general CFG abstractions.

From a more theoretical point of view, interesting future works consist of exploiting the proposed approach for analyzing code in order to investigate, on dynamic languages, several application contexts where static analysis by abstract interpretations has been exploited. First of all, we could trace (abstract) flows of information during execution [14, 20, 18, 19, 12, 11, 10] in order to tackle different security issues, such as the detection of (abstract) code injections [6, 5] or the formal characterization of dynamic code obfuscators and of their potency [9, 13]. Moreover, the ability to analyze malware code could be exploited for extracting code properties which could be used for analyzing code similarity [24], a technique useful for instance to identify or at least classify malicious code.

References

[1] Roberto Amadini, Graeme Gange, François Gauthier, Alexander Jordan, Peter Schachte, Harald Søndergaard, Peter J. Stuckey & Chenyi Zhang (2018): *Reference Abstract Domains and Applications to String Analysis*. *Fundam. Informaticae* 158(4), pp. 297–326, doi:10.3233/FI-2018-1650.

[2] Vincenzo Arceri & Isabella Mastroeni (2019): *An Automata-based Abstract Semantics for String Manipulation Languages*. In Alexei Lisitsa & Andrei P. Nemytykh, editors: *Proceedings Seventh International Workshop on Verification and Program Transformation, VPT@Programming 2019, Genova, Italy, 2nd April 2019*, *EPTCS* 299, pp. 19–33, doi:10.4204/EPTCS.299.5.

[3] Vincenzo Arceri & Isabella Mastroeni (2021): *Analyzing Dynamic Code: A Sound Abstract Interpreter for* Evil *Eval*. *ACM Trans. Priv. Secur.* 24(2), pp. 10:1–10:38, doi:10.1145/3426470.

[4] Vincenzo Arceri, Isabella Mastroeni & Sunyi Xu (2020): *Static Analysis for ECMAScript String Manipulation Programs*. *Appl. Sci.* 10, p. 3525, doi:10.3390/app10103525.

[5] Musard Balliu & Isabella Mastroeni (2010): *A Weakest Precondition Approach to Robustness*. *Trans. Comput. Sci.* 10, pp. 261–297, doi:10.1007/978-3-642-17499-5_11.

[6] Samuele Buro & Isabella Mastroeni (2018): *Abstract Code Injection - A Semantic Approach Based on Abstract Non-Interference*. In Isil Dillig & Jens Palsberg, editors: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, *Lecture Notes in Computer Science* 10747, Springer, pp. 116–137, doi:10.1007/978-3-319-73721-8_6.

[7] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim & Kyung-Goo Doh (2006): *A Practical String Analyzer by the Widening Approach*. In Naoki Kobayashi, editor: *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*, *Lecture Notes in Computer Science* 4279, Springer, pp. 374–388, doi:10.1007/11924661_23.

[8] Giulia Costantini, Pietro Ferrara & Agostino Cortesi (2015): *A suite of abstract domains for static analysis of string values*. *Softw. Pract. Exp.* 45(2), pp. 245–287, doi:10.1002/spe.2218.

[9] Roberto Giacobazzi, Neil D. Jones & Isabella Mastroeni (2012): *Obfuscation by partial evaluation of distorted interpreters*. In Oleg Kiselyov & Simon J. Thompson, editors: *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*, ACM, pp. 63–72, doi:10.1145/2103746.2103761.

[10] Roberto Giacobazzi & Isabella Mastroeni (2004): *Proving Abstract Non-interference*. In Jerzy Marcinkowski & Andrzej Tarlecki, editors: *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, *Lecture Notes in Computer Science* 3210, Springer, pp. 280–294, doi:10.1007/978-3-540-30124-0_23.

[11] Roberto Giacobazzi & Isabella Mastroeni (2010): *Adjoining classified and unclassified information by abstract interpretation*. *J. Comput. Secur.* 18(5), pp. 751–797, doi:10.3233/JCS-2009-0382.

[12] Roberto Giacobazzi & Isabella Mastroeni (2010): *A Proof System for Abstract Non-interference*. *J. Log. Comput.* 20(2), pp. 449–479, doi:10.1093/logcom/exp053.

[13] Roberto Giacobazzi & Isabella Mastroeni (2012): *Making Abstract Interpretation Incomplete: Modeling the Potency of Obfuscation*. In Antoine Miné & David Schmidt, editors: *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, *Lecture Notes in Computer Science* 7460, Springer, pp. 129–145, doi:10.1007/978-3-642-33125-1_11.

[14] Roberto Giacobazzi & Isabella Mastroeni (2018): *Abstract Non-Interference: A Unifying Framework for Weakening Information-flow*. *ACM Trans. Priv. Secur.* 21(2), pp. 9:1–9:31, doi:10.1145/3175660.

[15] Roberto Giacobazzi & Elisa Quintarelli (2001): *Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking*. In Patrick Cousot, editor: *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, *Lecture Notes in Computer Science* 2126, Springer, pp. 356–373, doi:10.1007/3-540-47764-0_20.

[16] Simon Holm Jensen, Peter A. Jonsson & Anders Møller (2012): *Remedying the eval that men do*. In Mats Per Erik Heimdahl & Zhendong Su, editors: *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, ACM, pp. 34–44, doi:10.1145/2338965.2336758.

[17] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann & Ben Hardekopf (2014): *JSAI: a static analysis platform for JavaScript*. In Shing-Chi Cheung, Alessandro Orso & Margaret-Anne D. Storey, editors: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, ACM, pp. 121–132, doi:10.1145/2635868.2635904.

[18] Isabella Mastroeni (2013): *Abstract interpretation-based approaches to Security - A Survey on Abstract Non-Interference and its Challenging Applications*. In Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh & John Hatcliff, editors: *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013*, *EPTCS* 129, pp. 41–65, doi:10.4204/EPTCS.129.4.

[19] Isabella Mastroeni & Durica Nikolic (2010): *Abstract Program Slicing: From Theory towards an Implementation*. In Jin Song Dong & Huibiao Zhu, editors: *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, *Lecture Notes in Computer Science* 6447, Springer, pp. 452–467, doi:10.1007/978-3-642-16901-4_30.

[20] Isabella Mastroeni & Damiano Zanardini (2017): *Abstract Program Slicing: An Abstract Interpretation-Based Approach to Program Slicing*. *ACM Trans. Comput. Log.* 18(1), pp. 7:1–7:58, doi:10.1145/3029052.

[21] Nikos Mavrogiannopoulos, Nessim Kisserli & Bart Preneel (2011): *A taxonomy of self-modifying code for obfuscation*. *Comput. Secur.* 30(8), pp. 679–691, doi:10.1016/j.cose.2011.08.007.

[22] Antoine Miné (2013): *Static analysis by abstract interpretation of concurrent programs. (Analyse statique par interprétation abstraite de programmes concurrents).* Available at `https://tel.archives-ouvertes.fr/tel-00903447`.

[23] Luca Negrini, Vincenzo Arceri, Pietro Ferrara & Agostino Cortesi (2021): *Twinning Automata and Regular Expressions for String Static Analysis*. In Fritz Henglein, Sharon Shoham & Yakir Vizel, editors: *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, *Lecture Notes in Computer Science* 12597, Springer, pp. 267–290, doi:10.1007/978-3-030-67067-2_13.

[24] Mila Dalla Preda, Roberto Giacobazzi, Arun Lakhotia & Isabella Mastroeni (2015): *Abstract Symbolic Automata: Mixed syntactic/semantic similarity analysis of executables*. In Sriram K. Rajamani & David Walker, editors: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, ACM, pp. 329–341, doi:10.1145/2676726.2676986.

[25] Gregor Richards, Christian Hammer, Brian Burg & Jan Vitek (2011): *The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications*. In Mira Mezini, editor: *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, *Lecture Notes in Computer Science* 6813, Springer, pp. 52–78, doi:10.1007/978-3-642-22655-7_4.

[26] Robert Endre Tarjan (1972): *Depth-First Search and Linear Graph Algorithms*. *SIAM J. Comput.* 1(2), pp. 146–160, doi:10.1137/0201010.

[27] Reinhard Wilhelm, Helmut Seidl & Sebastian Hack (2013): *Compiler Design - Syntactic and Semantic Analysis*. Springer, doi:10.1007/978-3-642-17540-4.